Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Dipartimento di Informatica - Scienze e Ingegneria

Corso di Laurea in Ingegneria e Scienze Informatiche

Modello ad agenti in Alchemist

Presentata da:
Filippo Nicolini

a.a. 2018-2019

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.

Gli agenti sono definiti come entità autonome distribuite nello spazio ed in grado di interagire tra pari e con l'ambiente in cui sono posti.

Per creare l'incarnazione sarà necessario definire gli agenti mappando il loro modello su quello fornito da Alchemist.

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 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***.

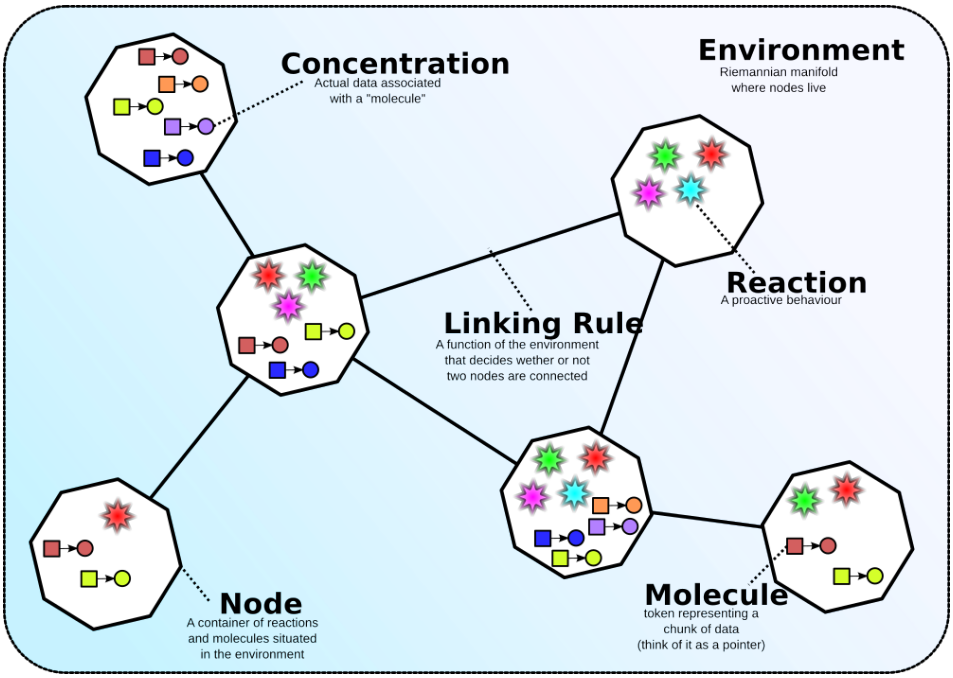

Figura 1.1: Illustrazione meta-modello di Alchemist

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.

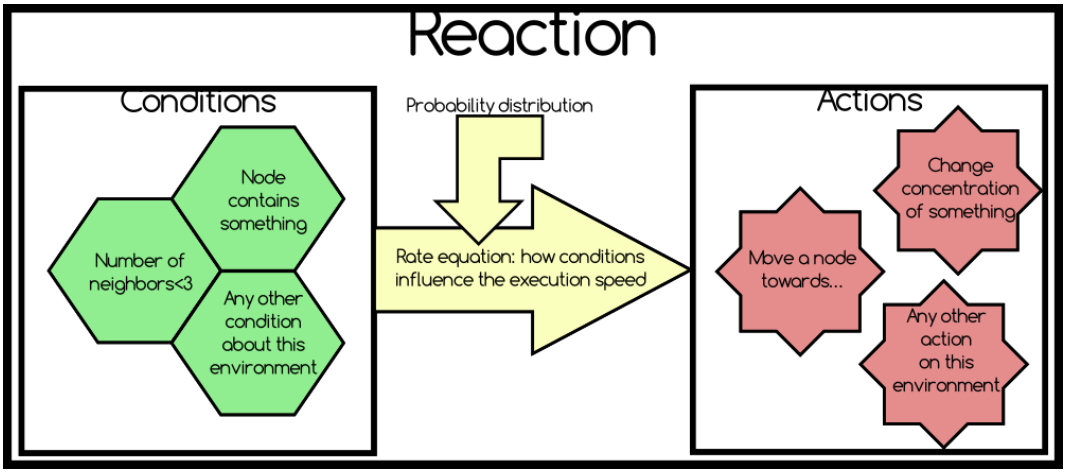

Figura 1.2: Illustrazione modello reazione di Alchemist

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, in un dato intervallo di tempo, generati da Alchemist e che innescano la verifica delle condizioni che possono portare alla potenziale esecuzione delle azioni.

Un'***Azione*** è la definizione di una serie di operazioni che modellano un cambiamento nel nodo o nell'environment.

