Introduzione

Questo lavoro ha l'obiettivo di implementare sul simulatore Alchemist il modello Spatial Tuples.

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.

Per creare l'incarnazione sarà necessario definire una base del modello ad agenti sul quale poi innestare Spatial Tuples.

Indice

Introduzione **i**

1 Alchemist **1**
1.1 Il meta-modello 1
1.2 Scrivere una simulazione 3

2 Agenti **7**
2.1 Agenti BDI 7
2.2 tuProlog 8

3 Spatial Tuples **11**
3.1 Tuple spaces 11
3.2 Modello Spatial Tuples 12

4 Progetto **13**
4.1 Mapping dei modelli 13
4.2 Definizione del linguaggio 14
4.3 Definizione incarnazione 17
4.3.1 Spostamento del nodo 20
4.3.2 Aggiornamento belief base 21
4.3.3 Implementazione Spatial Tuples 22
4.4 Simulazione 24
4.4.1 Raccolta dei dati 32

Bibliografia **33**

Elenco delle figure

1.1 Illustrazione meta-modello di Alchemist 2
1.2 Illustrazione modello reazione di Alchemist 3

Codici sorgenti

1.1 Incarnazione 4
1.2 Variabili simulazione 4
1.3 Environment 4
1.4 Default environment 5
1.5 Posizioni 5
1.6 Funzione linking-rule 5
1.7 Default linking-rule 5
1.8 Disposizione nodi e reazioni associate 6
4.1 Libreria agenti 16
4.2 Simple Agent Reasoning Cycle 19
4.3 Implementazione spostamento nodo 21
4.4 Implementazione piani per l'aggiunta e rimozione di belief . . 22
4.5 Simulazione modello Spatial Tuples con modello di coordinazione breadcrumbs 24
4.6 Teoria agente Hansel 25
4.7 Teoria agente Gretel 27
4.8 Teoria per gli spazi di tuple 31

Capitolo 1

Alchemist

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

1.1 Il meta-modello

Il meta-modello di Alchemist può essere compreso osservando 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 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***.

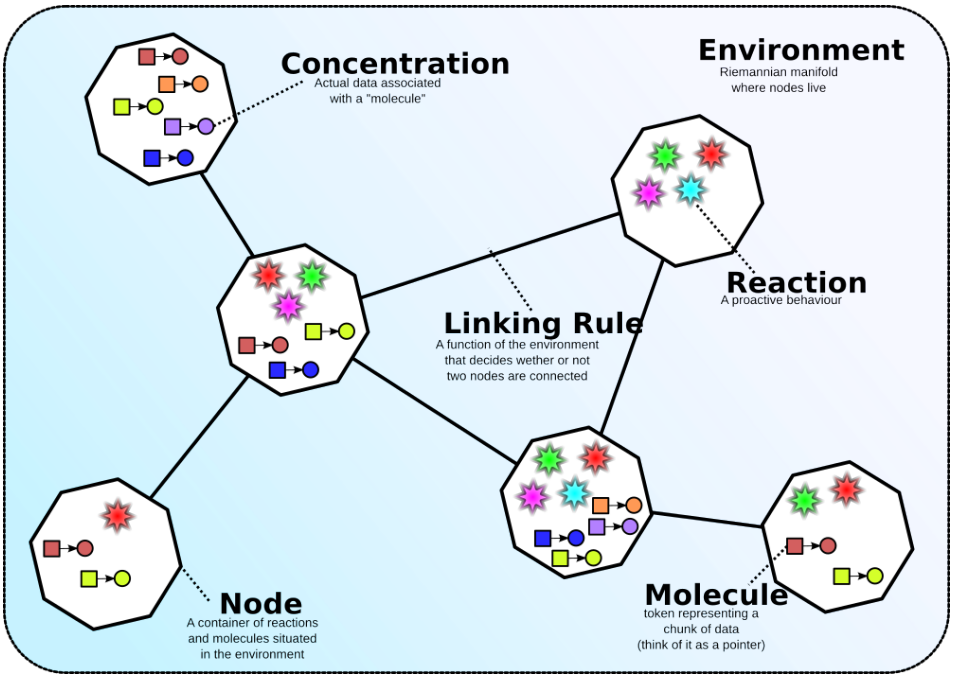

Figura 1.1: Illustrazione meta-modello di Alchemist

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.

