Alma Mater Studiorum
Università degli Studi di Bologna

Scuola di Ingegneria e Architettura

Campus di Cesena

Corso di laurea magistrale in Ingegneria Informatica

Titolo dell'elaborato

Esperimenti di Stigmergia Cognitiva in TuCSoN e Jason

Elaborato in

Sistemi Multi-Agente

Presentato da
Chiara Casalboni

Relatore
Prof. Andrea Omicini

Correlatore
Dott. Stefano Mariani

Sessione II - Anno Accademico 2012-2013

Indice

Ringraziamenti

Al termine di questo percorso desidero, innanzitutto, ringraziare il mio relatore, il professor Omicini, e il correlatore, il dottor Mariani, per il supporto, la professionalità e la simpatia.

Vorrei, inoltre, ringraziare mia madre, i miei fratelli e la mia splendida nipotina per l'amore e la sopportazione che hanno dimostrato in tutti questi anni. Grazie a nonna Marina perché è anche grazie a lei se ho potuto concludere il mio percorso di studi.

Grazie a Michele che mi è sempre stato vicino e mi ha accompagnato in questi anni sostenendomi anche nei momenti in cui non credevo nelle mie capacità.

Vorrei ringraziare anche i compagni/amici che hanno fatto parte dei primi tre anni di Ingegneria: Gardo, Davide e Manuele, non dimenticherò mai le risate fatte con voi. Ma un grazie va anche ai compagni che hanno fatto parte del percorso Magistrale.

Grazie alle vecchie e alle nuove amicizie: uscire con voi mi ha spesso fatto evadere e mi ha aiutato a superare momenti difficili e tristi.

Infine vorrei ringraziare Sara che ha sempre saputo come sostenermi e motivarmi e che mi dimostra ogni giorno quanto sia fiera di me e quanto mi voglia bene.

A tutti voi è dedicata questa tesi e a tutti voi un grazie sentito e sincero!

Introduzione

Fin dagli inizi del Diciannovesimo secolo scienziati e filosofi sono rimasti affascinati dal comportamento che certi insetti hanno in alcune situazioni. Per esempio vi sono le formiche rosse che vivono nelle foreste tropicali che sono in grado di sopravvivere alle alluvioni aggrappandosi le une alle altre quasi a formare una zattera; vi sono poi le vespe che sono in grado di costruire degli alveari perfettamente simmetrici; o ancora le formiche che sono in grado di formare vere e proprie processioni verso una fonte di cibo.

Gli studiosi per anni, osservando queste creature, non riuscivano a spiegare il loro comportamento se non grazie all'intervento di un super organismo in grado di organizzare il lavoro degli altri insetti.

All'inizio del Secolo scorso, poi, si passò all'idea che gli insetti erano in grado di auto-organizzarsi tramite dei meccanismi adottati da tutta la colonia. Si iniziò a parlare del concetto di *stigmergia* che può essere definito come un meccanismo di comunicazione indiretta tramite modificazioni apportate all'ambiente in cui gli agenti vivono, chiamate *tracce*.

In tempi relativamente recenti, si pensò che se la stigmergia riusciva a portare a risultati così rilevanti pur essendo messa in atto da agenti semplici come gli insetti, essa poteva portare a risultati ancora più importanti se messa in atto da agenti con capacità cognitive, agenti che fossero in grado di interpretare i segni che gli altri agenti lasciano nell'ambiente e ad agire di conseguenza. Questo tipo di stigmergia viene detta cognitiva proprio a voler sottolineare le capacità possedute dagli agenti.

Questo elaborato si pone come obiettivo quello di mostrare con dei semplici esperimenti, realizzati integrando i linguaggi `TuCSoN` e Jason, come si comportano gli agenti in un ambito di stigmergia cognitiva. L'attenzione deve essere posta a come gli agenti siano in grado di interpretare e reagire ai segni che altri agenti hanno predisposto all'interno dell'ambiente in cui vivono.

In particolare, nel Capitolo 1 si affronterà il percorso intrapreso dagli studiosi per giungere alla definizione di stigmergia cognitiva. Si partirà dalle idee dei ricercatori del Secolo XIX, i quali erano convinti che un comportamento così organizzato potesse dipendere solo da un essere esterno al sistema che, come un burattinaio coi suoi burattini, guidasse i comportamenti degli agenti, fino a giungere al concetto di stigmergia cognitiva.

Nel Capitolo 2, invece, si analizzeranno i due linguaggi di programmazione utilizzati per realizzare gli esperimenti. In particolare, si parlerà di `TuCSoN` che è il linguaggio utilizzato per realizzare la comunicazione indiretta tra gli agenti; `TuCSoN` infatti rende disponibile il concetto di centro di tuple nel quale gli agenti

possono inserire informazioni sotto forma di tuple. Si parlerà anche di Jason che è il linguaggio utilizzato per definire il comportamento degli agenti e quindi per renderli degli esseri cognitivi.

Nel Capitolo 3 si affronterà l'aspetto di integrazione dei due linguaggi utilizzati; si vedrà in che modo questi due linguaggi differenti, ma entrambi basati su Java, sono stati messi nelle condizioni di poter interagire.

Infine, nel Capitolo 4, verranno presentati due diversi esperimenti utili per evidenziare quanto detto nel Capitolo 1 a riguardo della stigmergia cognitiva e realizzati grazie all'integrazione tra i linguaggi `TuCSoN` e Jason come spiegato nei Capitoli 2 e 3.

Capitolo 1

Verso la Stigmergia Cognitiva

1.1 Cenni storici: dal superorganismo alla stigmergia

1.1.1 Società di insetti come superorganismi

Il comportamento collettivo degli insetti sociali ha sempre rappresentato un aspetto interessante per filosofi e scienziati che hanno cercato di spiegare come fosse possibile che creature così semplici riuscissero a coordinare le loro azioni. Per un osservatore esterno alla colonia, il comportamento degli insetti potrebbe sembrare essere guidato da un agente di coordinazione virtuale presente al centro della colonia [14].

Fin dal XIX secolo autori e studiosi hanno sempre cercato di spiegare questo fenomeno. L'autore Herbert Spencer, per esempio, nella seconda parte del XIX secolo affermò che *ogni società è un organismo*, chiamato *organismo sociale*, del tutto simile agli altri individui ma con una vita più lunga.

Nel 1877 la metafora dell'*organismo sociale* venne ripresa dal francese Alfred Espinas per descrivere il comportamento osservabile nelle società formate dagli animali: egli affermò che la loro tendenza naturale è quella di aggregarsi in gruppi più numerosi.

Nel 1911 l'entomologo Wheeler suggerì l'esistenza, all'interno delle colonie di insetti, di alcuni "*vincoli di funzionamento*" che permettono all'intera società di comportarsi come se fosse un unico organismo, chiamato successivamente *superorganismo*. Sfortunatamente a quei tempi non vi erano i mezzi sufficienti per determinare quali fossero tali vincoli.

1.1.2 Società di insetti come insieme di individui indipendenti

Nel 1930 il biologo francese Etienne Rabaud introdusse la propria idea secondo la quale l'unica causa del comportamento degli individui è da ricercare negli individui stessi. Rabaud cercò di dimostrare che ogni insetto all'interno della società si comporta come se fosse solo pur lavorando nello stesso modo degli altri,

ne deriva che il lavoro collettivo è solo la giustapposizione dei lavori compiuti individualmente.

Rabaud affermò che il comportamento individuale è il motore essenziale per qualsiasi azione o processo collettivi ed introduce due concetti chiave:

Interazione: gli animali sociali, vivendo costantemente a contatto l'uno con l'altro, potrebbero, con le loro azioni, influenzare gli altri individui modificando anche il loro comportamento.

Interattrazione: definito come un fenomeno sociale di base, si riferisce al fatto che gli animali appartenenti ad una specie sono attratti in modo specifico dagli animali della loro stessa specie.

Rabaud sostenne che l'*emergenza* di un comportamento collettivo e coordinato all'interno della società può essere spiegato da un insieme di interazioni specifiche tra gli individui.

Il concetto di *comportamento emergente* racchiude alcuni aspetti:

- Non c'è dietro un progetto razionale che abbia prodotto questo comportamento.
- Non deriva da un'analisi ingegneristica.
- Nessun individuo all'interno del sistema agisce consapevole del fatto che il proprio comportamento porterà ad uno comportamento specifico del sistema stesso.

Se ne deduce che l'*emergenza* è negli occhi dell'osservatore del sistema e non degli individui che ne fanno parte, in quanto essi non hanno una prospettiva globale: non sono consapevoli ne di ciò che globalmente accade ne di eseguire azioni intelligenti.

Queste idee sono state successivamente sviluppate da Grassé [4], il quale affermò che "*i gruppi sociali sono soprattutto caratterizzati dal fatto che ogni individuo, preso separatamente, produce uno stimolo preciso sui suoi compagni, mentre il gruppo (che può essere considerato come un singolo individuo) produce uno specifico stimolo che influenzerà il comportamento dei singoli individui*".

1.1.3 Il ruolo organizzativo delle interazioni e la stigmergia

Mentre Rabaud portava avanti le proprie ricerche, molti altri studiosi scoprirono che le interazioni specifiche tra gli individui sono coinvolte nell'origine di certi fenomeni collettivi.

Pierre-Paul Grassé fu il primo a fornire una visione sintetica di tale fenomeno collettivo combinando sia le interazioni tra gli individui sociali che il comportamento collettivo organizzato a livello della colonia.

L'idea di base portata avanti da Grassé è che la socialità non è solo una conseguenza che deriva dall'interazione, ma è una caratteristica biologica profondamente radicata negli individui di qualsiasi specie; Grassé chiamò questo comportamento *fame sociale* che guida gli individui alla ricerca dei propri compagni.

Inoltre, gli stimoli interindividuali fanno emergere un *effetto di gruppo* che descrive il fatto che quando un animale è sottoposto, in certe condizioni, ad un numero critico di stimoli specifici dai suoi compagni, il suo stato comportamentale viene alterato.

Inoltre, quando nelle società di animali si raggiunge un elevato numero di interazioni, si verifica l'emergenza di processi di integrazione e regolazione. Questa idea fu chiamata da Grassé *regolazione sociale* e indica che la società è in grado, attraverso determinati meccanismi, di ristabilire un equilibrio che era venuto a mancare oppure di coordinare le azioni per realizzare un compito. Questi meccanismi fanno affidamento sul fatto che ad uno stimolo segue una risposta: uno stimolo è un'azione compiuta da un individuo e la risposta è un'altra azione che è stata attivata dall'azione/stimolo. Ogni individuo, quindi, può essere considerato come una sorgente diretta di stimoli per gli altri individui.

Questo meccanismo apre la strada per una coordinazione indiretta delle attività degli individui, infatti i processi che regolano le interazioni non sono limitati all'influenza diretta degli stimoli prodotti dagli individui.

Il compito di ogni individuo è di organizzare l'ambiente in modo da creare strutture stimolanti che possano far scattare una specifica azione negli individui che ne vengono a contatto.

Il concetto di *stigmergia* (dal greco *stigma*: pungiglione e *ergon*: lavoro) fu inizialmente introdotto da Grassé per spiegare la coordinazione indiretta e la regolazione dei compiti nel contesto della costruzione del proprio nido da parte delle termiti.

Grassé mostrò che la coordinazione e regolazione delle attività di costruzione non dipendono dai lavoratori ma dalla struttura stessa: una determinata configurazione del nido attiva un'azione di una termite che procede a trasformare l'attuale configurazione in un'altra, la quale, a sua volta, può attivare il compimento di un'altra azione da parte dello stesso individuo oppure da un altro lavoratore della stessa colonia. La stigmergia, quindi, si presenta come un modo per comprendere la coordinazione e la regolazione delle attività collettive.

1.2 Stigmergia

Quando, in natura, si osservano dei sistemi che presentano un comportamento coordinato emerge l'idea di *intelligenza collettiva*, in quanto agenti semplici riescono, con il loro comportamento collettivo, a risolvere problemi complessi.

La capacità di risolvere un problema deriva dall'applicazione di un processo risolutivo che dipende dalla società degli agenti e non direttamente dai singoli agenti. Tali agenti non sono coordinati da un coordinatore generale ma sono in grado di auto-organizzarsi. La risoluzione del problema avviene poiché, grazie alle interazioni dei singoli, emerge un comportamento coordinato e finalizzato al raggiungimento di un obiettivo; quindi non c'è un coordinatore ma nemmeno un fine da perseguire, vi è solo un'organizzazione spontanea.

Non solo non vi è un coordinatore centralizzato, ma tutte le azioni sono eseguite in funzione di stimoli locali che gli agenti percepiscono dall'ambiente circostante; benché il comportamento sia locale, quello che si vede emergere è un comportamento su larga scala.

1.2.1 Proprietà dei sistemi computazionali nature-inspired

I sistemi naturali possono essere considerati dei veri e propri sistemi computazionali in quanto presentano aspetti propri di tali sistemi come [7]:

Robustezza: il sistema mantiene le sue funzionalità anche a fronte di moderati malfunzionamenti o mutamenti nell'ambiente.

Adattatività: capacità del sistema di modificare le funzioni svolte in modo da massimizzare l'efficienza nei confronti dei cambiamenti dell'ambiente. Questo aspetto si discosta dalla robustezza in quanto vi sono delle modifiche all'interno il sistema.

Auto-organizzazione: è il processo che porta un sistema dal completo disordine al completo ordine per mezzo di individui inconsapevoli di attuare questo cambiamento.

Considerato ciò, molte sono le attività di ricerca che si sono ispirate alla natura per definire dei modelli computazionali nature-inspired definendo due proprietà che un sistema di questo tipo deve avere [8]:

Autonomia a livello dei componenti: i singoli elementi che compongono il sistema devono agire autonomamente.

Auto-organizzazione a livello di sistema: gli elementi che compongono il sistema devono avere una sorta di auto-organizzazione per raggiungere risultati apprezzabili e visibili a livello di sistema.