In Alchemist un'incarnazione è un'istanza concreta del meta-modello appena descritta e che implementa una serie di componenti base come: la definizione di una molecola e del tipo di dati della concentrazione, un set di condizioni, le azioni e le reazioni. Incarnazioni diverse possono modellare universi completamente differenti.

Capitolo 2

Agenti

Un'agente è un entità che agisce in modo autonomo e continuo in uno spazio condiviso con altri agenti e le sue caratteristiche principali 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 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.

2.2 Agenti in tuProlog

Gli agenti descritti in tuProlog sono entità computazionali autonome che effettuano delle operazioni che impattano sul proprio stato e su quello dell'ambiente in cui sono immersi. La reattività è una caratteristica importante per quanto riguada gli agenti poichè fa si che siano pronti a reagire ad eventi o cambiamenti che avvengono nello spazio. Un'agente è quindi definito attraverso una teoria, la quale è composta da regole che corrispondono ai piani dell'agente. I piani sono formati da una serie di operazioni volte a svolgere un'azione.

2.3 Modello Jason

Il modello di agenti a cui si fa riferimento in questo testo è quello utilizzato in Jason. Jason è un interprete, per la versione estesa di AgentSpeak, che implementa la semantica operazionale di tale linguaggio e fornisce una piattaforma per lo sviluppo di sistemi multi-agente. AgentSpeak, come appena accennato, è un linguaggio di programmazione orientato agli agenti basato sulla programmazione logica e l'architettura BDI.

Gli agenti BDI (Beliefs-Desires-Intentions) 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.

Capitolo 3

Progetto

Il progetto, come descritto nell'introduzione, ha come obiettivo l'approccio all'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'agente 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 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.

3.2 Incarnazione

L'incarnazione in Alchemist corrisponde all'implementazione del modello che si vuole realizzare espresso nella forma del meta-modello che è stato descritto nella sezione 1.1. Dopo aver analizzato i due diversi mapping e le relative opportunità implementative è stato scelto per implementare il modello ad agenti quello realizzato nella seconda analisi che riferisce l'agente ad una reazione.

Lo sviluppo è quindi partito dalla definizione della classe **AgentIncarnation** che implementa l'interfaccia *Incarnation* nella quale sono definiti i metodi per la creazione delle entità del meta modello (Molecola, Concentrazione, Nodo, Distribuzione temporale, Reazione, Condizione, Azione). Per oguna di quest'ultime verrà presentata qui di seguito la logica implementativa.

Per la realizzazione del **nodo**, che è l'entità centrale del modello di Alchemist, è stata creata la classe **AgentsContainerNode** che estende la classe astratta *AbstractNode*. All'interno del nodo, sono definite le proprietà per la memorizzazione del riferimento all'ambiente a cui appartiene e una mappa che memorizza gli agenti presenti al suo interno in coppie (*nome_agente*, *oggetto_agente*). Per quanto riguarda i metodi sono stati imlementati quelli dell'interfaccia e aggiunti altri per la gestione della mappa (inserimento agente e recupero dello stesso dalla mappa).

Per la **distribuzione temporale** è stata utilizzata la classe *DiracComb*, già presente all'interno delle implementazioni di Alchemist, che permette la realizzazione di un pettine di Diract il cui spazio tra gli intervalli è definito dal parametro passato al costruttore. In questo caso il parametro è quello recuperato dalla configurazione della simulaizone poichè può variare per ogni specifica reazione.

Il metodo per la creazione delle **reazioni** viene invocato da Alchemist in base alla struttura del file di configurazione che specifica quali reazioni creare e in che nodo inserirle. Come visto precedentemente nella sezione 1.1 le reazioni sono formate da condizioni e azioni, quest'ultime scatenate se si verificano le condizioni associate. La classe che definisce questa entità è **AgentReaction** che estende la classe astratta *AbstractReaction*. All'interno della classe è definita la proprietà per memorizzare il nome dell'agente e i metodi definiti nell'interfaccia. Nel mapping realizzato la reazione si riferisce all'agente poichè ciò che la compone sono l'azione da scatenare e la clausola per innescarla ma l'agente vero e proprio, come vedremo successivamente, sarà implementato nelle azioni.

La **condizione** è creata utilizzando la classe *AbstractCondition*, già definita nelle implementazioni di Alchemist, e che permette, attraverso i metodi rimasti da implementare, di definire la clausola che si deve verificare per scatenare le azioni di quella reazione.

Per la realizzazione delle **azioni** si è deciso di definire una classe astratta che racchiuda al suo interno le funzionalità dell'agente così che le classi spe-

cifiche possano far eseguire all'agente solo le operazioni desiderate. Per fare ciò è stata creata la classe **AbstractAgent** che estende la classe astratta *AbstractAction* non implementando i metodi *cloneAction* e *execute* in modo tale da lasciare la loro definizione alle classi specifiche. Le proprietà della classe creata sono relative al nome dell'agente e al motore tuProlog, all'interno del quale è caricata la libreria che definisce le regole da poter utilizzare.