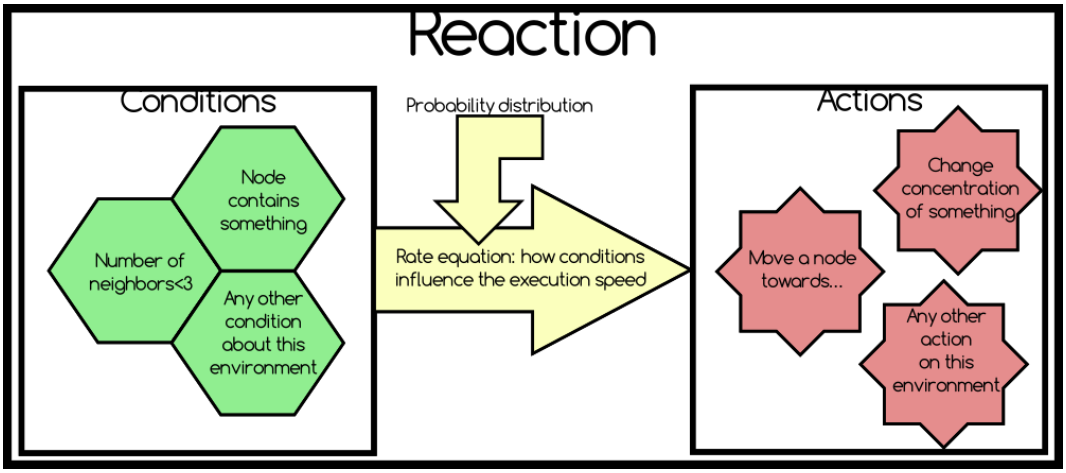

Figura 1.2: Illustrazione modello reazione di Alchemist

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: Posizioni

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 quello mostrato nel Codice sorgente 1.8.

```
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 beliefs 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 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

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 dalla proprietà spaziali del coordinamento degli agenti;
- lo spazio di tuple può essere concepito come un livello virtuale che aumenta la realtà fisica.

Questo modello supporta la consapevolezza dello spazio e la coordinazione basata sullo spazio dell'agente in scenari di calcolo pervasivo.

3.1 Tuple spaces

Uno spazio di tuple è un implementazione del paradigma di memoria associativa per l'elaborazione parallela/distribuita. Fornisce un repository di tuple al quale è possibile accedere contemporaneamente. Lo spazio di tuple può essere considerato come una forma di memoria condivisa distribuita. Ci sono due categorie di utilizzatori dello spazio di memoria: chi scrive e chi legge. I primi aggiungono tuple allo spazio di memoria condiviso mentre i secondi rimuovono quelle che corrispondono ad un determinato template.

3.2 Modello Spatial Tuples

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 informazioni 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 una tupla che risponde alla richiesta. 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.

Spatial Tuples può essere utile nei domini applicativi in cui la reatlà è aumentata o mista, ovvero in scenari in cui gli esseri umani si spostano all'interno di un ambiente fisico aumentato e devono coordinarsi con altri utenti o agenti di qualsiati tipo (situati o non situati).

Capitolo 4

Progetto

Il progetto, come descritto nell'introduzione, ha come obiettivo l'implementazione del meta-modello di Alchemist attraverso la definizione di un incarnazione che modelli Spatial Tuples attraverso 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'.

4.1 Mapping dei modelli

Il primo passo nell'evoluzione del progetto è stata l'analisi del mapping tra il meta-modello di Alchemist e il modello ad agenti, 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 complessi 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, ovvero il ciclo di ragionamento dell'agente.

4.2 Definizione del linguaggio

Dopo aver effettuato l'analisi dei vari mapping si è passati alla realizzazione di un linguaggio per l'incarnazione. Lo scopo di questa operazione è quella di fornire all'utilizzatore finale, ovvero il programmatore dell'agente, completa libertà per la definizione della teoria dell'agente ma che al tempo stesso permetta di gestire il comportamento dell'agente all'interno dell'ambiente di simulazione.

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 beliefs 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 invocati per reagire ad un evento. I piani che fanno parte del primo tipo sono:

- **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

- **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 inserire il modello Spatial Tuples all'interno dell'incarnazione, sono state analizzate le sue parti per decidere l'implementazione delle primitive da fornire al programmatore dell'agente, permettergli di far comunicare gli

agenti con gli spazi di tuple. Sono stati quindi definiti i piani che rispecchiano le primitive **out(t)**, **rd(tt)**, **in(tt)** definte dal modello Spaital Tuples e un piano per la ricezione delle tuple prelevate. Mentre i primi tre sono piani che innescano un'azione, il quarto è un piano invocato per reagire alle operazioni di *read* e *write*, e che il programmatore dell'agente sceglie come implementare.

- **writeTuple(T).** Scrive una tupla T nello spazio di tuple più vicino.
- **readTuple(TT).** Cerca negli spazi di tuple del suo vicinato una tupla che corrisponda al template TT.
- **takeTuple(TT)**. Stessa procedura di readTuple ma se la tuple viene trovata è rimossa dallo spazio di tuple.
- **onResponseMessage(T).** Piano invocato dal sistema in cui viene passata la tupla recuperata con gli eventi scatenati da readTuple e writeTuple.

Le implementazioni dei piani che scatenatno un evento (mostrata nel Codice sorgente 4.1) sono racchiuse all'interno della libreria caricata nel motore tuProlog di ogni agente.