Perciò per realizzare un sistema computazionale nature-inspired è necessario:

- Individuare le sorgenti di complessità nei sistemi naturali.
- Capire i meccanismi e i pattern che i sistemi naturali sfruttano per trattare con successo aspetti come apertura, distribuzione, larga scala e impredicibilità dell'ambiente.
- Mappare tali meccanismi e pattern con altri computazionalmente adatti ed equivalenti.
- Integrare i meccanismi definiti al punto precedente con i sistemi computazionali.

Osservando i sistemi naturali, risulta chiaro come essi siano l'unione di un elevato numero di componenti distribuiti ed eterogenei in natura, struttura e comportamento che devono essere in qualche modo coordinati. É quindi evidente che un modello computazionale debba definire sia il ruolo dell'ambiente per la coordinazione che un comportamento stocastico.

Il ruolo dell'ambiente

Il ruolo dell'ambiente in un sistema nature-inspired è di fondamentale importanza in quanto i componenti di tali sistemi possono coordinarsi, non solo direttamente, ma anche *indirettamente* attraverso l'ambiente.

L'ambiente, inoltre, è *attivo* in quanto presenta dinamiche che vanno ad influenzare la coordinazione dei componenti.

Infine l'ambiente, avendo una propria *struttura*, ha il compito di fornire agli agenti una nozione di località/vicinato, permettendogli di muoversi attraverso una topologia di qualsiasi tipo.

Comportamento stocastico

I modelli che non presentano meccanismi probabilistici non sono sufficientemente espressivi e non catturano le proprietà dei sistemi complessi presenti in natura. I sistemi computazionali che non presentano comportamenti stocastici, quindi, non si evolvono con la stessa adattatività e robustezza che presentano i sistemi naturali.

Per meglio comprendere il concetto si può pensare alle formiche, le quali quando sono in possesso di cibo rilasciano, durante il loro tragitto di ritorno verso la tana, il feromone che sarà poi utile per ritrovare la strada verso la fonte di cibo. Le formiche seguono tali tracce secondo certi meccanismi di probabilità, infatti se potessero scegliere loro stesse se seguire oppure no tale traccia di feromone in modo non deterministico, le colonie di formiche cesserebbero presto di esistere così come le conosciamo.

1.2.2 Stigmergia come sottocategoria di BIC

Quando si parla di interazione tra individui, si è portati a pensare ad un tipo di interazione esplicita, ad esempio la comunicazione linguistica o gestuale [3]. Gli esseri umani, come pure gli animali, però, sono in grado di comunicare anche senza utilizzare questi tipi di linguaggi convenzionali.

Accanto ai due tipi di comunicazione appena nominati, infatti, ve ne è una terza che viene detta *Behavioral Implicit Communication* (*BIC*), nella quale non vi sono azioni o segnali codificati in modo specifico, ma i messaggi sono il comportamento pratico in sé.

Per distinguere la *comunicazione* dalla semplice interazione è possibile definirla come *un processo nel quale le informazioni vanno dall'agente X (sender) all'agente Y con lo scopo di informare Y* [15].

Per meglio comprendere come un'azione possa avere contenuto informativo si espone il seguente esempio [15]: l'agente A sta raccogliendo dalla strada della spazzatura per metterla in un bidone; l'agente B ha lo stesso compito ma vedendo cosa sta facendo A, decide di pulire l'altro lato della strada. Poiché A sa che B lo sta osservando, le azioni pratiche che sta eseguendo possono anche essere usate come messaggio per B del tipo "*io sto pulendo qui!*". Questo tipo di comunicazione evita la negoziazione specifica per l'allocazione di task, poiché presuppone una convenzione implicita sul da farsi.

Per supportare questo tipo di comunicazione sono necessarie tre condizioni:

- L'ambiente deve permettere l'osservabilità delle azioni; un ambiente potrebbe addirittura costringere tale visibilità.
- Gli agenti devono essere in grado di comprendere ed interpretare un'azione pratica.
- Gli agenti devono essere in grado di comprendere gli effetti che le proprie azioni hanno sugli altri in modo da poter agire di conseguenza.

BIC si osserva sia dai comportamenti degli agenti, ma anche dalle loro tracce. La comunicazione attraverso le tracce ci porta ad introdurre una sottocategoria di BIC denominata *stigmergia* nella quale i destinatari non percepiscono il comportamento durante l'azione ma ne percepiscono le tracce e i risultati che ne derivano.

1.2.3 Stigmergia: cos'è, per cosa si utilizza e perché

Il concetto di *stigmergia*, come già accennato nella Sottosezione 1.1.3, fu introdotto da P.P. Grassé per spiegare il comportamento di coordinazione osservato nelle società di termiti dove *la coordinazione dei compiti e la regolazione delle costruzioni non è direttamente dipendente dai lavoratori, ma dalle costruzioni stesse* [6], definendo la stigmergia come *una classe di meccanismi che mediano le interazioni animale-animale e che sono fondamentali per raggiungere forme emergenti di comportamento coordinato al livello di società* [12].

Il concetto di stigmergia fornisce anche una spiegazione al *paradosso di coordinazione tra il livello individuale e comunitario*: guardando il comportamento di un gruppo di insetti sociali, sembra che essi stiano cooperando in modo organizzato e coordinato, mentre se si guarda un individuo singolo, sembra che stia lavorando come se fosse solo, senza interagire con gli altri e senza essere coinvolto in nessun comportamento collettivo.

La spiegazione del paradosso di coordinazione che deriva dal concetto di stigmergia è che gli insetti interagiscono indirettamente; ogni insetto influenza il comportamento degli altri insetti con la comunicazione indiretta attraverso l'ambiente.

Parlando di stigmergia se ne possono individuare due tipi [14]:

Stigmergia quantitativa: gli stimoli non differiscono qualitativamente ma modificano solo la probabilità di risposta da parte degli individui a questi stimoli.
Un esempio di questo tipo di stigmergia è dato dalla costruzione del nido da parte delle termiti. I lavoratori, per costruire il nido, utilizzano delle palline di terra impregnate di feromone che stimola gli altri lavoratori ad apporre le proprie palline di terra dove è presente il feromone, andando ad incrementare il livello di tale sostanza (rinforzo positivo); se non vi sono sufficienti individui, il feromone evapora e il compito di costruire quel nido verrà abbandonato.

Stigmergia qualitativa: gli individui interagiscono attraverso stimoli qualitativi e reagiscono ad essi. In altre parole gli individui mettono in atto azioni diverse a seconda dello stimolo con cui vengono in contatto, per esempio, un insetto risponde ad uno stimolo di tipo 1 con l'azione A e ad uno stimolo di tipo 2 con l'azione B.
Un esempio di questo tipo di stigmergia è dato dalla costruzione dell'alveare da parte delle vespe, in quanto l'architettura della costruzione fornisce stimoli di tipo diverso. Le vespe tendono ad inserire le nuove "tessere" nei posti che confinano con il maggior numero di tessere già apposte, quindi lo stimolo che deriva da un "posto vuoto" che confina con due altre tessere, sarà diverso (e meno potente) di uno che deriva da un "posto vuoto" che confina con ben tre tessere già apposte.

Nel contesto dei sistemi multi-agente la stigmergia è stata utilizzata come tecnica per risolvere problemi complessi, ma anche come approccio per la definizione e sviluppo dei sistemi.

L'utilizzo della stigmergia nei sistemi complessi è derivato dalla necessità di avere affidabilità nei sistemi e robustezza negli ambienti complessi e non predicibili, peculiarità che sono rintracciabili nei meccanismi di auto-organizzazione come la stigmergia [12].

Requisiti dell'ambiente

Nell'ambito della stigmergia, è evidente, l'ambiente gioca un ruolo fondamentale in quanto è mediatore dell'interazione tra gli agenti. L'ambiente, però, deve soddisfare determinati requisiti nei sistemi che prevedono l'auto-organizzazione [6]:

- L'ambiente deve consentire la registrazione delle tracce degli agenti così da rendere possibile la coordinazione tra gli agenti.
- L'ambiente deve agire in maniera appropriata alle emissioni di tracce implementando anche meccanismi di diffusione, aggregazione ed evaporazione delle stesse. Tali tracce, poi, devono essere rese disponibili per tutti gli agenti che abbiano la possibilità di venirne in contatto.
- L'ambiente deve caratterizzare un insieme coerente ed espressivo di astrazioni spaziali, in altre parole una topologia, che devono avere alcune nozioni di località per permettere agli agenti e alle risorse di essere associati agli spazi locali, permettendo l'interazione locale tra agenti e tra agenti ed ambiente.

1.2.4 Requisiti computazionali di un sistema stigmergico

Per realizzare un sistema computazionale che modelli il meccanismo della stigmergia è necessario che esso rispetti alcune caratteristiche che riguardano sia le reazioni dell'ambiente in risposta alle azioni dell'agente, che la struttura [5]:

- Deve essere possibile memorizzare le tracce degli agenti.
- Vi devono essere reazioni appropriate all'emissione di tali tracce; in altre parole è necessario che le tracce interagiscano con l'ambiente e che evaporino.
- Le tracce devono essere disponibili alla percezione degli altri agenti.
- L'ambiente deve avere una topologia (insieme coerente ed espressivo di astrazioni spaziali) allo scopo di descrivere e gestire la località delle azioni.

1.3 Stigmergia cognitiva

1.3.1 Coordinazione stigmergica

Nella vita di tutti i giorni, gli uomini usano regolarmente segnali mediati dall'ambiente per coordinare le proprie attività, basti pensare agli ambienti lavorativi oppure agli uffici pubblici in genere, nei quali troviamo moltissime informazioni legate alle risorse (etichette, segni, allarmi visivi) che gli uomini utilizzano

per organizzare e coordinare le loro attività sociali ed individuali utilizzando l'ambiente condiviso [8].

Queste considerazioni portano a riconoscere la *coordinazione stigmergica* come un fenomeno largamente utilizzato per l'organizzazione nelle società umane.

Quando si parla di coordinazione stigmergica [6] si parla di agenti che compiendo le proprie azioni o interagendo con l'ambiente stesso lasciano le proprie tracce (o marcatori); queste tracce vengono percepite da altri agenti, il cui comportamento verrà influenzato proprio dalle tracce stesse.

L'interazione stigmergica tra gli agenti è ciò che dà origine all'auto-organizzazione, poiché, nonostante essa avvenga a livello locale, dà origine a conseguenze percepibili a livello globale in termini di strutture e comportamenti.

La ricerca sulla coordinazione stigmergica ha portato a rilevanti e numerosi risultati anche nell'ambito dei sistemi multi-agente. Tali ricerche hanno anche evidenziato due pregiudizi sugli approcci basati sulla stigmergia:

- Generalmente il modello che viene utilizzato fa uso di agenti molto semplici, simili a formiche, che non esprimono alcuna abilità cognitiva.

- Il modello che solitamente viene utilizzato per l'ambiente è molto semplice in quanto vede le tracce di feromone (o simili) come un semplice meccanismo di evoluzione ambientale.

Seguendo la stessa linea di principio, la ricerca ha anche evidenziato il ruolo della stigmergia come meccanismo fondamentale di coordinazione mettendo in luce alcune importanti peculiarità:

- Le modifiche compiute sull'ambiente sono spesso responsabili di un'interpretazione del contesto di un sistema di segni convenzionale e condiviso.

- Gli agenti interagenti presentano abilità cognitive che possono essere utilizzate nell'interazione basata sulla stigmergia.

1.3.2 Stigmergia cognitiva

La *stigmergia cognitiva* fu introdotta come prima generalizzazione della coordinazione stigmergica che utilizza le capacità cognitive degli agenti per permettere e promuovere l'auto-organizzazione delle attività sociali. Gli agenti, in questo caso, lasciano delle tracce nell'ambiente che potranno poi essere percepite da altri agenti che, in seguito, le interpreteranno in accordo con un sistema di segni convenzionale.

Per portare un esempio chiarificatore, è come se l'ambiente fosse un muro sul quale gli agenti lasciano delle scritte (tracce) che possono essere viste dagli altri individui che passano vicino a quel muro. I sistemi sociali presentano moltissimi esempi di tracce con un significato (per esempio i segnali stradali).

Quando le tracce acquisiscono un significato universalmente riconosciuto, esse diventano *segni* e la stigmergia diventa *stigmergia cognitiva* [6].

Nella stigmergia cognitiva non vengono più utilizzati degli agenti semplici e reattivi come formiche, ma vengono utilizzati agenti razionali, eterogenei, con capacità di adattamento e sufficientemente intelligenti per interpretare correttamente i segni nell'ambiente e per reagire in maniera appropriata.

Questo tipo di stigmergia, quindi, viene detta cognitiva proprio per sottolineare la differenza negli agenti che vengono utilizzati rispetto a quelli utilizzati nella stigmergia classica, le cui capacità assomigliano a quelle degli insetti.

Come per la stigmergia classica, anche per quella cognitiva l'ambiente è di fondamentale importanza in quanto ha il ruolo di mediatore per l'interazione tra gli agenti. Il contesto generale della stigmergia cognitiva è dato da un insieme di agenti con compiti ed obiettivi specifici, che svolgono le loro attività individuali e sociali condividendo il campo di lavoro con gli altri agenti.

Inoltre, l'interazione tra gli agenti è indiretta; in questo caso, l'interazione è anche disaccoppiata nel tempo e nello spazio.

L'ambiente non è soltanto un contenitore passivo ma incapsula meccanismi e processi reattivi che permettono l'emergenza di comportamenti coordinati locali e globali. Inoltre, tale ambiente non ha solo uno stato che può essere osservato e modificato dagli agenti, ma incapsula alcune leggi che possono essere attivate dalle azioni degli agenti o, semplicemente, dal passare del tempo e che possono alterare lo stato dell'ambiente indipendentemente dalle intenzioni degli agenti.