Le entità **molecola** e **concentrazione** sono state utilizzate solo per la tematizzazione delle simulazioni utilizzando come molecola il nome dell'agente e come valore 0. Associando, nel pannello della simulazione, colori diversi ad ogni nome la visione della simulaizone viene agevolata.

Questa implementazione delle classi utilizzate all'interno dell'incarnazione è quella base; altri metodi e altre proprietà verranno aggiunte con la realizzazione delle funzionalità specifiche, come vedremo successivamente.

Referenziazione oggetti Java-tuProlog

All'interno della classe **AbstractAgent**, come accennato precedentemente, sono implementate le funzionalità da utilizzare nella definizione del ciclo di ragionamento che verrà descritto nella sezione 3.3. Al suo interno sono stati utilizzati inoltre meccanismi per la referenziazione di oggetti Java all'interno di tuProlog, i quali consentono di effettuare invocazioni, dalla teoria dell'agente, di metodi definiti nelle classi Java. Per fare ciò sono stati registrati all'interno della teoria due variabili identificate con delle stringhe e riferite ad oggetti Java, una per il nodo (all'interno del quale è posizionato l'agente) e una per l'agente stesso.

```
1 private final Library lib =
    this.engine.getLibrary("alice.tuprolog.lib.OOLibrary");
2 ...
3 // Object reference for internal actions
4 ((OOLibrary) this.lib).register(new
    Struct("agent"), this);
```

```
5 // Object reference for external actions
6 ((OOLibrary) this.lib).register(new Struct("node"),
    this.getNode());
```

Codice sorgente 3.1: Associazione riferimento oggetti Java per tuProlog

3.3 Ciclo di ragionamento

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

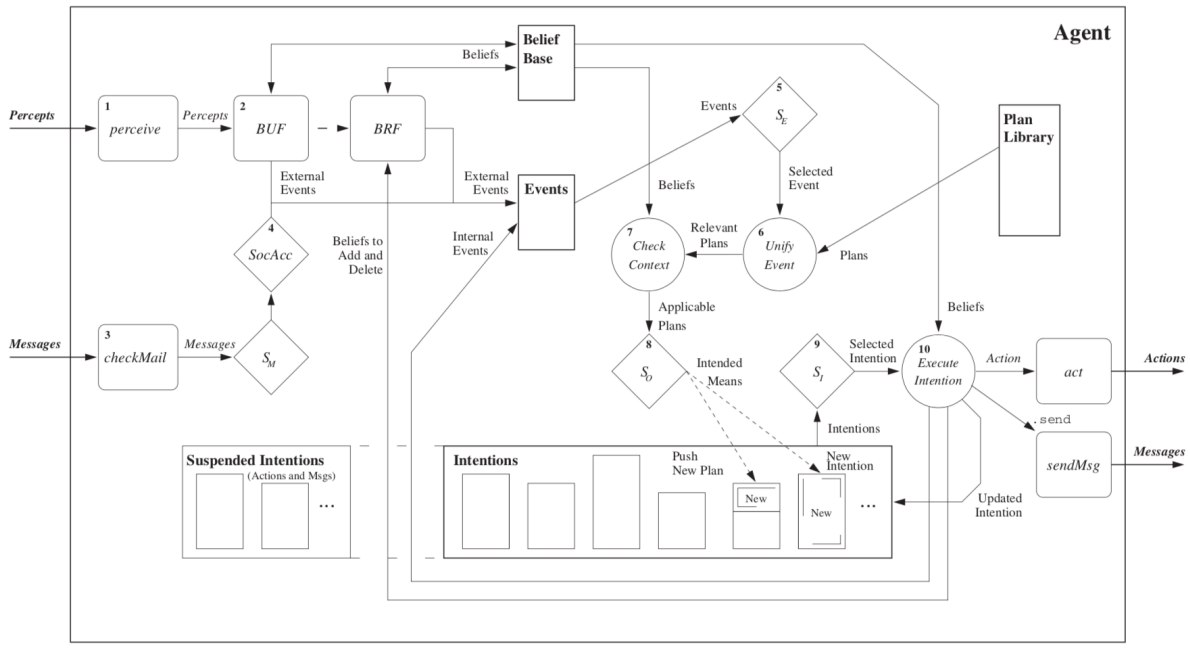

Figura 3.1: Ciclo di ragionamento di un agente Jason

Per costruire il ciclo di ragionamento per gli agenti, da implementare all'interno di Alchemist, ci si è ispirati a quello proposto da Jason che è rappresentato in figura 3.1.

