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 una '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***.

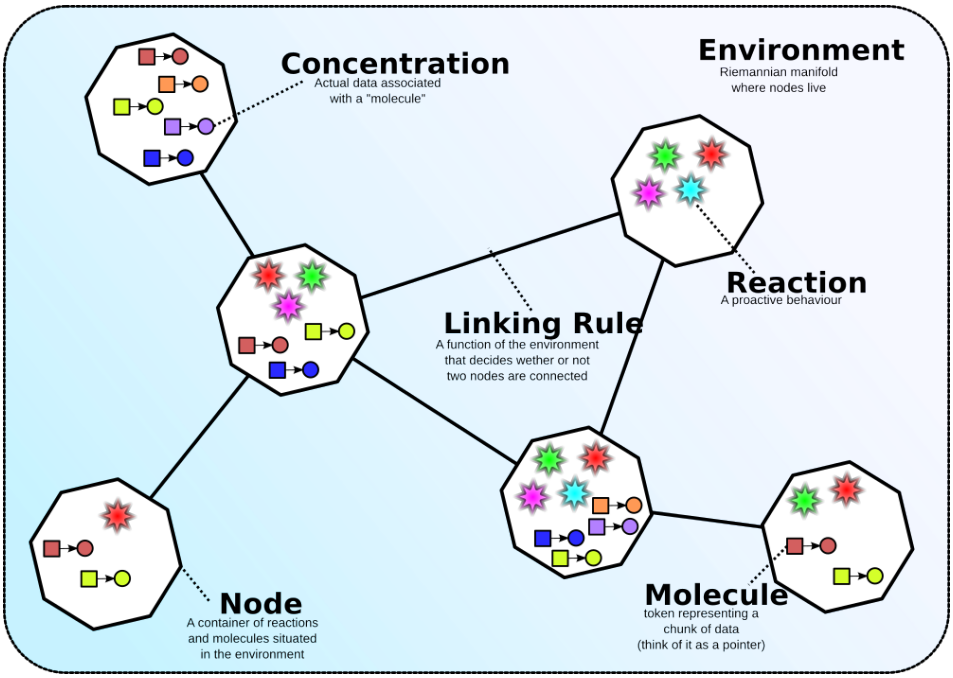

Figura 1.1: Illustrazione meta-modello di Alchemist

Una ***Reazione*** è un qualsiasi evento che può cambiare lo stato dell'environment ed è. definita tramite una lista di condizioni, 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 tasso 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.

Una ***Condizione*** è una funzione che prende come input l'environment corrente e restituisce come output un booleano e un numero. Se la condizione

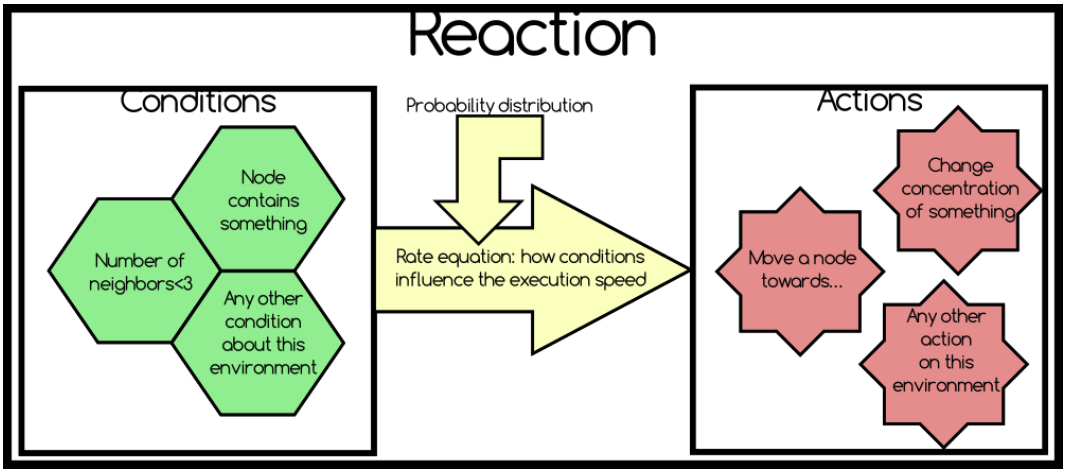

Figura 1.2: Illustrazione modello reazione di Alchemist

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

Nelle prossime occorrenze, dove si utilizza la chiave 'type' il valore associato 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

Con 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

POSITION
TODO POSITION??