Quindi l'ambiente può essere concepito come un insieme di strumenti con stato che forniscono funzionalità specifiche utili agli agenti che stanno svolgendo il loro lavoro individuale. Tali strumenti sono pensati allo stesso tempo per essere condivisi da tutti gli agenti.

Requisiti dell'ambiente

I requisiti dell'ambiente enunciati nella Sottosezione 1.2.3 sono gli stessi che devono essere tenuti presenti anche per l'ambiente nell'ambito della stigmergia cognitiva. Naturalmente, per far sì che gli agenti interpretino le tracce come segni in maniera corretta, potrebbe essere necessario che l'ambiente fornisca strumenti come dizionari e ontologie. Comunque, sebbene questa scelta potrebbe rinforzare la coordinazione, con la stigmergia cognitiva ciò non è strettamente necessario in quanto questo aspetto è espletato dall'intelligenza richiesta agli agenti.

1.3.3 Requisiti computazionali per la stigmergia cognitiva

Oltre ai requisiti già enunciati nella Sottosezione 1.2.4, i sistemi computazionali per la stigmergia cognitiva devono avere le seguenti caratteristiche:

- La stigmergia cognitiva accoglie strumenti che supportano gli agenti nell'interpretazione simbolica delle tracce come segni, come dizionari ed ontologie.
- L'intelligenza degli agenti è una precondizione necessaria, a differenza di ciò che accade nella stigmergia classica.

Capitolo 2

TuCSoN e Jason

2.1 Dai modelli di coordinazione tradizionali agli auto-organizzanti

Per molto tempo sono stati utilizzati modelli di coordinazione basati su regole predicibili, le quali hanno effetti conoscibili e riproducibili perfettamente.

Con l'avvento di sistemi più complessi nei quali si rendono necessari aspetti come impredicibilità, apertura, scalabilità e dinamismo, i modelli di coordinazione tradizionali non sono più sufficienti. I modelli computazionali moderni necessitano di tecniche di auto-organizzazione in grado di far emergere da un sistema le proprietà globali richieste, nonostante aspetti e comportamenti fortemente impredicibili. Vi è quindi la necessità di un sistema di coordinazione che sia facilmente e velocemente adattabile alle situazioni che vengono a crearsi.

Vi sono delle proprietà che un sistema che utilizza un *modello di coordinazione auto-organizzante* deve avere:

Topologia: i servizi sono distribuiti sui nodi della rete del sistema; ogni nodo è connesso direttamente con un insieme di nodi con i quali può interagire e scambiare informazioni. Trattandosi di sistema aperto, l'insieme deve poter cambiare in modo dinamico senza perdere le proprietà di interazione.

Località: la topologia di una rete è fortemente legata alle interazioni che si avranno tra i componenti del sistema; tali interazioni possono essere tra un agente ed un medium di coordinazione oppure tra due medium di coordinazione. Le interazioni devono sempre essere locali, quindi in uno stesso nodo della rete oppure tra due nodi vicini; in questo modo ogni individuo ha sempre una visione limitata e parziale del mondo che lo circonda.

On-line character: nei sistemi auto-organizzanti le regole di coordinazione reagiscono anche al passare del tempo e non solo nel momento in cui avviene l'interazione. I meccanismi di coordinazione possono essere visti come dei servizi eseguiti in background e sempre on-line.

Tempo: solitamente le regole di coordinazione sono legate al tempo.

Probabilità: come succede in natura, anche nei modelli di coordinazione auto-organizzanti, viene utilizzato un modello stocastico nel quale ogni azione ha una determinata probabilità di essere eseguita.

Perciò la coordinazione auto-organizzante si riferisce alla gestione delle interazioni locali con lo scopo di far emergere delle caratteristiche a livello di sistema. Questo si raggiunge distribuendo i media di coordinazione sull'ambiente, che è topologicamente connesso, i quali eseguono delle regole di coordinazione probabilistiche e temporizzate [2].

I modelli di coordinazione, inoltre, possono essere divisi in due classi:

Control-driven: questa classe comprende quei modelli che si concentrano su azioni di comunicazione regolando la topologia dell'interazione tra i componenti delle applicazioni.

Information-driven: i modelli appartenenti a questa classe si concentrano sulle informazioni che vengono scambiate all'atto della comunicazione. Il medium di coordinazione funge da spazio di dati condiviso e ciò promuove il disaccoppiamento spazio-temporale tra le entità. Fanno parte di questa categoria i modelli tuple-based (Linda e derivati).

I modelli control-driven non risultano essere la scelta migliore per entità di coordinazione fortemente autonome, dinamiche e non predicibili come gli agenti mobili. D'altra parte, l'orientamento dei modelli information-driven verso l'informazione di comunicazione fornisce un supporto più naturale alla coordinazione delle applicazioni orientate alle informazioni.

Un modello di coordinazione basato su uno spazio dati condiviso, come uno spazio di tuple Linda, fornisce molti aspetti importanti per la progettazione e lo sviluppo di sistemi orientati alle informazioni, per esempio accesso associativo alle informazioni o disaccoppiamento nel tempo e nello spazio degli agenti.

Un esempio di modello di coordinazione auto-organizzante e orientato alle informazioni (information-driven) è TuCSoN.

2.2 TuCSoN

TuCSoN (Tuple Centres Spread over the Network) è un modello utilizzato per la coordinazione di processi distribuiti, di agenti autonomi, mobili ed intelligenti.

TuCSoN fornisce un medium di coordinazione, chiamato *centro di tuple*, che non è altro che uno *spazio* di tuple arricchito con la possibilità di definire comportamenti specifici.

L'interazione attraverso lo spazio di tuple offriva il vantaggio di avere una coordinazione guidata dall'informazione: gli agenti si sincronizzavano, cooperavano e competevano basandosi sulla disponibilità delle informazioni nello spazio condiviso attraverso l'accesso associativo, la consumazione e la produzione di informazioni. Gli spazi di tuple hanno, però, il limite di permettere ai medium di coordinazione solo comportamenti prefissati; in alcuni casi, quindi, si potrebbe verificare che lo spazio di tuple non sia in grado di supportare le politiche di coordinazione richieste dallo specifico caso. Nel caso si volessero aggiungere nuove funzionalità allo spazio di tuple, allo scopo di ottenere forme di interazione più articolate, un primo possibile approccio risolutivo potrebbe consistere nell'incorporare nuove primitive che permettano di realizzare le politiche di coordinazione prima non consentite. Tuttavia, questa soluzione non si adatta allo scenario di sistema aperto, poiché richiederebbe l'accoppiamento delle entità coordinabili, le quali dovrebbero essere a conoscenza delle nuove primitive introdotte. Una

seconda possibile soluzione potrebbe consistere quindi nell'implementare specifici protocolli direttamente nei coordinabili che vengono caricati della conoscenza necessaria ad implementare la politica di coordinazione richiesta. Anche questo approccio risulta però scorretto, in quanto, costringe gli agenti ad essere coordination-aware e quindi non in grado di astrarre dai dettagli coordinativi.

Il modello TuCSoN fornisce la possibilità di mantenere l'interfaccia standard dello spazio di tuple ma potendone arricchire il comportamento [10].

Le entità che caratterizzano un sistema TuCSoN sono [9]:

Agenti: sono le entità coordinabili che sono sparse nella rete; tali entità sono intelligenti, proattive e mobili.

Centri di tuple: fungono da medium di coordinazione.

Nodi: rappresentano l'astrazione topologica di base che ospita al suo interno i centri di tupla.

Nelle prossime sezioni vediamo alcuni aspetti fondamentali del modello TuCSoN.

2.2.1 Modello di coordinazione

Il modello di coordinazione TuCSoN è pensato per supportare la progettazione e lo sviluppo di applicazioni information-oriented basate su agenti mobili che sono a conoscenza della topologia di rete.

Per far fronte alla mobilità, un nodo di una rete che vuole accettare di trattare con agenti mobili deve definire il proprio spazio di comunicazione locale, utilizzato dagli agenti per interagire con gli altri agenti e con il framework di interazione locale.

TuCSoN si focalizza sulla coordinazione dello spazio di interazione degli agenti, in altre parole si focalizza sull'interazione con le risorse d'ambiente e gli altri agenti e non intende implementare un nuovo sistema ad agenti. Al contrario TuCSoN è stato pensato per essere associato ad una infrastruttura ad agenti già esistente e, di conseguenza, assume che le funzionalità di base per l'autenticazione dell'agente, l'esecuzione e la migrazione siano già fornite [10], [11].

Le caratteristiche più rilevanti di TuCSoN sono trattate nelle prossime due sezioni.

2.2.2 Spazio di coordinazione

Lo spazio di coordinazione di TuCSoN fa affidamento su una molteplicità di astrazioni di comunicazione indipendenti, chiamate *centri di tuple* che sono diffusi per tutta la rete e utilizzati dagli agenti per interagire con altri agenti.

Lo spazio di coordinazione di TuCSoN può essere visto sia come spazio di interazione globale, definito come l'insieme di tutti i centri di tuple disponibili su ogni nodo della rete, ciascuno identificato dal suo nome completo, che come spazio di interazione locale, definito come l'insieme di tutti i centri di tuple disponibili su tutti i nodi di un singolo hosting device, e a cui si può fare riferimento evitando di specificare l'indirizzo di rete [11].

Come già detto in precedenza, un *centro di tuple* è uno *spazio di tuple* aumentato, caratterizzato da un nome localmente univoco. Ogni centro di tuple può

essere acceduto dagli agenti mobili attraverso un'interfaccia di comunicazione Linda-like.

Essendo ogni centro di tupla associato ad un nodo, ogni nodo fornisce la propria versione di spazio di comunicazione di TuCSoN.

A differenza degli spazi di tupla Linda nei quali il comportamento è fissato e non è possibile aggiungere nuove primitive e nessun nuovo comportamento può essere definito in risposta alle operazioni di comunicazione, il comportamento dei centri di tuple come canale di comunicazione può essere definito ad-hoc sulle necessità dell'applicazione. Definire un nuovo comportamento per un centro di tuple significa, essenzialmente, specificare una nuova transizione di stato in risposta ad un evento di comunicazione standard [10].

2.2.3 Linguaggio di coordinazione

Il linguaggio di coordinazione di TuCSoN definisce un'insieme di primitive di coordinazione per consentire all'agente di interagire con i centri di tuple. Il linguaggio di comunicazione utilizzato è costituito da tuple logiche.

Le operazioni di coordinazione sono composte da due fasi:

Invocation: è la fase di invio della richiesta di esecuzione da parte dell'agente verso il centro di tuple designato.

Completion: è la fase durante la quale l'agente riceve dal centro di tuple il risultato dell'operazione.

La forma generica per ogni operazione di comunicazione di TuCSoN ammissibile compiuta da un agente è:

```
tcname?operation(tuple)
```

con la quale si richiede al centro di tuple `tcname` di eseguire `operation` utilizzando `tuple` [11].

Gli agenti interagiscono scambiando tuple attraverso i centri di tuple, per mezzo di un piccolo insieme di primitive di comunicazione [9]:

- `out(T)`: inserisce la tupla `T` nel centro di tuple.
- `in(TT)`: rimuove una tupla che fa match con il template `TT` dal centro di tuple e la restituisce al chiamante. Nel caso nessuna tupla che soddisfi la richiesta sia presente nel centro di tuple, l'attività di coordinazione viene sospesa e verrà ripresa non appena una tupla che soddisfi il template si renda disponibile. Nel caso, invece, più di una tupla soddisfi il template, ne viene scelta una in modo non deterministico.
- `rd(TT)`: il funzionamento è analogo alla `in(TT)` ma in questo caso la tupla non viene cancellata dal centro.
- `inp(TT)`: è analoga alla `in(TT)` ma con l'aggiunta di una semantica di *success/failure* grazie alla quale, nel caso in cui non vi sia una tupla che corrisponda al template, si giunge direttamente ad un fallimento.
- `rdp(TT)`: è analoga alla `rd(TT)` ma con l'aggiunta di una semantica di *success/failure* grazie alla quale, nel caso in cui non vi sia una tupla che corrisponda al template, si giunge direttamente ad un fallimento.

- `in_all(TT)`: rimuove tutte le tuple che fanno match con il template e le restituisce al chiamante, nel caso non ve ne siano restituisce una lista vuota.
- `rd_all(TT)`: legge, senza rimuoverle, tutte le tuple che fanno match con il template e le restituisce al chiamante, nel caso non ve ne siano restituisce una lista vuota.
- `no_all(TT)`: controlla in una volta sola l'assenza dal centro di tuple di ogni tupla che fa match con il template specificato; in caso di successo il comportamento risulta del tutto uguale alla `no(TT)`, in caso di fallimento, vengono restituite tutte le tuple che fanno match con il template.
- `no(TT)`: l'operazione ha successo solo se nel centro non vi è alcuna tupla che faccia match con il template specificato. Nel caso vi sia una tupla che corrisponde, l'esecuzione viene sospesa e ripresa non appena il match non sussista più.
- `nop(TT)`: simile alla `no(TT)` ma non è bloccante.
- `get()`: legge tutte le tuple presenti nel centro senza rimuoverle e le restituisce in una lista al chiamante; nel caso non vi siano tuple la lista verrà restituita vuota.
- `set(LT)`: operazione che permette di inserire la tupla `LT` all'interno del centro di tuple.
- `spawn`: inizia un'attività computazionale parallela all'interno del nodo target; non ha semantica sospensiva.
- `uin`, `uinp`, `urd`, `urdp`, `uno`, `unop`: sono dette *primitive uniform* e si differenziano dalle versioni non-uniform per il fatto che la ricerca della tupla corrispondente avviene con probabilità uniforme per tutte le tuple dello spazio.