3.3.1 Ciclo di ragionamento in Jason

Per comprendere meglio la rappresentazione del ciclo di ragionamento implementato in Jason (rappresentato in Figura 3.1) qui di seguito sono spiegati i 10 step da cui è composto. 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 3.1 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 3.1 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 3.1 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 3.1 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 3.1 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 3.1 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.

3.3.2 Ciclo di ragionamento in Alchemist

Il ciclo di ragionamento in Figura 3.1 è ciò che si è cercato di replicare per definire quello usato in questo progetto. Ad ogni iterazione del ciclo l'agente deve essere in grado di:

- percepire le modifiche dell'ambiente che lo circonda
- ricevere i messaggi e selezionarne uno da leggere (metodologia FIFO)
- aggiornare la 'belief base'
- selezionare un'intenzione da perseguire (metodologia Round-Robin)
- eseguire la prima operazione dello stack dell'intenzione selezionata e riaggiungere l'intenzione alla coda.

Per poter gestire le intenzioni sono stati previsti una serie di meccanismi sia lato Alchemist che lato tuProlog. All'interno della classe *AbstracAgent* un'array contente gli identificativi delle intenzioni e le funzioni per crearle, rimuoverle e selezionarle per l'esecuzione mentre in tuProlog, attraverso apposite regole, viene gestita l'esecuzione delle operazioni che la compongono.

Crezione intenzione

Nel momento in cui Alchemist riceve un'evento da notificare all'agente, viene costruito un apposito fatto ed attraverso l'utilizzo del predicato *clause(Head, Body)* che permette di recuperare il corpo di una regola conoscendo la sua testa. Se ci fossero più alternative disponibili, vengono tutte recuperate e da ognuna si ottiene la lista di operazioni che compongono il corpo da cui verrà generata l'intenzione.

La creazione di un'intenzione segue questo processo:

- recupero lista di operazioni per una certa intenzione
- generazione di un identificativo univoco
- creazione del fatto *intention(Id,ListaOperazioni)*
- inserimento dell'id nello stack lato Java e del fatto all'interno della teoria tuProlog

Per ogni evento si avrà quindi una corrispondente intenzione che, come spiegato precedentemente, verrà eseguita periodicamente fino alla sua conclusione.

Selezione e avvio esecuzione intenzione

Ad ogni ciclo di ragionamento, dopo aver creato le intenzioni per gli eventi occorsi, si passa alla selezione dell'intenzione che deve essere eseguita. Per garantire ad ognuna equa possibilità viene utilizzato il meotodo Round-Robin, che prevede l'utilizzo di un 'token' circolare che viene passato virtualmente tra gli elementi e chi ne è in possesso può compiere un'azione, in questo caso sarebbe l'esecuzione di un'operazione. L'implementazione è effettuata tramite l'utilizzo di un array al quale viene prelevato l'elemento di testa e che poi una volta eseguito viene riaggiunto in coda. Per avviare l'esecuzione e passare il controllo alla teoria viene risolto tramite il motore tuProlog *execute(IntentionID)*.

Gestione esecuzione intenzione

Per la gestione dell'esecuzione delle intenzioni lato tuProlog sono state definite una serie di regole all'interno della libreria base dell'agente caricata nella classe *AbstracAgent*. Nella sezione precedente è stato selezionato l'identificativo dell'intenzione da eseguire, la quale poi viene scatenata. Nella libreria dell'agente sono presenti due regole che hanno come testa *execute(IntentionID)* ma corpi differenti per due diverse risoluzioni: la definizione è mostrata nel Codice sorgente 3.2.

```
1  execute(I) :-
2      intention(I, []),
3      !,
4      agent <- removeCompletedIntention(I).
5
6  execute(I) :-
7      retract(intention(I, [ACTION | STACK])),
8      execute(I, ACTION, TOP),
9      !,
10     append(TOP, STACK, NEWSTACK),
11     assertz(intention(I, NEWSTACK)).
```

Codice sorgente 3.2: Regole per l'esecuzione dell'intenzione

Le due regole mostrate operano, come appena detto, due azioni diverse. La prima recupera il fatto dell'intenzione utilizzando l'identificativo e una lista vuota: se va a successo vuol dire che l'intenzione selezionata non ha più operazioni nel suo stack e quindi può essere eliminata. Ed infatti, attraverso l'oggetto Java dell'agente, referenziato nella variabile *agent* in tuProlog, viene invocato il metodo per cancellarla. Questo metodo elimina l'identificativo dallo stack lato Alchemist e poi rimuove il fatto dalla teoria. Nella seconda regola, invece, viene rimosso il fatto relativo all'identificativo dell'intenzione e recuperato lo stack delle operazioni scomponendolo in testa e coda, rispettivamente 'ACTION' e 'STACK'. Successivamente viene richiamata l'esecuzione dell'azione in testa, la cui eventuale sequenza di operazioni innescata dalla sua esecuzione viene restituita nella variabile 'TOP'. Quest'ultima viene quindi concatenata alla coda ('STACK') in modo tale che le prossime volte che l'intenzione viene selezionata le operazioni vengano eseguite.