```
1   addBelief(B) :-
2     assertz(belief(B)),
3     assertz(added_belief(B)).
4
5   removeBelief(B) :-
6     retract(belief(B)),
7     assertz(removed_belief(B)).
8
9  writeTuple(T) :-
10    assertz(write(T)).
11
12 readTuple(TT) :-
```

```
13      assertz(read(TT)).
14
15  takeTuple(TT) :-
16      assertz(take(TT)).
```

Codice sorgente 4.1: Libreria agenti

4.3 Definizione incarnazione

Lo sviluppo dell'incarmazione 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, 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) e valori di velocità e direzione per lo spostamento.

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 azioni e le condizioni che devono verificarsi per far avvenire le azioni. Come proprietà della classe è presente solo una stringa che memorizza il nome dell'agente.
La creazione delle **condizioni** è stata definita 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 tramite un parametro. Nell'azione 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.

Per le azioni 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'interfaccia *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 4.1 nella quale sono descritti i piani che definiscono il linguaggio e invocabili nella teoria dell'agente.
La classe inoltre contiene i metodi per il comportamento dell'agente nel ciclo di ragionamento e quelli per l'inizializzazione. Quest'ultima è identificata come la prima esecuzione del ciclo di ragionamento dell'agente e che quindi racchiude delle istruzioni basilari per la sua corretta operatività.

Estendendo la classe appena descritta è stata creata **SimpleAgent** che racchiude tutte le caratteristiche principali di un agente descritte precedentememte. Alla sua istanziazione, 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'im-

plementazione del primo è mostrata 4.2 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 'bob' verrà creato l'agente 'cloned_bob').

```
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.positionUpdate();
10
11     this.getBeliefBaseChanges();
12
13     this.notifyBeliefBaseChanges();
14  }
```

Codice sorgente 4.2: Simple Agent Reasoning Cycle

Se l'agente non è inizializzato vengono invocati in sequenza i metodi *inizializeAgent* e *initReasoning*. Il primo inizializza l'agente, impostando nella teoria alcuni belief tra cui quello per la posizione e per lo spostamento del nodo. Il secondo invece effettua la chiamata al piano 'init' eseguendo la configurazione iniziale definita dal programmatore dell'agente.

Il metodo ***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.

4.3.1 Spostamento del nodo

Per realizzare lo spostamento sono stati utilizzati i 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 della classe *AgentsContainerNode*. Oltre a queste proprietà sono presenti 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. L'implementazione del metodo è mostrata nel Codice sorgente 4.3. 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.