2.2.4 Artefatti come mediatori tra centri di tuple e ambiente

Vi sono sistemi o parti di sistemi che sono meglio caratterizzabili come risorse, o attrezzi, utilizzati per raggiungere obiettivi e che non hanno nè scopi interni e nemmeno dei comportamenti proattivi ma semplicemente delle funzionalità che possono essere opportunamente sfruttate come servizio. Queste entità vengono dette *artefatti* e sono esplicitamente pensate per includere e fornire determinate funzioni sfruttate dagli agenti per raggiungere i propri scopi personali o globali; in altre parole gli artefatti supportano l'esecuzione dei compiti individuali e sociali degli agenti [13].

Per far fronte alla complessità delle attività, nei sistemi complessi vengono utilizzati *artefatti di mediazione* condivisi che permettono e facilitano l'interazione tra i componenti.

L'ambiente è tipicamente composto da artefatti che costituiscono il workspace sociale. Gli artefatti sono gli strumenti che caratterizzano l'ambiente degli agenti e che gli agenti possono selezionare ed utilizzare per i propri scopi.

Nel caso dei sistemi multi-agente cognitivi a cui aspiriamo, gli artefatti possono essere caratterizzati da:

Funzione: la funzionalità che il programmatore ha voluto che l'artefatto fornisca.

Interfaccia di utilizzo: l'insieme di operazioni che gli agenti possono invocare per utilizzare l'artefatto e sfruttarne le funzionalità.

Istruzioni operative: una descrizione di come utilizzare l'artefatto per ottenerne le funzionalità.

Struttura e comportamento: si riferisce all'aspetto interno dell'artefatto, cioè all'implementazione.

Poiché lo scopo di questa tesi è realizzare degli esperimenti di stigmergia cognitiva, è necessario dire che gli artefatti in ambito della stigmergia cognitiva devono prima di tutto promuovere la *consapevolezza* (awareness), rendendo gli agenti apparentemente consapevoli di ciò che fanno gli altri agenti in modo da guidare, in qualche modo, le proprie attività.

La consapevolezza risulta essere un tassello fondamentale per l'emergenza di forme di coordinazione in ambiti nei quali non vi è un piano prestabilito che definisce ciò che gli elementi del sistema devono fare e come devono interagire tra di loro [12].

2.3 Jason

Poiché in questo elaborato si propone di realizzare esperimenti di stigmergia cognitiva, la necessità è quella di avere degli agenti intelligenti. Infatti, come già detto nella Sezione 1.3, una delle peculiarità della stigmergia cognitiva, a differenza di quella classica, è di avere degli agenti razionali e con capacità cognitive e non dei semplici agenti simili a formiche.

Per raggiungere questo obiettivo la strada percorribile è quella di utilizzare Jason che è un linguaggio utilizzato per realizzare dei sistemi multi-agente che permette di definire i comportamenti che gli agenti devono avere.

2.3.1 Agenti in Jason

Un *agente* è un'entità autonoma che mostra una certa autonomia in quanto in grado di svolgere un compito determinando quale sia il modo migliore per soddisfarlo.

Il nome "agente" deriva dal fatto che queste entità siano attive; essi sono collocati nell'ambiente che viene *percepito* dagli agenti stessi e che viene modificato attraverso le loro azioni. Gli agenti hanno, inoltre, le seguenti proprietà [1]:

Autonomia: deve essere possibile delegare dei compiti agli agenti che dovranno decidere autonomamente come risolverli seguendo solo linee generali di comportamento.

Proattività: gli agenti devono avere un comportamento goal-oriented, perciò se gli si assegna un compito, ci si aspetta che questo venga portato a termine.

Reattività: gli agenti devono reagire ai cambiamenti che avvengono nell'ambiente, poiché devono essere in grado ripianificare le loro azioni a seguito di un cambiamento nell'ambiente: i piani, nella vita reale, possono portare a problemi che devono essere risolti.

Abilità sociali: gli agenti devono essere in grado di cooperare e collaborare con gli altri agenti; per realizzare questo tipo di abilità, è necessario che gli agenti comunichino ad un livello di conoscenza, cioè siano in grado di comunicare le proprie convinzioni, i propri obiettivi e i propri piani.

Generalmente gli agenti si trovano in un ambiente condiviso nel quale, ogni agente che vi opera ha la propria sfera di influenza, cioè ogni agente riesce a controllare totalmente o parzialmente una parte di ambiente. Ovviamente le sfere di influenza degli agenti possono sovrapporsi, è quindi necessario che essi tengano presente di come le proprie azioni possano influire sul comportamento altrui. Infine, gli agenti non hanno una completa visione di tutti gli altri agenti che compongono il sistema.

2.3.2 BDI

Il modello BDI (Belief - Desire - Intention) utilizzato da Jason è stato ispirato dal comportamento umano, in particolare dal ragionamento pratico degli esseri umani. In questo modello gli agenti vengono modellati con capacità di ragionamento simili agli esseri umani. In particolare per BDI si intende [1]:

Belief - Convinzione: le convinzioni sono informazioni che gli agenti hanno sul mondo; tali informazioni potrebbero anche essere datate e/o non accurate.

Le convinzioni possono essere viste come la *componente informativa* del sistema.

Desire - Desiderio: i desideri sono tutti le possibili azioni che un agente potrebbe voler svolgere; avere un desiderio non significa necessariamente che esso verrà messo in atto, esso potrebbe, però, influenzare le azioni dell'agente. É possibile anche che un agente abbia dei desideri che siano incompatibili l'uno con l'altro. I desideri si possono considerare come opzioni per gli agenti.

I desideri possono essere considerati come la *componente motivazionale* del sistema.

Intention - Intenzione: le intenzioni sono azioni sulle quali l'agente ha deciso di lavorare. Le intenzioni potrebbero essere degli obiettivi che sono stati assegnati all'agente oppure delle opzioni di azioni possibili che l'agente può svolgere. Perciò un agente parte con degli obiettivi da svolgere e poi continua ad operare considerando quali delle opzioni disponibili siano compatibili con i propri obiettivi; le opzioni che vengono scelte sono le intenzioni che l'agente si impegna a svolgere.

Le intenzioni rappresentano la *componente decisionale* del sistema.

Il processo che porta dalle convinzioni, desideri ed intenzioni alle azioni viene detto *ragionamento pratico* (*practical reasoning*). Tale ragionamento si compone di due attività distinte [1]:

Deliberation: questa fase ha come obiettivo quello di scegliere quali goal si vogliono raggiungere; gli output di questa fase risultano essere le intenzioni che portano allo svolgimento di un'azione.

Una peculiarità delle intenzioni è quella di essere *persistenti*, cioè se un agente decide di perseguire un'intenzione esso cercherà di eseguirla; naturalmente è necessario che l'agente si renda conto se questa azione non può in alcun modo essere eseguita e, in questo caso, dovrà interrompere i tentativi di eseguirla.

La scelta di un'intenzione funge, inoltre, da filtro di ammissibilità per le intenzioni future, in quanto è possibile che ve ne siano alcune che non sono compatibili che l'intenzione che si è scelta.

Le intenzioni sono strettamente legate alle convinzioni future, poiché volere qualcosa significa credere che ciò sia possibile e che avrà successo. É però anche possibile che le intenzioni possano fallire.

Means-ends reasoning: è il processo che porta a decidere quali azioni mettere in atto per raggiungere gli obiettivi. Gli output di questa fase risultano essere i piani.

Il means-ends reasoning, quindi, è un processo che, presi in ingresso un intenzione, le convinzioni che l'agente ha sull'ambiente e le azioni disponibili, restituisce come output un piano che, se eseguito, porta al raggiungimento del goal.

2.3.3 Ciclo di ragionamento degli agenti

Il ciclo di ragionamento che un agente Jason segue si divide in dieci passi. In Figura 2.1 si possono notare diversi tipi di elementi geometrici: i rettangoli rappresentano componenti architetturali principali che determinano lo stato di un agente, i riquadri "stondati", i rombi e i cerchi rappresentano le funzioni utilizzate durante il ciclo di ragionamento. Mentre i primi due elementi rappresentano funzioni che possono essere personalizzate dal programmatore, i cerchi sono parti essenziali dell'interprete e, quindi, non possono essere modificate. I rombi, infine, denotano una funzione di selezione in quando prendono una lista di elementi e ne selezionano uno.

Andiamo ora ad analizzare i dieci passi del ciclo di ragionamento [1]. Si tenga presente che i primi quattro passi sono utilizzati dall'agente per aggiornare i propri belief.

Passo 1 - Percepire l'ambiente

Durante il ciclo di ragionamento, l'agente per prima cosa si occupa di percepire l'ambiente in modo da poter aggiornare i propri belief sullo stato dell'ambiente. Le percezioni vengono registrate come una lista di literals, ognuno di questi rappresenta, quindi, una singola percezione ed è una rappresentazione simbolica di una particolare proprietà dello stato corrente dell'ambiente. Il metodo utilizzato per ottenere tali percezioni è `perceive`.

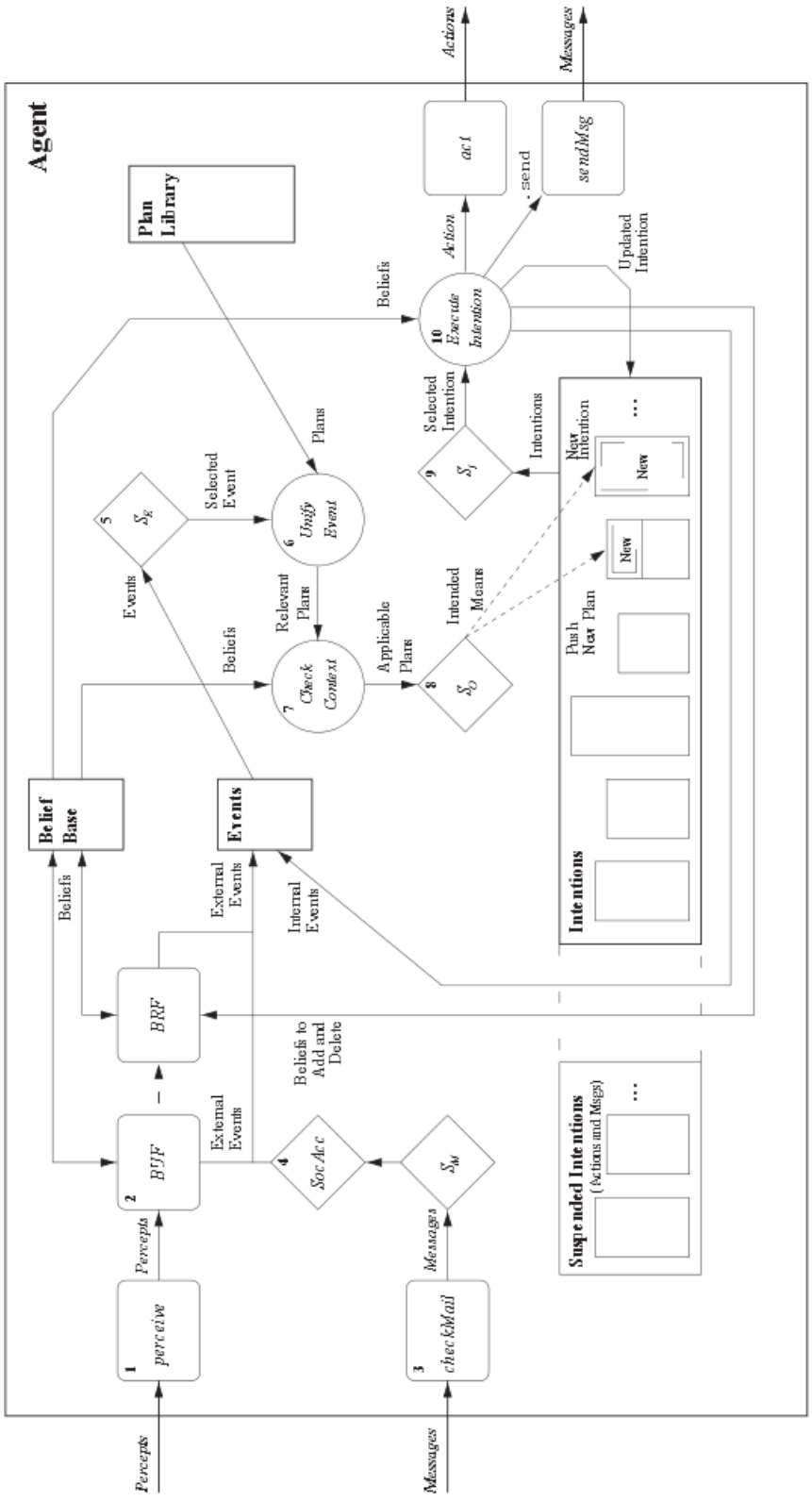

Figura 2.1: Ciclo di ragionamento di un agente Jason

Passo 2 - Aggiornamento della base di conoscenza

Dopo aver ottenuto le percezioni dall'ambiente, è necessario aggiornare la propria base di conoscenza (belief base). Questa operazione viene compiuta utilizzando un metodo chiamato `buf` che si occupa di aggiungere alla base di conoscenza le nuove percezioni e di eliminare tutte quelle che non si trovano nella lista derivante dal passo precedente.

Passo 3 - Comunicazioni dagli altri agenti

Poiché anche gli altri agenti sono fonte di informazioni, in questa fase si controlla se vi sono dei messaggi nella *mailbox* dell'agente. Questo controllo viene effettuato tramite il metodo `checkMail` che, nella sua definizione tradizionale, restituisce i messaggi ricevuti all'interno di una lista. É possibile considerare un solo messaggio a ciclo di ragionamento; tale messaggio viene scelto da S_M (funzione di selezione del messaggio) che, di default, sceglie il primo messaggio della lista.

Passo 4 - Selezione dei messaggi socialmente accettabili

Prima che i messaggi vengano processati dall'agente, questi vengono analizzati da una funzione detta *social acceptance function* implementata dal metodo `SocAcc` che determina se possano o meno essere accettati dall'agente. Questo metodo può essere personalizzato e nella sua definizione di default la funzione accetta tutti i messaggi provenienti da tutti gli agenti.