Essendo l'ordine importante, come si può vedere dal codice mostrato, viene prima inserita la regola per la rimozione e successivamente quella per effettuare l'esecuzione delle azioni.

Gestione esecuzione azioni

Fino ad ora è stato mostrato come è stata implementata la gestione delle intenzioni, sia lato Alchemist (Java) che lato tuProlog, e la loro esecuzione ma non sono state descritte le azioni che le compongono. Le tipologie di azioni che sono state implementate sono interne ed esterne. Quelle interne sono azioni che hanno impatto internamente all'agente e sono definite nell'implementazione della sua classe. Invece, quelle esterne, hanno impatto sull'ambiente e vengono gestite dal nodo che, per l'implementazione creata, è sia l'elemento inserito nell'ambiente sia esso stesso uno spazio per gli agenti posizionati al suo interno.

Un'altra differenziazione che si può fare è quella tra le varie tipologie di goal definite: achievement, test, concurrent. Relativamente al mondo Jason sono riferiti rispettivamente ai simboli '!', '?' e '!!' o '!?' e stanno ad indicare i goal, ovvero operazioni 'prioritarie' rispetto ad altre.

3.4 Funzionalità agente

All'interno del ciclo di ragionamento implementato nella classe specifica dell'agente possono essere richiamate le funzionalità definite nella classe astratta oppure altre definite localmente. Nell'attuale implementazione dell'incarnazione, nella classe *AbstracAgent* sono state definite funzioni per lo scambio di messaggi tra agenti, lo spostamento del nodo in cui sono posizionati gli agenti, la notifica dell'aggiornamento (inserimento o rimozione) della 'belief base'.

Nelle prossime sezioni sono mostrate le implementazioni delle funzionalità appena descritte.

3.4.1 Scambio di messaggi

Per la realizzazione dello scambio di messaggi tra agenti si è modificata la classe **AbstractAgent** inserendo due code per i messaggi, una per quelli

in entrata e una per quelli in uscita, e i relativi metodi per la loro gestione (lettura e inserimento in entrambe le code).

Le due code sono state realizzate utilizzando oggetti di tipo *Queue* e istanziati come *LinkedList*: una lista conterrà i messaggi in entrata mentre l'altra quelli in uscita. Per gestire meglio i messaggi sono state create due classi InMessage e OutMessage, dove la prima si riferisce a quelli ricevuti mentre la seconda a quelli inviati. Entrambe sono state definite all'interno della classe **AbstractAgent**. La classe InMessage ha al suo interno le proprietà sender e payload usate per descrivere rispettivamente il mittente e il contenuto del messaggio. La classe OutMessage contiene le proprietà sender, receiver e payload per descrivere il mittente, il destinatario e il contenuto del messaggio. In entrambe le classi sono definiti solo i metodi getter poichè le proprietà sono assegnate solo nel costruttore della classe.

La gestione delle code tramite i metodi avviene secondo la politica FIFO, quindi l'inserimento è fatto in coda e la rimozione dalla testa. Come avviene la consegna dei messaggi vera e propria verrà spiegato successivamente nella sezione 3.4.1.

Inizialmente sia la lettura che la scrittura erano gestite effettuando chiamate tramite il motore tuProlog per inserire o recuperare i messaggi nelle code ed anche per la loro lettura. Dopo la rifattorizzazione e l'inserimento degli oggetti Java all'interno della teoria l'implementazione è stata semplificata permettendo una gestione più snella.

Per la **lettura dei messaggi** è stato definito il metodo *readMessage*, il quale può essere inserito nel ciclo di ragionamento per permetter all'agente di gestire i messaggi ricevuti. All'interno del metodo viene recuperato, se presente, il primo messaggio della coda e successivamente viene composto il predicato 'onReceiveMessage(S,M)' dove S è il mittente e M è il contenuto del messaggio. Attraverso l'utilizzo del motore tuProlog, il predicato appena descritto viene eseguito nella teoria dell'agente che se opportunamente programmata può essere in grado di ricevere il messaggio ed operare di conseguenza.