```
1public void changeNodePosition(final Time time) {
2  final Position currPos = this.getNodePosition();
3  final double radAngle =  this.nodeDirectionAngle;
4
5  // radius = space covered = time spent * speed
6  final double radius = (time.toDouble() -
    this.lastUpdataTime.toDouble()) * this.nodeSpeed;
7
8  final double x = currPos.getCoordinate(0) +
    radius * Math.cos(radAngle);
9
10  final double y = currPos.getCoordinate(1) +
    radius * Math.sin(radAngle);
11
12  this.environment.moveNodeToPosition(
13     this, this.environment.makePosition(x, y));
14
15  this.lastUpdataTime = time;
16 }
```

Codice sorgente 4.3: Implementazione spostamento nodo

4.3.2 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.

```
1  addBelief(B) :-
2    assertz(belief(B)),
3    assertz(added_belief(B)).
4
5  removeBelief(B) :-
6    retract(belief(B)),
7    assertz(removed_belief(B)).
```

Codice sorgente 4.4: Implementazione piani per l'aggiunta e rimozione di belief

4.3.3 Implementazione Spatial Tuples

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 dell'istanza 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 3.2, 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 di Spatial Tuples è necessario definire un metodo nella classe *AbstracAgent* che consente agli agenti, durante il loro ciclo di ragionamento, di recuperare i beliefs 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*, cerca 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.

4.4 Simulazione

Per testare l'incarnazione creata è stato utilizzato il pattern di coordinazione '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 4.5.

```
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: [1,2,2,0.2]}
15 programs:
16    -
17      - time-distribution: 1
18        program: "gretel"
19
```

```
20  - in: {type: Circle, parameters: [1000,2,2,4.5]}
21  programs:
22     -
23       - time-distribution: 1
24         program: "blackboard"
```

Codice sorgente 4.5: 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(0.025,4.8)),
5       addBelief(counter(0,0)),
6       addBelief(spiralCorner(0.08)),
7       addBelief(stopped(false)).
8
9   onAddBelief(position(X,Y)) :-
10      belief(stopped(false)),
11      removeBelief(counter(C1,C2)),
12      handlePosition(C1,C2,X,Y).
13
14  onAddBelief(position(X,Y)) :-
15      true.
16
17  handlePosition(C1,C2,X,Y) :-
18      C1 < 300,
19      C2 < 10,
20      C1N is C1 + 1,
21      C2N is C2 + 1,
```

```
22      addBelief(counter(C1N,C2N)),
23      writeTuple(breadcrumb(hansel,here)).
24
25  handlePosition(C1,C2,X,Y) :-
26      C1 < 300,
27      C1N is C1 + 1,
28      C2 >= 10,
29      addBelief(counter(C1N,0)),
30      removeBelief(movement(S,D)),
31      belief(spiralCorner(V)),
32      D1 is D + V,
33      addBelief(movement(S,D1)).
34
35  handlePosition(C1,C2,X,Y) :-
36      C1 >= 300,
37      C2N is C2 + 1,
38      addBelief(counter(0,C2N)),
39      removeBelief(spiralCorner(V)),
40      TEMP is V * 0.30,
41      V1 is V + TEMP,
42      addBelief(spiralCorner(V1)).
43
44  onAddBelief(distance(A,ND,OD)) :-
45      true.
46
47  onAddBelief(distance(gretel,ND)) :-
48      removeBelief(stopped(false)),
49      addBelief(stopped(true)),
50      removeBelief(movement(_,D)),
51      addBelief(movement(0,D)).
52
```

```
53  onAddBelief(distance(A,ND)) :-
54      true.
55
56  onAddBelief(movement(_,_)) :-
57      true.
58
59  onRemoveBelief(movement(_,_)) :-
60      true.
61
62  onAddBelief(counter(C1,C2)) :-
63      true.
64
65  onRemoveBelief(counter(C1,C2)) :-
66      true.
67
68  onAddBelief(spiralCorner(V)) :-
69      true.
70
71  onRemoveBelief(spiralCorner(V)) :-
72      true.
73
74  onResponseMessage(msg(stop(hansel),X,Y)) :-
75      removeBelief(movement(_,D)),
76      addBelief(movement(0,D)),
77      writeTuple(stop(gretel)).
```