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 1.5: 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 1.6: 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' si 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 ad esempio Point e Circle. La classe Circle necessata 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 ricevuto alla creazione della reazione e che può essere passato per instanziare condizioni e azioni). Un'esempio di displacements è il seguente.

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

Codice sorgente 1.7: 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.

I ***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.

Il ciclo di ragionamento di un agente avviene come descritto in figura 2.1.

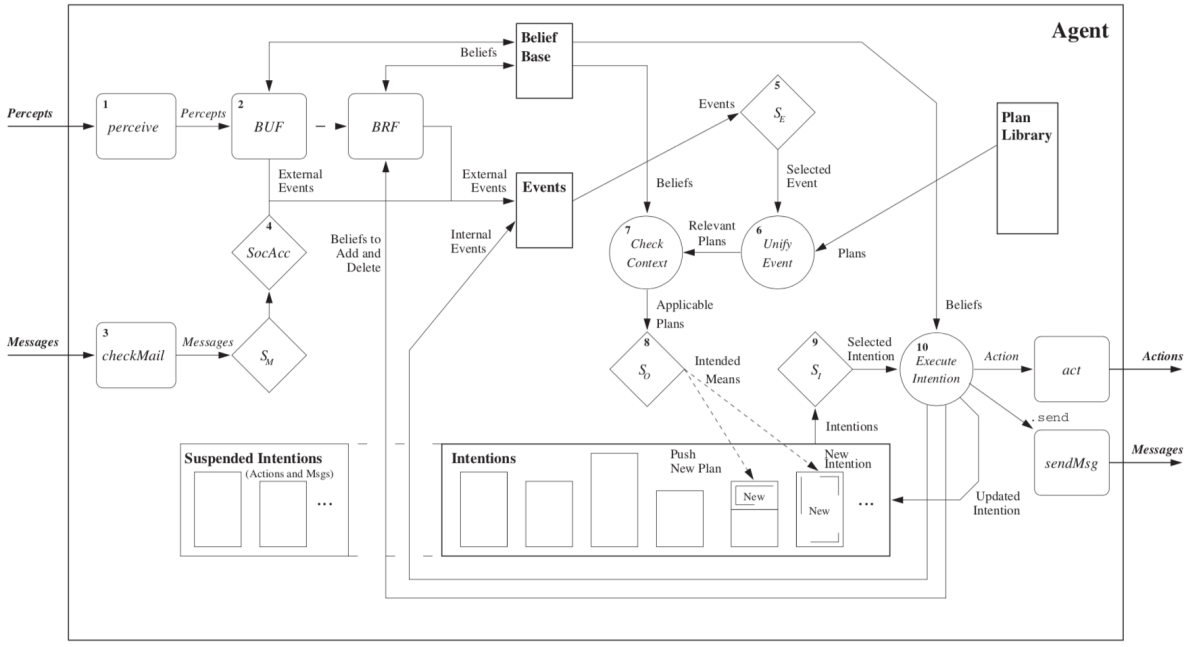

Figura 2.1: Ciclo di ragionamento di un agente

2.2 Ciclo di ragionamento

RIVEDERE IL PEZZO QUI SOTTO E CONTINUARLO....

La prima cosa che fa un agente nel ciclo di ragionamento è controllare l'ambiente per aggiornare i *beliefs* con lo stato attuale. Quindi è necessario che l'agente abbia dei componenti per percepire i cambiamenti dell'ambiente. Una volta ottenuta la lista dei cambiamenti percepiti è necessario aggiornare la *belief base* (che contiene l'insieme dei *beliefs* dell'utente allo stato attuale).

// **TODO CONTINUARE** la descrizione del ciclo di ragionamento dell'agente....

2.3 tuProlog

SPIEGARE TUPROLOG

Capitolo 3

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 gli agenti all'interno del simulatore.

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 mentre, 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. 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.
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.