L'**invio di messaggi** viene scatenato dalla teoria tramite cosiddetta un'azione interna. Le azioni interne sono azioni che hanno luogo internamente all'agente e, in questo caso, prevede l'inserimento di un messaggio nella coda di quelli in uscita. L'azione interna prima di essere invocata passa per due controlli: il primo avviene nella teoria dell'agente e il secondo nel metodo di disambiguazione definito in Alchemist. Nella teoria dell'agente il controllo avviene verificando la presenda di predicati del tipo 'is_internal(ACTION)' dove 'ACTION' è l'azione da eseguire. Nel caso specifico controllerà dell'esistenza del predicato 'is_internal(iSend(S,M))': in caso positivo verrà eseguita l'operazione definita nel Codice sorgente 3.3 per innescare l'azione lato Java.

```
1 agent <- executeInternalAction(ACTION)
```

Codice sorgente 3.3: Invocazione esecuzione azione interna

Consegna dei messaggi

Fino a questo momento gli agenti sono in grado di leggere messaggi presenti nella coda in entrata oppure inserirne di nuovi nella coda di quelli in uscita. La parte mancante è quella della consegna dei messaggi in uscita ai rispettivi destinatari. Per fare ciò si è pensato di creare un'apposito agente e definire una classe specifica, la classe **PostmanAgent**. L'agente Postman, che deve essere instanziato in configurazione, implementa la classe AbstracAgent come descritto nel Codice sorgente 3.4. I metodi definiti al suo interno sono:

- il costruttore, diversamente dagli altri agenti non viene passata la reazione
- cloneAction, definito nell'interfaccia Action, per clonare l'azione
- execute, definito nell'interfaccia Action, si riferisce al ciclo di ragionamento

```
1 public PostmanAgent(String name, Node node,
    RandomGenerator rand) {
2    super(name, node, rand);
3 }
4
5 @Override
6 public Action cloneAction(Node node, Reaction
    reaction) {
7    return new PostmanAgent("cloned_" +
    this.getAgentName(), node,
    this.getAgentRandomGenerator());
8 }
9
10 @Override
11 public void execute() {
12    getNode().postman();
13 }
```

Codice sorgente 3.4: Agente Postman

Come è possibile osservare, nel suo ciclo di ragionamento l'agente Postman non fa altro che eseguire l'azione postman definita nel nodo ed implementata come descritto nel Codice sorgente 3.5. Inizialmente vengono create una mappa di appoggio per gli agenti e una lista dei messaggi in uscita. Per ogni nodo viene recuperata la sua mappa di agenti, quelli contenuti al suo interno, e viene aggiunta a quella temporanea, la quale viene poi iterata per recuperare tutti i messaggi in uscita da ogni agente aggiunti alla lista creata in precedenza. Una volta ottenuti i messaggi viene iterata la lista e, utilizzando la mappa degli agenti creata in precedenza per recuperare l'istanza del destinatario di ogni messaggio, viene inserito invocato il metodo *addIncomingMessage* che aggiunge un messaggio alla coda di quelli in entrata per quell'agente.

```
1 Map<String, AbstractAgent> tmpAgentMap = new
    LinkedHashMap<>();
2 List<AbstractAgent.OutMessage> outMessages = new
    ArrayList<>();
3
4 // For each node in the environment get all its
    agents
5 this.environment.getNodes().forEach(node -> {
6     tmpAgentMap.putAll(((AgentsContainerNode)
    node).getAgentsMap());
7 });
8
9 // For each agent takes outgoing messages
10 tmpAgentMap.forEach((agentName, agent) -> {
11
    outMessages.addAll(agent.consumeOutgoingMessages());
12 });
13
14 // Send each message to the receiver
15 outMessages.forEach(message -> {
16     tmpAgentMap.get(message.getReceiver().toString())
17     .addIncomingMessage(message);
18 });
```

Codice sorgente 3.5: Metodo dell'agente Postman

3.4.2 Spostamento del nodo

Per lo spostamento del nodo all'interno dello spazio si è deciso di modificare la classe **AgentsContainerNode** aggiungendo tre proprietà per memorizzare direzione, velocità e il tau (tempo della simulazione) dell'ultimo

aggiornamento. Inoltre sono stati definiti i metodi per recuperare e impostare i valori della velocità e della direzione. Per quanto riguarda l'aggiornamento della posizione è stato definito il metodo *changeNodePosition(Time updateTau)* che prende in input il tau dell'aggiornamento ed è implementato come mostrato dal Codice sorgente 3.6. Questo metodo è quindi responsabile di calcolare la nuova posizone del nodo utilizzando le proprietà descritte in precedenza. Per prima cosa viene recuperata la posizione corrente del nodo che viene utilizzata come centro per la costruzione di una circonferenza, la quale avrà come raggio la distanza calcolata moltiplicando la velocità per il tempo trascorso dall'ultimo aggiornamento. La nuova posizione del nodo si troverà dunque sulla circonferenza e per selezionare il punto viene utilizzata la direzione, espressa in radianti.

```
1 Position currentPosition = this.getNodePosition();
2 double radAngle = this.nodeDirectionAngle;
3 // radius = space covered = time spent * speed
4 double radius = (updateTau.toDouble() -
    this.lastUpdateTau.toDouble()) * this.nodeSpeed;
5 double x = currentPosition.getCoordinate(0) +
    radius * Math.cos(radAngle);
6 double y = currentPosition.getCoordinate(1) +
    radius * Math.sin(radAngle);
7
8 this.environment.moveNodeToPosition(this,
    this.environment.makePosition(x, y));
9 this.lastUpdateTau = updateTau;
```