Passo 5 - Selezione di un evento

In ogni ciclo di ragionamento viene gestito un solo evento per volta anche se ve ne è più di uno pendente; questa scelta viene effettuata da una funzione detta *event selection function* (S_ε); nella sua definizione di default viene scelto il primo evento della lista.

Passo 6 - Recupero di tutti i piani rilevanti

Dopo aver selezionato un evento, è necessario trovare un piano che permetta all'agente di agire in modo da gestire l'evento. La prima cosa che viene fatta è trovare nel componente *Plan Library* tutti i piani che possono essere rilevanti per l'evento in questione. Ciò viene effettuato recuperando dalla libreria tutti i piani che hanno un triggering event che può essere unificato con l'evento selezionato.

I piani hanno la seguente sintassi:

```
triggering_event :  context <- body.
```

dove:

Triggering event: identifica l'evento che fa attivare il piano. Quando si verificano gli eventi che corrispondono alle condizioni di attivazione del piano, allora il piano diventa "attivo", cioè l'agente lo considera eseguibile.

Context: Il context contiene le condizioni che permettono all'agente di stabilire se un piano può essere eseguito. Se il context è vuoto, il piano è sempre eseguibile.

Body: contiene le azioni che l'agente deve eseguire.

Passo 7 - Scelta dei piani applicabili

Avendo ottenuto al passo precedente i piani rilevanti, è ora necessario scegliere tra questi, quelli che presentano un context che soddisfi le condizioni attuali, cioè quei piani che siano applicabili.

Passo 8 - Selezione di un piano applicabile

Il piano applicabile scelto è detto *intended means* perché il corso dell'azione definito da quel piano sarà il modo in cui l'agente intende gestire l'evento. La scelta di un piano tra quelli applicabili viene svolta dalla *applicable plan selection function* (S_O) che, di default, sceglie il primo piano trovato nella Plan Library (questo ordinamento dipende da come il programmatore ha inserito tali piani). Si noti che il piano selezionato viene inserito al top dello stack delle intention esistente, in caso di internal event, cioè se vi è un cambiamento nei goal (in altre parole se vi è un goal che deve essere eseguito prima di poter portare a termine quello in esame), o viene creata una nuova intention, in caso di external event, cioè se è stato percepito un cambiamento nell'ambiente.

Passo 9 - Selezione di una intention per un'altra esecuzione

In questo passo la *intention selection function* (S_I) seleziona una delle intention dall'insieme delle intention dell'agente.

Passo 10 - Esecuzione di una intention

In questo passo viene eseguita la prima formula del body del piano; questa formula può essere di sei tipi diversi:

Environment action: un'azione nel corpo del piano comunica all'agente che qualcosa debba avvenire nell'ambiente.

Achievement goal: è un nuovo goal da perseguire che genera, quindi, un internal event.

Test goal: questa formula è utilizzata per verificare che una determinata proprietà sia ancora nella base di conoscenza dell'agente, oppure per recuperare informazioni che sono già contenute nella base di conoscenza.

Mental notes: è una convinzione che deve essere aggiunta o eliminata dalla base di conoscenza.

Internal action: sono azioni eseguite dall'agente ma fornite altrove, per esempio in codice Java.

Expressions: sono delle semplici espressioni che devono essere valutate.

L'esecuzione di una formula comporta l'aggiornamento del set delle intention. Infine quando tutte le formule del piano sono state eseguite e quindi rimosse, allora anche l'intero piano viene rimosso dall'intention e un nuovo ciclo di ragionamento può avere inizio ricominciando dal controllo dello stato dell'ambiente.

Capitolo 3

Integrazione TuCSoN e Jason

In questo capitolo si andrà a mostrare come si è operata l'integrazione tra TuCSoN e Jason, infatti per poter utilizzare le peculiarità di entrambi i linguaggi è necessario che essi siano in grado di interagire tra di loro.

In particolare, negli esperimenti che verranno realizzati per evidenziare gli aspetti di stigmergia cognitiva di cui si è discusso precedentemente, si utilizzerà Jason per modellare al meglio gli agenti protagonisti degli esempi, mentre il centro di tuple di TuCSoN fungerà da medium per la comunicazione e coordinazione tra gli agenti.

L'intervento rilevante che è stato effettuato per permettere ai due linguaggi di interagire riguarda la creazione di una classe astratta denominata *AbstractTucsonJasonAgent* che rappresenta un agente il cui comportamento viene definito in linguaggio Jason ma in grado di interagire con l'infrastruttura TuCSoN utilizzata come medium di coordinazione. Questa classe è stata poi specializzata definendone una implementazione concreta denominata *TucsonJasonAgent* all'interno della quale viene incapsulata la gestione specifica delle primitive di coordinazione invocate dall'agente per interagire con il centro di tuple.

Inoltre, per esperimenti che necessitano di dare la possibilità di aggiungere delle percezioni agli agenti è necessario predisporre la classe *AgentsRepository* che permette di tenere traccia degli agenti presenti e che mette a disposizione dei metodi utili per aggiungere o rimuovere le percezioni.

3.1 AbstractTucsonJasonAgent

La creazione della classe astratta *AbstractTucsonJasonAgent* è stata necessaria per permettere ai linguaggi TuCSoN e Jason di poter interagire, infatti questa classe è un'estensione della classe *AgArch*, che definisce come gli agenti ricevono percezioni, eseguono azioni e comunicano con l'ambienti e gli altri agenti. La classe astratta *AbstractTucsonJasonAgent* permette di definire un agente il cui comportamento viene implementato in Jason ma che è in grado di interagire con l'infrastruttura TuCSoN. Da notare è anche che tale classe detiene una lista di *Literal* che rappresentano le percezioni dell'agente e mette a disposizione i metodi per poter andare a modificare tale lista.

La specializzazione di questa classe astratta, chiamata *TucsonJasonAgent*, ha essenzialmente il compito di fare da tramite tra gli agenti Jason e il centro

di tuple TuCSoN, infatti in questa classe vengono inizializzati i centri di tuple necessari e vengono anche definite tutte quelle primitive TuCSoN che gli agenti utilizzano per accedere al centro di tuple.

Più nello specifico in questa classe vengono definite delle azioni che l'agente Jason può utilizzare e queste corrisponderanno esattamente all'esecuzione di primitive TuCSoN sul centro di tuple. Andando, quindi, a esaminare il metodo *act* di questa classe, si possono trovare le primitive utilizzate:

out: la primitiva denominata out si comporta esattamente come l'originale TuCSoN, infatti nella sua implementazione non fa altro che richiamare la primitiva *out* di TuCSoN sul centro di tuple.

Listing 3.1: TucsonJasonAgent - out

```
if (action.getActionTerm().getFunctor().equals("out")) {
  try {
    LogicTuple tuple = LogicTuple.parse
      (action.getActionTerm().getTerm(1).toString());

    ITucsonOperation op =
      acc.out(this.getTarget(action), tuple, null);

    LogicTuple res=null;
    if (op.isResultSuccess()) {
      System.out.println("[" + myName()
      + "]: Operation succeeded.");

      res=op.getLogicTupleResult();
      System.out.println("[" + myName()
      + "]: Operation result is " + res);
    } else {
      System.out.println("[" + myName()
      + "]: Operation failed.");
    }

  //return confirming the action execution was OK
  action.setResult(true);
  feedback.add(action);

  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

in: la primitiva denominata in si comporta esattamente come l'originale TuCSoN, infatti nella sua implementazione non fa altro che richiamare la primitiva in di TuCSoN sul centro di tuple.

Listing 3.2: TucsonJasonAgent - in

```
if (action.getActionTerm().getFunctor().equals("in")) {
  try {
    LogicTuple tuple = LogicTuple.parse
      (action.getActionTerm().getTerm(1).toString());
```

```
    ITucsonOperation op =
      acc.in(this.getTarget(action), tuple, null);

    LogicTuple res = null;
    if (op.isResultSuccess()) {
      System.out.println("[" + myName()
      + "]: Operation succeeded.");

      res = op.getLogicTupleResult();
      System.out.println("[" + myName()
      + "]: Operation result is " + res);
    } else {
      System.out.println("[" + myName()
      + "]: Operation failed.");
    }

  //return confirming the action execution was OK
  action.setResult(true);
  feedback.add(action);

  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

rdp: la primitiva denominata rdp si comporta esattamente come l'originale TuCSoN, infatti nella sua implementazione non fa altro che richiamare la primitiva rdp di TuCSoN sul centro di tuple. L'unica differenza che si può notare deriva dal fatto che se nessuna tupla fa match con quella cercata, come risultato dell'operazione si ottiene *no*, questo al fine di semplificare la gestione del comportamento lato Jason dell'agente.

Listing 3.3: TucsonJasonAgent - rdp

```
if (action.getActionTerm().getFunctor().equals("rdp")) {
  try {
    LogicTuple tuple = LogicTuple.parse
      (action.getActionTerm().getTerm(1).toString());

    ITucsonOperation op =
      acc.rdp(this.getTarget(action), tuple, null);

    LogicTuple res = null;
    if (op.isResultSuccess()) {
      System.out.println("[" + myName()
      + "]: Operation succeeded.");

      res = op.getLogicTupleResult();
      System.out.println("[" + myName()
      + "]: Operation result is " + res);

      Unifier un = this.getTS().getC()
        .getSelectedIntention().peek().getUnif();
      Term toUnify = action.getActionTerm().getTerm(1);
```

```
      Structure resU = Structure.parse(res.toString());

      boolean uRes = un.unifies(toUnify, resU);


    } else {
      System.out.println("[" + myName()
      + "]: Operation failed.");
      Unifier un = this.getTS().getC()
        .getSelectedIntention().peek().getUnif();
      Term toUnify = action.getActionTerm().getTerm(1);
      boolean uRes = un.unifies(toUnify,
        Structure.parse(((Structure)toUnify)
        .getFunctor()+ "(no)"));
    }

  //return confirming the action execution was OK
  action.setResult(true);
  feedback.add(action);

  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

noOthers: la primitiva denominata noOthers è utilizzata soltanto in uno dei due esperimenti e viene richiamata dall'agente per accertarsi che non vi siano altri agenti in un determinato spazio. Tale primitiva viene utilizzata nell'esperimento che si pone come obiettivo quello di ricreare l'ambiente di un incrocio nel quale sono presenti due segnali stradali (stop e precedenza). La primitiva, quindi avrà successo solo nel caso in cui nessun altro, escludendo l'agente che richiama tale primitiva, sia presente all'interno dell'incrocio, viene quindi utilizzata la nop di TuCSoN. Nel caso, però, l'agente che richiama questa primitiva sia quello che è fermo al segnale di precedenza, se vi è un agente fermo al segnale di stop, il primo non deve tenere conto di tale agente.

Listing 3.4: TucsonJasonAgent - noOthers

```
if (action.getActionTerm().getFunctor()
  .equals("noOthers")) {
  try {

    int nDriver = 3;
    while (nDriver  > 0) {
      for (int i=1; i <= CrossroadModel.AGENTS_N; i++) {

        String curr = "driver" + i;
        if ( !curr.equals(this.myName()) ) {
          LogicTuple tuple =
            LogicTuple.parse("driver(" + curr + ",R)");
          ITucsonOperation op =
            acc.nop(this.getTarget(action), tuple, null);
            if (op.isResultSuccess()) {
              nDriver--;
```

```
            }else{
              tuple =
                LogicTuple.parse("stopped("+ curr + ")");
              op = acc
                .rdp(this.getTarget(action),tuple,null);
              if(op.isResultSuccess()){
                nDriver--;
              }
            }
          }
        }
        if(nDriver > 0) {
          nDriver = 3;
          this.sleep();
        }
      }

    //return confirming the action execution was OK
    action.setResult(true);
    feedback.add(action);

    } catch (Exception e) {
      e.printStackTrace();
    }
}
```

inp: la primitiva denominata inp si comporta esattamente come l'originale TuCSoN, infatti nella sua implementazione non fa altro che richiamare la primitiva inp di TuCSoN sul centro di tuple. Questa primitiva è stata definita per il secondo esperimento e l'unica differenza che si può notare deriva dal fatto che se nessuna tupla fa match con quella cercata, come risultato dell'operazione si ottengono *-1* e *none*, questo al fine di semplificare la gestione del comportamento lato Jason dell'agente.

Listing 3.5: TucsonJasonAgent - inp

```
if (action.getActionTerm().getFunctor().equals("inp")) {
  try {
    LogicTuple tuple = LogicTuple.parse
      (action.getActionTerm().getTerm(1).toString());

    ITucsonOperation op =
      acc.inp(this.getTarget(action), tuple, null);

    LogicTuple res = null;
    if (op.isResultSuccess()) {
      System.out.println("[" + myName()
      + "]: Operation succeeded.");

      res = op.getLogicTupleResult();
      System.out.println("[" + myName()
      + "]: Operation result is " + res);


      Unifier un = this.getTS().getC()
```

```
        .getSelectedIntention().peek().getUnif();
      Term toUnify = action.getActionTerm().getTerm(1);

      Structure resU = Structure.parse(res.toString());

      boolean uRes = un.unifies(toUnify, resU);


    } else {
      System.out.println("[" + myName()
      + "]: Operation failed.");
      Unifier un = this.getTS().getC()
        .getSelectedIntention().peek().getUnif();
      Term toUnify = action.getActionTerm().getTerm(1);
      boolean uRes = un.unifies(toUnify, Structure
        .parse(((Structure)toUnify)
        .getFunctor()+ "(-1,none)"));
    }

  //return confirming the action execution was OK
  action.setResult(true);
  feedback.add(action);

  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

3.2 AgentsRepository

La classe *AgentsRepository* è necessaria per poter aggiungere delle percezioni agli agenti; infatti, dovendo integrare Jason e TuCSoN non è possibile aggiungere o rimuovere le percezioni degli agenti in modo classico ma è necessario utilizzare questa classe.