Codice sorgente 4.6: Teoria agente Hansel

```
1   init :-
2       removeBelief(movement(_,_)),
3       randomDirection(D),
4       addBelief(movement(0.025,D)),
```

```
5       takeTuple(breadcrumb(hansel,here)),
6       randomSteps(P),
7       addBelief(counterStep(P)),
8       addBelief(stopped(false)).
9
10  onAddBelief(position(X,Y)) :-
11      belief(stopped(false)),
12      removeBelief(counterStep(S)),
13      handlePosition(S,X,Y),
14      takeTuple(breadcrumb(hansel,here)).
15
16  onAddBelief(position(X,Y)) :-
17      true.
18
19  handlePosition(S,X,Y) :-
20      S > 0,
21      S1 is S - 1,
22      addBelief(counterStep(S1)).
23
24  handlePosition(S,X,Y) :-
25      S =< 0,
26      randomSteps(P),
27      addBelief(counterStep(P)),
28      removeBelief(movement(_,D)),
29      randomDirection(RD),
30      D1 is D - RD,
31      addBelief(movement(0.025,D1)).
32
33  onAddBelief(distance(hansel,ND,OD)) :-
34      ND < 0.4,
35      removeBelief(stopped(false)),
```

```
36        addBelief(stopped(true)),
37        removeBelief(movement(_,D)),
38        addBelief(movement(0,D)).
39
40    onAddBelief(distance(A,ND,OD)) :-
41        true.
42
43    onAddBelief(distance(A,ND)) :-
44        true.
45
46    onAddBelief(movement(_,_)) :-
47        true.
48
49    onRemoveBelief(movement(_,_)) :-
50        true.
51
52    onAddBelief(counterStep(C)) :-
53        true.
54
55    onRemoveBelief(counterStep(C)) :-
56        true.
57
58    onResponseMessage(msg(breadcrumb(hansel,here),X,Y
      )) :-
59        removeBelief(counterStep(_)),
60        addBelief(counterStep(15)),
61        removeBelief(movement(_,D)),
62        addBelief(movement(0.05,D)),
63        changeDirection(X,Y).
64
65    changeDirection(X2,Y2) :-
```

```
66        belief(position(X1,Y1)),
67        DX is X2 - X1,
68        DY is Y2 - Y1,
69        calculateAtan(DY,DX,RAD),
70        removeBelief(movement(S,D)),
71        addBelief(movement(S,RAD)).
72
73    calculateAtan(DY,DX,RAD) :-
74        DX > 0,
75        RAD is atan(DY / DX).
76
77    calculateAtan(DY,DX,RAD) :-
78        DX < 0,
79        DY >= 0,
80        TMP is atan(DY / DX),
81        RAD is TMP + 3.14.
82
83    calculateAtan(DY,DX,RAD) :-
84        DX < 0,
85        DY < 0,
86        TMP is atan(DY / DX),
87        RAD is TMP - 3.14.
88
89    calculateAtan(DY,DX,RAD) :-
90        DX == 0,
91        DY > 0,
92        RAD is 3.14 / 2.
93
94    calculateAtan(DY,DX,RAD) :-
95        DX == 0,
96        DY < 0,
```

```
97        RAD is -3.14 / 2.
98
99    randomSteps(R) :-
100       rand_int(50,R).
101
102   randomDirection(R) :-
103       rand_float(X),
104       TMP is X * 3.14,
105       R is TMP - 1.57.
```

Codice sorgente 4.7: Teoria agente Gretel

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

Codice sorgente 4.8: Teoria per gli spazi di tuple

L'agente Hansel nella ricerca dell'agente Gretel esegue il seguente algoritmo. L'esecuzione dell'algoritmo è ciclica.

(1) ho trovato l'agente Hansel?

(2) si: imposto la velocità a 0 e aggiungo il belief stopped

(3) no: ho trovato una briciola?

    (4) si: resetto il contatore dei passi e seguo la direzione della briciola a velocità 0.05

    (5) no: ho fatto N passi senza trovare una briciola?

        (6) si: resetto il contatore dei passi, imposto una nuova direzione random e la velocità a 0.025

        (7) no: decremento il contatore dei passi

4.4.1 Raccolta dei dati

Per valutare il grado di successo della simulazione sono state effettuate analisi sui risultati delle simulazioni.

TODO:*inserire analisi dati una volta effettuate le simulazioni nel modo corretto*

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)