Codice sorgente 3.6: Metodo per l'aggiornamento della posizione del nodo

Inizialmente l'aggiornamento era stato inserito erroneamente nel ciclo di ragionamento dell'agente. È stato rimosso quando ci si è resi conto che così facendo, in caso di più agenti all'interno dello stesso nodo, la posizione veniva aggiornata molteplici volte ed ognuna con parametri diversi poichè direzione

e velocità possono essere modificate costantemente. A seguito di un'analisi, è stato deciso di creare un agente specifico per l'aggiornamento della posizione inserito direttamente nel codice, diversamente dall'agente Postman che deve essere aggiunto nella configurazione della simulazione. L'agente appena descritto è definito dalla classe **MovementAgent**, la cui implementazione è praticamente identica a quella dell'agente Postman se non per il comportamento nel ciclo di ragionamento. Nel metodo *execute*, come si può osservare dal Codice sorgente 3.7, effettua due invocazioni di metodi definiti nella classe del nodo: la prima per l'aggiornamento della posizione al metodo descritto nel Codice sorgente 3.6 e la seconda per aggiornare i belief della distanza dagli altri agenti presenti nelle teorie. L'aggiornamento della posizione avviene utilizzando i valori di velocità e direzione presenti in quel momento. È quindi chiaro che se vi fossero più agenti che modifichino quelle proprietà solo le ultime impostate saranno quelle effettivamente utilizzate.

```
1 public MovementAgent(String name, Node node,
    RandomGenerator rand, Reaction reaction) {
2    super(name, node, rand, reaction);
3 }
4
5 @Override
6 public Action cloneAction(Node node, Reaction
    reaction) {
7    return new MovementAgent("cloned_" +
    this.getAgentName(), node,
    this.getAgentRandomGenerator(), reaction);
8 }
9
10 @Override
11 public void execute() {
12    this.getNode().changeNodePosition(
13       this.getAgentReaction().getTau());
```

```
14
15     this.getNode().updateAgentsPosition();
16 }
```

Codice sorgente 3.7: Agente per lo spostamento del nodo

Questo agente viene aggiunto alla creazione di ogni nodo: è quindi creata una reazione, chiamata 'movementReact', alla quale viene aggiunta la stessa condizione utilizzata per gli altri agenti e come azione l'agente appena definito. La reazione è definita con una distribuzione temporale di 1 tau, in modo tale che se si vuole simulare l'aggiornamento della posizione, come se arrivasse da un apparato GPS, sarà sufficiente impostare valori più alti per le distribuzioni temporali di altri agenti. Terminata la creazione, la reazione viene aggiunta al nodo che verrà poi popolato con altri agenti. In questo modo la funzionalità dello spostamento del nodo è implicita nell'incarnazione ed è il programmatore dell'agente che la può sfruttare implementando opportunamente gli agenti.

3.4.3 Aggiornamento della '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 tuProlog 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 ad Alchemist.

```
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 3.8: Piani per l'aggiornamenot della 'belief base'

I piani, definiti nella teoria, che permettono questo comportamento, sono mostrati nel Codice sorgente 3.8. L'aggiornamento della 'belief base' inizia con un'invocazione, da parte dell'agente, di uno di questi due piani che porta all'aggiunta del belief e di un secondo belief fittizio. Lato Alchemist, durante il suo ciclo di ragionamento, l'agente invoca il metodo *beliefBaseChanges()* implementato nella classe **AbstractAgent**. All'interno del metodo vengono ritirati uno ad uno i belief fittizi inseriti in precedenza e poi per ognuno viene eseguito il piano 'onAddBelief(B)' o 'onRemoveBelief(B)', dove B è il belief modificato, a seconda che il belief recuperato sia rispettivamente di aggiunta o di rimozione.

3.5 Simulazione

Attraverso la configurazione e l'esecuzione delle simulazioni è possibile vedere il comportamento delle incarnazioni definite all'interno di Alchemist. Nelle prossime sezioni verrà mostrato quali sono le parti che compongono la configurazione di una simulazione e qual è la configurazione scelta.

3.5.1 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.

```
8  incarnation: agent
```

Codice sorgente 3.9: 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.