Per assicurarsi che una sola istanza di tale classe venga creata, si è deciso di utilizzare il pattern *Singleton* e per questo motivo la classe *AgentsRepository* ha il costruttore privato e un metodo *getter* statico che crea un'istanza della classe la prima volta che viene richiamato e, per le successive invocazioni di questo metodo, continua a restituire sempre la stessa istanza.

Listing 3.6: AgentsRepository - pattern Singleton

```
public synchronized static AgentsRepository getRepository(){
  if(AgentsRepository.repo == null) {
    AgentsRepository.repo = new AgentsRepository();
  }
  return AgentsRepository.repo;
}

private AgentsRepository() {
  this.agents =
    new HashMap<String, AbstractTucsonJasonAgent>();
```

```
}
```

La classe *AgentsRepository* incapsula una HashMap che ha come chiave il nome dell'agente e come valore il riferimento all'istanza dell'architettura specifica *AbstractTucsonJasonAgent* di esso; gli agenti vengono aggiunti all'HashMap al momento della loro inizializzazione (metodo *init* della classe *AbstractTucsonJasonAgent*) richiamando il metodo *putAgArch*.

Listing 3.7: AgentsRepository - putAgArch

```
public void putAgArch(  String agName,
            AbstractTucsonJasonAgent agArch){
  this.agents.put(agName, agArch);
}
```

Grazie all'HashMap si è in grado di accedere, aggiungere ed eliminare le percezioni degli agenti. In particolare per aggiungere una percezione ad un dato agente si utilizza il metodo *addPercept*.

Listing 3.8: AgentsRepository - addPercept

```
public void addPercept(String agName, Literal p) {
  this.agents.get(agName).addPercept(p);
}
```

Col quale si inserisce la percezione p nella base di conoscenza dell'agente *agName*, grazie al metodo *addPercept* della classe *AbstractTucsonJasonAgent*.

Al contrario, per ripulire l'agente dalle percezioni del ciclo di ragionamento precedente si utilizza il metodo *clearPercepts*.

Listing 3.9: AgentsRepository - clearPercepts

```
public void clearPercepts(String agName) {
  this.agents.get(agName).clearPercepts();
}
```

Col quale si richiama il metodo *clearPercepts* della classe *AbstractTucsonJasonAgent*, infatti come detto precedentemente, essa possiede una lista di Literal che rappresentano le percezioni dell'agente.

Capitolo 4

Esperimenti di Stigmergia Cognitiva

In questo capitolo verranno esposti gli esperimenti che sono stati messi in atto per meglio mostrare quanto descritto nei precedenti capitoli. In particolare, si vedrà come gli agenti siano in grado di reagire in maniera adeguata a segni in relazione al loro colore e alla loro forma.

Il primo esperimento sviluppato si pone come obiettivo quello di ricreare la situazione che si viene a creare ad un incrocio stradale nel momento in cui alcuni veicoli vengono posti di fronte a dei segnali stradali che devono essere interpretati in modo tale che gli automobilisti possano agire conseguentemente ed adeguatamente.

Il secondo esperimento, invece, vuole ricreare un ambiente lavorativo nel quale il capo ufficio desidera assegnare dei compiti ai propri impiegati solo con l'ausilio di post-it colorati; ognuno di essi ha un significato specifico che i dipendenti conoscono, perciò riusciranno a reagire adeguatamente nel momento in cui siano presenti nuovi post-it nel luogo designato alla loro deposizione.

4.1 Scopo degli esperimenti

Lo scopo degli esperimenti che verranno descritti in seguito è quello di mostrare con esempi pratici cosa significhi *stigmergia cognitiva*.

Come già detto nella Sottosezione 1.3.2, quando le tracce acquisiscono un significato universalmente riconosciuto, esse diventano *segni* e la stigmergia diventa *stigmergia cognitiva*.

Negli esperimenti, quindi, verranno mostrati dei segni che sono inseriti nell'ambiente da alcuni agenti e che sono universalmente riconosciuti da altri, i quali sono in grado di interpretarli ed agire conseguentemente.

4.2 Esperimento 1 - Crossroad Casestudy

4.2.1 Descrizione

Nell'esperimento denominato *Crossroad Casestudy* viene mostrato un incrocio di strade, ognuna delle quali presenta una corsia per senso di percorrenza. Vi

sono inoltre quattro agenti (automobili) che percorrono le strade ciclicamente.

In questo esperimento i segni sono rappresentati da due diversi tipi di segnali stradali: un segnale di stop e un segnale di precedenza posti all'incrocio sulla strada verticale. Gli agenti devono, ovviamente, reagire differentemente ai due tipi di segnali: al segnale di stop devono fermarsi anche nel caso in cui non vi sia nessuno; al segnale di precedenza, l'agente arresta il suo movimento solo se da destra o da sinistra sta giungendo un'altra automobile (Figura 4.1).

L'obiettivo di questo esperimento è quello di mostrare che gli agenti sono in grado di agire adeguatamente grazie al fatto che interpretano nella maniera opportuna le forme e i colori dei segnali che incontrano sul loro cammino.

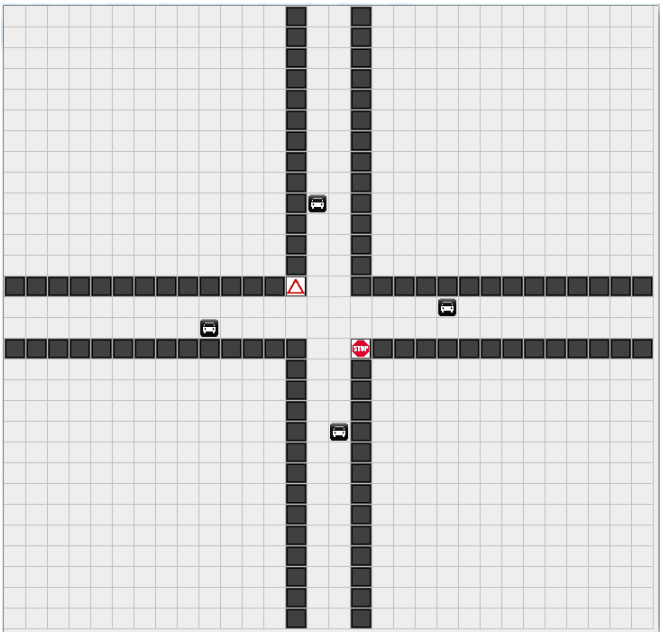

Figura 4.1: Esperimento crossroad all'avvio

4.2.2 Implementazione

L'implementazione di questo esperimento si articola in una parte basata su classi Java ed una parte basata su Jason.

Per quanto riguarda la parte basata sul linguaggio Java, le classi sono state organizzate in tre package: il package denominato *crossroad* nel quale si trovano le classi che si riferiscono all'ambiente utilizzato, nel package *crossroad.agent*, invece, si trovano le classi che si riferiscono agli agenti ed, infine, nel package *crossroad.internalaction* si trovano le classi che definiscono le internal action utilizzate dagli agenti Jason.

Package crossroad

La classe che rappresenta il modello della scena, chiamata *CrossroadModel*, estende la classe *GridWorldModel* che permette di creare una griglia delle di-

mensioni desiderate e di definire gli ostacoli e gli agenti necessari per creare l'ambiente dell'esperimento che si vuole mostrare.

Per assicurarsi che una sola istanza di tale classe venga creata, si è deciso di utilizzare il pattern *Singleton* e per questo motivo la classe *CrossroadModel* ha il costruttore privato e un metodo *getter* statico che crea un'istanza della classe la prima volta che viene richiamato e, per le successive invocazioni di questo metodo, continua a restituire sempre la stessa istanza.

Inoltre la classe *CrossroadModel* rende disponibili dei metodi utili per il movimento degli agenti.

Il metodo *move* viene utilizzato per far muovere l'agente, a seconda della direzione scelta, egli si muoverà di un punto sulla griglia, sempre che questo non sia occupato da un ostacolo.

Listing 4.1: CrossroadModel - move

```
boolean move(Move dir, int ag) throws Exception {
  Location l = getAgPos(ag);
    switch (dir) {
    case UP:
        if (isFree(l.x, l.y - 1)) {
          setAgPos(ag, l.x, l.y - 1);
        }
        break;
    case DOWN:
        if (isFree(l.x, l.y + 1)) {
            setAgPos(ag, l.x, l.y + 1);
        }
        break;
    case RIGHT:
        if (isFree(l.x + 1, l.y)) {
            setAgPos(ag, l.x + 1, l.y);
        }
        break;
    case LEFT:
        if (isFree(l.x - 1, l.y)) {
            setAgPos(ag, l.x - 1, l.y);
        }
        break;
    case STAY:
      setAgPos(ag, l.x, l.y);
      break;
    }
    return true;
}
```

Per impostare o recuperare la posizione dei segnali presenti nell'ambiente, inoltre, sono stati definiti appositi metodi *setter* e *getter*.

Listing 4.2: CrossroadModel

```
public Location getStopSign() {
  return stopSign;
}

public Location getPrecSign() {
```

```
    return precSign;
}

public void setStopSign(int x, int y) {
    stopSign = new Location(x, y);
    data[x][y] = STOP_ROAD_SIGN;
}

public void setPrecSign(int x, int y) {
    precSign = new Location(x, y);
    data[x][y] = PREC_ROAD_SIGN;
}
```

Ai fini del movimento dell'agente, è necessario che esso sia a conoscenza della strada in cui si trova; nell'esperimento il circuito stradale è stato suddiviso in quattro differenti strade e un incrocio, rappresentato dal quadrato centrale. Il metodo *getRoadName* restituisce proprio il nome della strada nella quale si trova l'agente in quel momento.

Listing 4.3: CrossroadModel - getRoadName

```
public String getRoadName (int x, int y){
    for (int i = 0; i < crossroad.length; i++) {
    if(x == crossroad[i].x && y == crossroad[i].y){
      return "crossroad";
    }
  }
    if(x==15){
      for (int i = 0; i < downToUp.length; i++) {
        if(y == downToUp[i].y){
          return "road1";
        }
      }
    }else if(x==14){
      for (int i = 0; i < upToDown.length; i++) {
        if(y == upToDown[i].y){
          return "road2";
        }
      }
    }
    if(y==15){
      for (int i = 0; i < leftToRight.length; i++) {
        if(x == leftToRight[i].x){
          return "road3";
        }
      }
    }else if(y==14){
      for (int i = 0; i < rightToLeft.length; i++) {
        if(x == rightToLeft[i].x){
          return "road4";
        }
      }
    }
    return "";
}
```

Poiché si vuole che gli agenti si muovano ciclicamente, quando uno di essi raggiunge l'estremo dell'ambiente esso dovrà uscire dalla mappa e verrà fatto "rientrare" solo dopo un secondo. Per poter effettuare questa operazione è necessario che l'agente sia consapevole del raggiungimento della fine della strada, ciò viene svolto tramite il metodo *isRoadEnded*.

Listing 4.4: CrossroadModel - isRoadEnded

```
public Boolean isRoadEnded(String road, int x, int y) {
  if( (road.equals("road1") && x == 15 && y == 0) ||
    (road.equals("road2") && x == 14 && y == 29) ||
    (road.equals("road3") && x == 29 && y == 15) ||
    (road.equals("road4") && x == 0 && y == 14)){
      return true;
  }
  return false;
}
```

A tal fine, per impostare la visibilità e per conoscerne lo stato è risultato necessario definire dei metodi che agiscono su di una struttura *HashMap* (*agVisibility*) che memorizza se un agente è da visualizzare oppure no all'interno della mappa.

Listing 4.5: CrossroadModel

```
public boolean goOut(int agId) {
  agVisibility.put(agId, false);
  return true;
}

public boolean goIn(int agId) {
  agVisibility.put(agId, true);
  return true;
}

public boolean isVisible(int agId) {
  if(agVisibility.get(agId)) {
    return true;
  }
  return false;
}
```

E' inoltre utile, ai fini della simulazione, poter sapere quando l'agente si trova all'interno dell'area dell'incrocio. A tale scopo è stato definito il metodo *isInsideCrossroad* che, oltre a dare questa informazione, restituisce anche una stringa rappresentante l'azione specifica che l'agente sta effettuando, ovvero se sta entrando nell'area dell'incrocio oppure ne sta uscendo.

Listing 4.6: CrossroadModel - isInsideCrossroad

```
public String isInsideCrossroad(String road, int x, int y) {
  if( (x == 15 && y == 20 && road.equals("road1")) ||
    (x == 14 && y == 9 && road.equals("road2")) ||
    (x == 9 && y == 15 && road.equals("road3")) ||
    (x == 20 && y == 14 && road.equals("road4"))){
      return "enter";
```

```
  }

  if( (x == 15 && y == 13 && road.equals("road1")) ||
    (x == 14 && y == 16 && road.equals("road2")) ||
    (x == 16 && y == 15 && road.equals("road3")) ||
    (x == 13 && y == 14 && road.equals("road4"))){
      return "exit";
  }
  return "";
}
```

Poiché il comportamento degli agenti dipende fortemente dal fatto che siano, oppure no, presenti dei segnali stradali lungo la strada che percorrono, è stato necessario definire un metodo, denominato *isNearRoadSignal* che ritorna *true* nel caso l'agente sia vicino al punto in cui potrebbe esservi un segnale stradale. Da notare che questo metodo non comunica all'agente se vi sia oppure no un segnale stradale, ma solo se egli si trova in prossimità del punto in cui *potrebbe* trovarsene uno.

Listing 4.7: CrossroadModel - isNearRoadSignal

```
public Boolean isNearRoadSignal(String road, int x, int y) {
  if( (x == 15 && y == 16 && road.equals("road1")) ||
    (x == 14 && y == 13 && road.equals("road2")) ||
    (x == 13 && y == 15 && road.equals("road3")) ||
    (x == 16 && y == 14 && road.equals("road4"))){
      return true;
  }
  return false;
}
```

Infine, il metodo *world* è stato utilizzato per definire gli elementi che compongono l'ambiente, come gli ostacoli, le strade, gli agenti. In particolare se ne definiscono le posizioni che essi devono avere.