3.2 Fasi di sviluppo

Il passo successivo è stato quello di stilare un piano di sviluppo per affrontare il problema attraverso step incrementali per permettere di semplificare l'implementazione di un modello complesso come quello ad agenti. Le fasi in cui è stato suddiviso il progetto sono state le seguenti:

1. Scambio di messaggi tra due agenti (ad esempio ping pong): inizialmente gli agenti possono anche risiedere nello stesso nodo ma poi l'implementazione deve valere indifferentemente dal posizionamento degli agenti;

2. Gestione del flusso di controllo attraverso l'inserimento di un operazione, come lo spostamento del nodo, prima di effettuare la risposta al messaggio. In seguito deve comprendere inoltre la percezione dell'agente verso l'ambiente;

3. Gestione del flusso condizionato inserendo una clausola per la quale lo scambio di messaggi debba avvenire o meno;

3.3 Implementazione

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 GLOBAL,LOCAL,NEIGHBORHOOD), *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.
La classe **PostmanAction** sovrascrive l'implementazione del metodo *execute* in modo da invocare, ad ogni evento lanciato dal simulatore, un metodo nel nodo che provvederà a prelevare i messaggi in uscita da ogni agente e recapitarli ai corretti destinatari.

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 (3.1) e una per l'agente Pong (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: Ping agent

```
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: Pong agent

Come si può notare, l'agente Ping ha al suo interno la definizione del piano 'init' che consente l'invio del primo messaggio dando 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 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 3.4 posiziona ogni agente su un nodo diveso. 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

Dopo aver realizzato lo scambio di messaggi si è passati alla realizzazione dello spostamento del nodo che ospita l'agente e alla percezione che l'agente può avere dell'ambiente nel quale è inserito.

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 compreso tra 0-360) e il momento della simulazione in cui è avvenuto l'ultimo aggiornamento della posizione. Oltre a queste proprietà sono stati inseriti anche i metodi per aggiornarne i valori: per la velocità il valore viene semplicemente sovrascritto mentre per la direzione si è deciso di fornire la possibilità di scegliere se settare un certo valore o aggiungere un delta al valore corrente. Prima di aggiornare direzione o velocità viene invocato il metodo per lo spostamento del nodo poichè è necessario attuare la variazione di posizione avvenuta dal precedente aggiornamento.
Il metodo che effettua lo spostamento vero e proprio del nodo è *changeNodePosition* che prende come parametro il tempo della simulazione corrente e viene chiamato ad ogni ciclo ci ragionamento per effettuare 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 moltiplicato per la velocità. La nuova posizione è un punto della circonferenza che viene individuato utilizzando l'angolo o direzione del nodo.

Per aggiungere la percezione dell'agente relativa alla posizone del nodo che lo ospita all'interno dell'ambiente è stata modificata la teoria dell'agente. È stato aggiunto un belief che rappresenta un 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, passata come parametro, risulta all'interno del bounding box. Dal ciclo di ragionamento, utilizzando la libreria 'alice.tuprolog', viene chiamato il piano *checkPosition* passandogli come termini le coordinate della posizone del nodo, le quali vengono inoltrate ai piani *isInFieldX* e *isInFieldY* che verificano che i valori rientrino rispettivamente nei limiti di ascisse e ordinate. Se la posizione è fuori da uno dei limiti viene aggiunto un belief contenente la rispettiva codifica: T(top), R(right), B(bottom), L(left).

```
18  isInFieldX(X) :-
19    field(T,R,B,L),
20    (
21      not (X =< R) -> asserta(reachedLimit('R'));
22      not (X >= L) -> asserta(reachedLimit('L'));
23      true
24    ).
25
26  isInFieldY(Y) :-
27    field(T,R,B,L),
```

```
28    (
29      not (Y =< T) -> asserta(reachedLimit('T'));
30      not (Y >= B) -> asserta(reachedLimit('B'));
31      true
32    ).
33
34  checkPosition(X,Y) :-
35    isInFieldX(X),
36    isInFieldY(Y).
```

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

Se presenti, i belief aggiunti per il raggiungimento dei limiti del bounding box, vengono 'consumati' all'interno del ciclo di ragionamento utilizzando la libreria 'alice.tuProlog' e poi viene invocato il metodo *changeDirectionAngle* per modificare la direzione del nodo per riportarlo all'interno dei limiti.

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

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)