```
9  variables:
10   myVar: &myVar
11     par1: 0
12     par2: "string"
13   mySecondVar: &myVar2
14     par: "value"
```

Codice sorgente 3.10: Variabili simulazione

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

```
15 environment:
16   type: OSMEnvironment
17   parameters: [/maps/foo.pbf]
```

Codice sorgente 3.11: Environment

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

```
18 environment:
19   type: Continuous2DEnvironment
```

Codice sorgente 3.12: 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.

```
20 positions:
21   type: LatLongPosition
```

Codice sorgente 3.13: 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.

```
22 network-model:
23   type: EuclideanDistance
24   parameters: [10]
```

Codice sorgente 3.14: 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.

```
25 network-model:
26   type: NoLinks
```

Codice sorgente 3.15: 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 3.16.

```
27  displacements:
28    - in: {type: Circle, parameters: [5,0,0,2]}
29      programs:
30      -
31        - time-distribution: 1
32          program: "reactionParam"
33        - time-distribution: 2
34          program: "doSomethingParam"
35    - in: {type: Point, parameters: [1,1]}
36      programs:
37      -
38        - time-distribution: 1
39          program: "pointReactionParam"
```

Codice sorgente 3.16: Disposizione nodi e reazioni associate

3.5.2 Preparazione alla simulazione

Per mostrare quanto scritto fino a questo punto si è deciso di realizzare una simulazione utilizzando un esempio fra quelli descritti nella pagina di

Jason. Il problema in questione è quello dei 'Goldminers' che prevede l'utilizzo di tre entità (minatore, miniera e deposito) e il cui svolgimento è il seguente. I minatori e il deposito sono posizionati casualmente e i minatori conoscono la posizione del deposito, attraverso un'opportuna configurazione iniziale. Le miniere, che potrebbero essere posizionate con Spatial Tuples, contengono una certa quantita di minerale. I minatori muovendosi casualmente, ad esempio utilizzando la distribuzione di levy, cercano una miniera che contenga del minerale. Quando trova una miniera recupera una risorsa, la porta al deposito e poi torna alla miniera per continuare ad estrarre risorse finchè non sono esaurite. Se una miniera non ha più risorse il minatore si muove in modo casuale per cercarne una nuova.

Per realizzare la simulazione è stato scritto il file di configurazione indicato nel Codice sorgente 3.17

```
1  incarnation: agent
2
3  network-model:
4    type: ConnectWithinDistance
5    parameters: [2]
6
7  displacements:
8    - in: {type: Circle, parameters: [2,2,2,0.2]}
9      programs:
10       -
11         - time-distribution: 10
12           program: "miner"
13
14   - in: {type: Circle, parameters: [1,2,2,0.2]}
15     programs:
16       -
17         - time-distribution: 10
18           program: "deposit"
```

```
19
20      - in: {type: Circle, parameters: [1,2,2,0.2]}
21        programs:
22          -
23            - time-distribution: 10
24              program: "postman"
25
26      - in: {type: Circle, parameters: [10,2,2,5]}
27        programs:
28          -
29            - time-distribution: 10
30              program: "goldmine"
```

Codice sorgente 3.17: Simulazione modello agenti su problema Goldminers

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

Il minatore (miner) esegue nella parte di inizializzazione l'impostazione della configurazione iniziale dello stato, della posizione del deposito e aggiorna velocità e direzione del nodo. Successivamente nel ciclo di ragionamento alterna 4 stati:

1. (stato ricerca): l'agente è alla ricerca di una miniera con una pepita e si sposta in modo casuale (seguendo la distribuzione di Levy). Ad ogni spostamento invia una richiesta nel vicinato per prelevare la pepita.

2. (ricezione pepita): la pepita è stata prelevata. L'agente salva la posizione della miniera, recupera le coordinate del deposito e si muove in quella direzione.

3. (arrivo deposito): nelle vicinanze del deposito l'agente manda un messaggio al deposito per consegnare la pepita. Inviato il messaggio vengono recuperate le coordinate della miniera salvate in precedenza e si dirige verso quel punto.

4. (arrivo miniera): tornato alla miniera viene impostato nuovamente lo stato 1.

La classe Java che carica questa teoria è **SimpleAgent** che definisce un agente che, nel suo ciclo di ragionamento, invoca tutte le funzionalità descritte in precedenza.

Per quanto riguarda la miniera (goldmine) la sua teoria prevede solamente che, alla sua inizializzazione, venga generato un numero casuale di pepite contenute al suo interno. Questa teoria viene caricata nella classe **Goldmine** che è l'implementazione della classe astratta **AbstractSpatialTuple** che implementa il modello Spatial Tuples. Al suo interno sono definiti i metodi per la gestione delle tuple definiti dal modello, ovvero riferiti a 'in', 'out', 'rd'.

Per definire il deposito viene utilizzata anche qui la classe **SimpleAgent** ma nella sua teoria non sono definiti comportamenti.

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)