Listing 4.8: CrossroadModel - world

```
static CrossroadModel world() throws Exception {
    CrossroadModel model = CrossroadModel.create(30, 30, 4);

    downToUp = new V2d[30];
    for (int y = 0; y < downToUp.length; y++) {
        downToUp[y] = new V2d(15, y);
  }
    upToDown = new V2d[30];
    for (int y = 0; y < upToDown.length; y++) {
        upToDown[y] = new V2d(14, y);
  }
    leftToRight = new V2d[30];
    for (int x = 0; x < leftToRight.length; x++) {
        leftToRight[x] = new V2d(x, 15);
  }
    rightToLeft = new V2d[30];
    for (int x = 0; x < rightToLeft.length; x++) {
        rightToLeft[x] = new V2d(x, 14);
```

```
  }
    crossroad = new V2d[]{  new V2d(14,14),
                new V2d(14,15),
                new V2d(15,14),
                new V2d(15,15)};

    model.setId("Scenario 1");

    model.setStopSign(16, 16); //SEGNALI
    model.setPrecSign(13, 13);

    model.add(CrossroadModel.OBSTACLE, 13, 0);
    ...
    model.add(CrossroadModel.OBSTACLE, 29, 16);

    model.setAgPos(0, 14, 0);
    model.setAgPos(1, 22, 14);
    model.setAgPos(2, 15, 24);
    model.setAgPos(3, 5, 15);
    agVisibility.put(0, true);
    agVisibility.put(1, true);
    agVisibility.put(2, true);
    agVisibility.put(3, true);

    return model;
}
```

La seconda classe che si trova nel package *crossroad* è *CrossroadView* che si occupa di inizializzare i componenti grafici e di mettere a disposizione i metodi che permettono di disegnare i vari elementi presenti nella scena come gli agenti e i segnali stradali. In particolare troviamo quattro diversi metodi utilizzati per disegnare elementi differenti.

Il metodo *draw*, che va a sovrascrivere il metodo della classe parent di *CrossroadView* (cioè *GridWorldView*), viene utilizzato per discriminare se l'oggetto che si desidera disegnare è un segnale di *stop* oppure di *precedenza*.

Listing 4.9: CrossroadView - draw

```
@Override
public void draw(Graphics g, int x, int y, int object) {
  switch (object) {
  case CrossroadModel.STOP_ROAD_SIGN: drawStopSign(g, x, y);
  break;
  case CrossroadModel.PREC_ROAD_SIGN: drawPrecSign(g, x, y);
  break;
  }
}
```

Anche il metodo *drawAgent* sovrascrive il metodo già definito nella classe *GridWorldView* in quanto si desidera visualizzare l'agente con l'immagine di un'automobile Figura 4.2.

Figura 4.2: Immagine dell'agente automobile

Listing 4.10: CrossroadView - drawAgent

```
@Override
public void drawAgent(  Graphics g, int x, int y,
            Color c, int id) {
  if(( (CrossroadModel) model).isVisible(id)){
    File img = new File("resources/images/car.png");
    BufferedImage bimg = null;
    try {
      bimg = ImageIO.read(img);
    } catch (IOException e) {
      e.printStackTrace();
    }
    g.drawImage(bimg, x * cellSizeW + 2, y * cellSizeH + 2,
          cellSizeW - 4, cellSizeW - 4, null);
  }
}
```

Infine, per disegnare i segnali stradali sono stati utilizzati due metodi molto simili a quello appena mostrato in quanto differiscono solo per il tipo di immagine che è stata inserita: uno dei due va ad inserire un segnale di stop (Figura 4.3), mentre l'altro va ad inserirne uno di precedenza (Figura 4.4).

Figura 4.3: Segnale di Stop

Figura 4.4: Segnale di Precedenza

Listing 4.11: CrossroadView

```
private void drawStopSign(Graphics g, int x, int y) {
  File img = new File("resources/images/stopsign.jpg");
  BufferedImage bimg = null;
  try {
    bimg = ImageIO.read(img);
  } catch (IOException e) {
    e.printStackTrace();
  }
  g.drawImage(bimg, x * cellSizeW + 2, y * cellSizeH + 2,
        cellSizeW - 4, cellSizeW - 4, null);
}

private void drawPrecSign(Graphics g, int x, int y) {
  File img =
    new File("resources/images/precedencesign_rot180.jpg");
  BufferedImage bimg = null;
  try {
    bimg = ImageIO.read(img);
  } catch (IOException e) {
    e.printStackTrace();
  }
  g.drawImage(bimg, x * cellSizeW + 2, y * cellSizeH + 2,
        cellSizeW - 4, cellSizeW - 4, null);
}
```

Infine, nel package *crossroad* si trova la classe *CrossroadEnv* che estende la classe *Environment* e, oltre ad inizializzare Model e View, si occupa anche di richiamare il movimento corrispondente nel Model dipendentemente dal movimento scelto dall'agente tramite il metodo *executeAction*.

Listing 4.12: CrossroadEnv

```
@Override
public boolean executeAction(String agName, Structure act) {
  boolean result = false;
    try {
      int agId = getAgIdBasedOnName(agName);

        if (act.getFunctor().equals("moveUp")) {
          result = model.move(Move.UP, agId);
        } else if (act.getFunctor().equals("moveDown")) {
            result = model.move(Move.DOWN, agId);
        } else if (act.getFunctor().equals("moveRight")) {
            result = model.move(Move.RIGHT, agId);
        } else if (act.getFunctor().equals("moveLeft")) {
            result = model.move(Move.LEFT, agId);
        } else if (act.getFunctor().equals("stay")){
            result = model.move(Move.STAY, agId);
        } else if (act.getFunctor().equals("goOut")){
            result = model.goOut(agId);
        } else if (act.getFunctor().equals("goIn")){
            result = model.goIn(agId);
        } else {
            logger.info("executing: " + act + ",
```

```
                    but not implemented!");
            }
        } catch (InterruptedException e) {
        } catch (Exception e) {
            logger.log(Level.SEVERE, "error executing "
                    + act + " for " + agName, e);
        }
        return result;
}
```

Package crossroad.agent

All'interno di questo package si trovano le classi *AbstractTucsonJasonAgent* e *TucsonJasonAgent* di cui si è discusso nella Sezione 3.1.

Nella classe *TucsonJasonAgent* sono stati definiti cinque centri di tuple: uno per ogni strada e uno per il crossroad. Tali centri verranno utilizzati per "segnalare" la presenza di un agente oppure di un segnale stradale, come verrà descritto in seguito.

Listing 4.13: TucsonJasonAgent - Centri di tuple

```
crossroadTC =
  new TucsonTupleCentreId("crossroadTC", this.ip, this.port);
road1TC =
  new TucsonTupleCentreId("road1TC", this.ip, this.port);
road2TC =
  new TucsonTupleCentreId("road2TC", this.ip, this.port);
road3TC =
  new TucsonTupleCentreId("road3TC", this.ip, this.port);
road4TC =
  new TucsonTupleCentreId("road4TC", this.ip, this.port);
```

Inoltre, in questa classe, sono state implementate solo alcune delle primitive di coordinazione discusse nella Sezione 3.1: `out`, `in`, `rdp` e `noOthers`.

Package crossroad.internalaction

Nel package denominato *crossroad.internalaction* sono state inserite le classi che definiscono delle azioni interne che vengono utilizzate dall'agente Jason.

La prima internal action, denominata *checkInsideCrossroad*, viene utilizzata dall'agente Jason per sapere se sta entrando, o uscendo, dall'area considerata *crossroad* oppure se si trova in posizioni neutrali. Questa internal action utilizza il metodo *isInsideCrossroad* definito nella classe *CrossroadModel* che, come già detto, restituisce una stringa "enter" nel caso in cui l'agente stia entrando nell'area crossroad, "exit" nel caso ne stia uscendo e una stringa vuota altrimenti.

Listing 4.14: Internal action - checkInsideCrossroad

```
public class checkInsideCrossroad   extends
                    DefaultInternalAction {
  @Override
  public Object execute(  TransitionSystem ts,
              Unifier un,
              Term[] terms) throws Exception {
```

```
    try {
      CrossroadModel model = CrossroadModel.get();
      int agId =
        this.getAgIdBasedOnName(terms[0].toString());
      String road = terms[1].toString();
      Location l = model.getAgPos(agId);

      String dir = model.isInsideCrossroad(road, l.x, l.y);

      return un.unifies(terms[2], new Atom(dir.toString()));
    } catch (Throwable e) {
      e.printStackTrace();
      return false;
    }
  }

  private int getAgIdBasedOnName(String agName) {
        return (Integer.parseInt(agName.substring(6))) - 1;
    }
}
```

La seconda internal action, denominata *getRoad*, viene utilizzata per sapere in quale strada si trova l'agente; per conoscere questa informazione viene utilizzato il metodo *getRoadName* della classe *CrossroadModel* che restituisce, in formato di stringa, il nome della strada. Nel caso in cui la stringa restituita sia "crossroad" il valore restituito da questa internal action non viene aggiornato con una nuova stringa ma si utilizza l'ultima strada conosciuta dall'agente.

Listing 4.15: Internal action - getRoad

```
public class checkInsideCrossroad   extends
                  DefaultInternalAction {
  @Override
  public Object execute(  TransitionSystem ts,
              Unifier un,
              Term[] terms) throws Exception {
    try {
      CrossroadModel model = CrossroadModel.get();

      int agId =
        this.getAgIdBasedOnName(terms[0].toString());
      if(!model.isVisible(agId)) model.goIn(agId);
      Location l = model.getAgPos(agId);

      String road = model.getRoadName(l.x, l.y);

      if(road.equals("crossroad")){
        return un.unifies(terms[2], terms[1]);
      }

      return un.unifies(terms[2], new Atom(road));
    } catch (Throwable e) {
      e.printStackTrace();
      return false;
    }
```

```
  }

  private int getAgIdBasedOnName(String agName) {
        return (Integer.parseInt(agName.substring(6))) - 1;
    }
}
```

La terza internal action, denominata *getRoadId*, viene utilizzata all'avvio dell'agente Jason per assegnare alle strade lo stesso valore numerico dell'agente che vi si trova. Naturalmente, nel caso vi sia la necessità di utilizzare più agenti, questa internal action può essere tranquillamente sostituita da *getRoad*, apportando solo qualche piccola modifica.

Listing 4.16: Internal action - getRoadId

```
public class checkInsideCrossroad   extends
                    DefaultInternalAction {
  @Override
  public Object execute(  TransitionSystem ts,
              Unifier un,
              Term[] terms) throws Exception {
    try {
      int agId =
        this.getAgIdBasedOnName(terms[0].toString());

      return un.unifies(terms[1], new Atom("road" + agId));
    } catch (Throwable e) {
      e.printStackTrace();
      return false;
    }
  }

  private int getAgIdBasedOnName(String agName) {
        return (Integer.parseInt(agName.substring(6))) - 1;
    }
}
```

La quarta internal action, denominata *isNearRoadSignalPoint*, dati la posizione dell'agente e la strada sulla quale si trova, richiama il metodo *isNearRoadSignal* definito nella classe *CrossRoadModel* che risponde positivamente (*true*) solo se l'agente si trova nel punto in cui si richiede che esso abbia la reazione adeguata nel caso in cui sia presente un segnale stradale.

Listing 4.17: Internal action - isNearRoadSignalPoint

```
public class checkInsideCrossroad   extends
                    DefaultInternalAction {
  @Override
  public Object execute(  TransitionSystem ts,
              Unifier un,
              Term[] terms) throws Exception {
    try {
      CrossroadModel model = CrossroadModel.get();

      int agId =
```

```
        this.getAgIdBasedOnName(terms[0].toString());
      String road = terms[1].toString();
      Location l = model.getAgPos(agId);

      Boolean near = model.isNearRoadSignal(road, l.x, l.y);

      return
        un.unifies(terms[2], new Atom(near.toString()));
    } catch (Throwable e) {
      e.printStackTrace();
      return false;
    }
  }

  private int getAgIdBasedOnName(String agName) {
        return (Integer.parseInt(agName.substring(6))) - 1;
    }
}
```

Infine, l'internal action denominata *isRoadEnded*, viene utilizzata per sapere se l'agente che la richiama si trova alla fine della strada che sta percorrendo. Questa informazione viene dedotta tramite l'omonimo metodo presente nella classe *CrossroadModel*. Sapere se l'agente è oppure no alla fine della strada su cui si trova risulta utile in quanto si è deciso che, quando esso raggiunge la fine della strada, deve uscire dalla mappa per un certo periodo di tempo, per poi rientrarvi sulla strada adiacente a quella appena percorsa. Questa decisione è stata presa per rendere la visione dell'esperimento potenzialmente infinita, senza avere la necessità di interrompere la simulazione e farla ricominciare dall'inizio.

Listing 4.18: Internal action - isRoadEnded

```
public class checkInsideCrossroad   extends
                    DefaultInternalAction {
  @Override
  public Object execute(  TransitionSystem ts,
              Unifier un,
              Term[] terms) throws Exception {
    try {
      CrossroadModel model = CrossroadModel.get();

      int agId =
        this.getAgIdBasedOnName(terms[0].toString());
      String road = terms[1].toString();
      Location l = model.getAgPos(agId);

      Boolean ended = model.isRoadEnded(road, l.x, l.y);

      return
        un.unifies(terms[2], new Atom(ended.toString()));
    } catch (Throwable e) {
      e.printStackTrace();
      return false;
    }
  }
```

```
  private int getAgIdBasedOnName(String agName) {
        return (Integer.parseInt(agName.substring(6))) - 1;
    }
}
```

Agenti Jason

Per quanto riguarda gli agenti Jason, ne sono stati definiti due tipi: *boot* e *driver*.

L'agente *boot* viene utilizzato esclusivamente per immettere le tuple che corrispondono ai segnali stradali nei centri di tuple delle strade su cui essi si trovano. Esso rappresenta dunque un agente che si occupa dell'inizializzazione delle tuple nei centri di tuple, per poi terminare immediatamente la sua esecuzione.

Listing 4.19: boot Agent

```
/* Initial goals */

  !boot.

/* Plans */

  +!boot :true <-
    .my_name(Name);
    out(road1,signal(stop));
    out(road2, signal(prec));
    .kill_agent(Name).
```

L'agente *driver*, invece, rappresenta l'automobilista che si muove all'interno dell'ambiente. Appena creato, questo agente applica il piano *!start*, tramite il quale si occupa di recuperare il nome della strada su cui esso si trova, tramite l'internal action *getRoadId*, ed inserisce questa informazione nella propria base di conoscenza. In seguito l'agente applica il piano *!move*.

Listing 4.20: Driver - !start

```
+!start : true <-
    .my_name(N);
    .print("Hello i'm ",N);
    crossroad.internalaction.getRoadId(N, RoadId);
    +myRoad(RoadId);
    !move.
```

Il piano *!move* rappresenta il movimento ciclico compiuto dall'agente; innanzitutto controlla in quale strada si trova e aggiorna la base di conoscenza, dopodiché effettua il movimento ed in seguito controlla se si trova all'interno dell'incrocio, se è vicino ad un segnale stradale e se si trova alla fine della strada percorsa. Queste operazioni vengono svolte tramite dei piani che verranno descritti in seguito.

Listing 4.21: Driver - !move

```
+!move : true <-
    !getRoad;
    !doMove;
    .my_name(N);
    ?myRoad(RoadId);
    !checkInsideCrossroad(N, RoadId);
    !checkNearRoadSignal(N, RoadId);
    crossroad.internalaction.isRoadEnded(N, RoadId, Bool);
    !checkRoadEnded(Bool, RoadId);
    !move.
```

Tramite il piano *!getRoad*, l'agente ispeziona la base di conoscenza alla ricerca dell'identificativo attuale della strada, elimina questo dato e, grazie all'internal action *crossroad.internalaction.getRoad*, si occupa di effettuarne l'aggiornamento.

Listing 4.22: Driver - !getRoad

```
+!getRoad : true <-
      .my_name(N);
      -myRoad(RoadId);
      crossroad.internalaction.getRoad(N,RoadId,R);
      +myRoad(R).
```

Tramite il piano *!doMove*, l'agente si occupa di effettuare il movimento all'interno dell'ambiente. Poiché l'identificativo della strada determina anche il tipo di movimento che l'agente deve effettuare, a seconda della strada sulla quale esso si trova, dovrà effettuare un movimento diverso.

Listing 4.23: Driver - !doMove

```
+!doMove : myRoad(road1) <-
      moveUp.
+!doMove : myRoad(road2) <-
      moveDown.
+!doMove : myRoad(road3) <-
      moveRight.
+!doMove : myRoad(road4) <-
      moveLeft.
```

Come precedentemente accennato, ad ogni movimento dell'agente, esso deve effettuare alcuni controlli. Uno di questi è relativo al fatto che l'agente stia entrando oppure uscendo dall'incrocio. Tale controllo viene svolto dall'agente grazie al piano denominato *!checkInsideCrossroad*, il quale, dopo aver utilizzato l'internal action omonima, richiama un piano che a seconda del valore di ritorno dell'internal action inserisce o elimina la tupla corrispondente all'agente dal centro di tuple dell'incrocio. Ovviamente, nel caso in cui l'agente non stia nè entrando nè uscendo dall'incrocio, nessuna tupla verrà inserita nel centro di tuple.

Listing 4.24: Driver - !checkInsideCrossroad

```
+!checkInsideCrossroad(N, RoadId) : true <-
  crossroad.internalaction.
        checkInsideCrossroad(N, RoadId, Dir);
  !insertTupleCrossroad(N, RoadId, Dir).

+!insertTupleCrossroad(N, RoadId, enter) : true <-
  out(crossroadTC,driver(N,RoadId)).
+!insertTupleCrossroad(N, RoadId, exit) : true <-
  in(crossroadTC,driver(N,RoadId)).
+!insertTupleCrossroad(_,_,_).
```

Un altro controllo che l'agente deve effettuare consiste nel verificare se egli si trova in prossimità di un punto nel quale potrebbe esservi un segnale stradale e, in tal caso, deve agire adeguatamente andando a leggere dal centro di tuple della strada su cui si trova se vi sono delle tuple che corrispondono ai segnali stradali; questa operazione viene svolta tramite il piano *!checkNearRoadSignal*. Per semplicità si è deciso che, nel caso in cui non fosse presente nessun segnale, la primitiva `rdp` avrebbe dovuto restituire la stringa *no*, altrimenti, a seconda del tipo l'agente deve agire conseguentemente.

Nel caso vi sia un segnale di *stop*, l'agente deve inserire una tupla nella forma *stopped(NomeAgente)* in modo che gli agenti che non hanno un segnale di stop ma solo di precedenza non tengano conto della sua presenza. Successivamente attende fermo per 2 secondi e controlla che non vi sia nessuno tramite la primitiva `noOthers`, in tal caso rimuove la relativa tupla *stopped* dal centro di tuple e il piano termina.

Listing 4.25: Driver - !checkNearRoadSignal

```
+!checkNearRoadSignal(N, RoadId) : true <-
  crossroad.internalaction.
      isNearRoadSignalPoint(N,RoadId,Bool);
  !checkRoadSignal(RoadId,Bool).

+!checkRoadSignal(RoadId,false).
+!checkRoadSignal(RoadId,true) : true <-
  rdp(RoadId, signal(S));
  !doSignal(S).

+!doSignal(stop) : true <-
  .my_name(N);
  out(crossroadTC, stopped(N));
  .wait(2000);
  noOthers(crossroadTC);
  in(crossroadTC, stopped(N)).
+!doSignal(prec) : true <-
  noOthers(crossroadTC).
+!doSignal(no).
+!doSignal(_).
```

Infine, dopo aver utilizzato l'internal action *isRoadEnded* per verificare se si trova, oppure no, al termine di una strada, l'agente mette in atto il piano denominato *!checkRoadEnded* che, richiamando i metodi opportuni, gestisce la

visibilità e il movimento dell'agente così da dare l'impressione all'utente che l'agente esca dalla mappa e che un altro agente rientri dalla strada adiacente.

Listing 4.26: Driver - !checkRoadEnded

```
+!checkRoadEnded(false,_).
+!checkRoadEnded(true, road1) : true <-
  goOut;
  stay;
  .wait(1000);
  goIn;
  moveLeft.
+!checkRoadEnded(true, road2) : true <-
  goOut;
  stay;
  .wait(1000);
  goIn;
  moveRight.
+!checkRoadEnded(true, road3) : true <-
  goOut;
  stay;
  .wait(1000);
  goIn;
  moveUp.
+!checkRoadEnded(true, road4) : true <-
  goOut;
  stay;
  .wait(1000);
  goIn;
  moveDown.
```

4.2.3 Risultati dell'esperimento

In questo primo esperimento gli agenti sono chiamati ad agire in modo appropriato in relazione al tipo di segnale stradale che incontrano sulla loro strada, a patto che ve ne sia uno.

Osservando per un periodo sufficientemente prolungato la simulazione in modo che all'incrocio si presentino varie casistiche (per esempio sono presenti tutte le macchine, oppure solo due, etc.), gli agenti, che si trovano a percorrere la strada che presenta il segnale di precedenza, si fermano solo nel caso in cui provengano delle macchine dalle strade che risultano perpendicolari a quella in cui essi si trovano, in quanto la strada rimanente presenta il segnale di *stop*, perciò gli agenti provenienti da quella strada non sono tenuti in considerazione. Non appena gli agenti che si muovono orizzontalmente hanno superato l'incrocio, l'agente torna a percorrere la propria strada.

Al contrario, gli agenti che percorrono la strada e si trovano dinnanzi al segnale di *stop*, attendono che tutti gli agenti che provengono dalle altre tre strade abbiano superato l'incrocio per poi ripartire e proseguire sulla propria strada.

In questo primo esperimento, quindi, si può osservare come gli agenti, "vedendo" vari tipi di segnali stradali, agiscano in maniera diversa ma sempre appropriata in accordo con i significati che il codice della strada impone.

4.3 Esperimento 2 - Office Casestudy

4.3.1 Descrizione

Nell'esperimento denominato *Office Casestudy* viene mostrato un ambiente lavorativo nel quale sono stati inseriti quattro agenti: un capo ufficio (*chief*) e tre impiegati (*clerk*). Il *chief* comunica ai propri dipendenti i compiti che devono essere svolti attraverso l'utilizzo di post-it colorati che vengono posti nell'ufficio dei task. Ogni colore corrisponde ad un diverso compito e spetta ai dipendenti interpretarne il significato e mettere in atto l'azione appropriata (Figura 4.5).

All'interno dell'ufficio si trovano diversi elementi:

Task Office: in alto a sinistra si trova l'ufficio dei task nel quale il *chief* andrà a posizionare i propri post-it e dal quale i *clerk* andranno a prelevare gli stessi.

Chief Office: in alto a destra si trova l'ufficio del capo nel quale egli si trova ad inizio simulazione e dove ritorna dopo aver inserito un nuovo task all'interno dell'ufficio dei task.

Printer: in colore verde viene rappresentata la stampante; perciò a post-it verde corrisponde un utilizzo da parte dei *clerk* della stampante.

Phone: in colore rosa viene rappresentato il telefono; perciò a post-it rosa corrisponde un utilizzo da parte dei *clerk* del telefono.

Desk: in colore azzurro vengono rappresentate le scrivanie dei *clerk*, ogni impiegato ha la propria scrivania; perciò a post-it azzurro corrisponde un utilizzo da parte dei *clerk* della propria scrivania, inoltre nel caso in cui gli impiegati non abbiano compiti da svolgere, essi tornano alla propria scrivania per sistemare le scartoffie.

4.3.2 Implementazione

Analogamente al primo esempio, l'implementazione di questo esperimento si articola in una parte basata su classi Java ed una parte basata su Jason.

Per quanto riguarda la parte basata sul linguaggio Java, le classi sono state organizzate in quattro package: il package denominato *office* nel quale si trovano le classi che si riferiscono all'ambiente utilizzato, nel package *office.agent*, invece, si trovano le classi che si riferiscono agli agenti, nel package *office.internalaction* si trovano le classi che definiscono le internal action utilizzate dagli agenti Jason ed, infine, nel package *office.pathfinder* si trovano le classi utilizzate per la ricerca del percorso che deve compiere l'agente per andare da un punto all'altro.

Package office

La classe che rappresenta il modello della scena, chiamata *OfficeModel*, estende la classe *GridWorldModel* che, come già detto nella Sottosezione 4.2.2, permette di creare una griglia delle dimensioni desiderate e di definire gli ostacoli e gli agenti necessari per creare l'ambiente dell'esperimento che si vuole mostrare.

Per assicurarsi che una sola istanza di tale classe venga creata, si è deciso ancora una volta di utilizzare il pattern *Singleton* e per questo motivo la

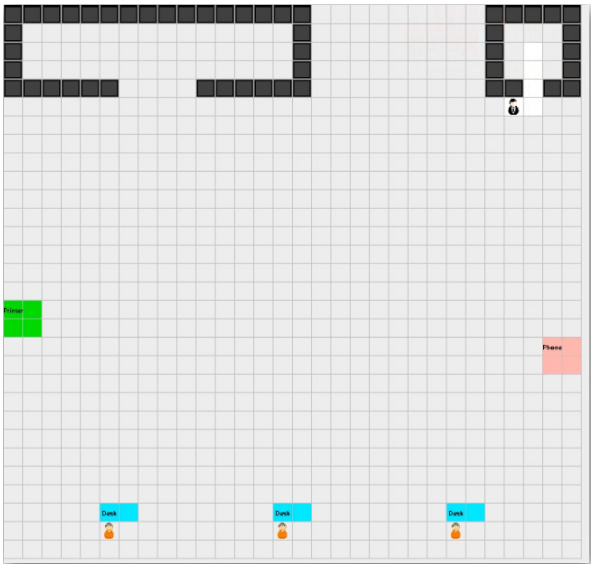

Figura 4.5: Esperimento office all'avvio

classe *OfficeModel* ha il costruttore privato e un metodo *getter* statico che crea un'istanza della classe la prima volta che viene richiamato e, per le successive invocazioni di questo metodo, continua a restituire sempre la stessa istanza.

Analogamente a quanto già visto per la classe *CrossroadModel*, anche questa rende disponibili dei metodi utili per il movimento degli agenti.

Il metodo *move* viene utilizzato per far muovere l'agente, a seconda della direzione scelta, egli si muoverà di un punto sulla griglia, sempre che questo non sia occupato da un ostacolo.

Listing 4.27: OfficeModel - move

```
boolean move(Move dir, int ag) throws Exception {
  Location l = getAgPos(ag);
    switch (dir) {
    case UP:
        if (isPositionFree(l.x, l.y - 1)) {
          setAgPos(ag, l.x, l.y - 1);
        }
        break;
    case DOWN:
        if (isPositionFree(l.x, l.y + 1)) {
            setAgPos(ag, l.x, l.y + 1);
        }
        break;
    case RIGHT:
        if (isPositionFree(l.x + 1, l.y)) {
            setAgPos(ag, l.x + 1, l.y);
        }
        break;
    case LEFT:
        if (isPositionFree(l.x - 1, l.y)) {
```

```
            setAgPos(ag, l.x - 1, l.y);
        }
        break;
    case STAY:
      setAgPos(ag, l.x, l.y);
      break;
    }
    return true;
}
```

Come si può notare, nel metodo appena mostrato è stato utilizzato *isPositionFree* che non è un metodo definito nel *GridWorldModel* come il metodo *isFree* utilizzato nell'esperimento precedente, in quanto in questo caso si ha l'esigenza di definire dei nuovi tipi di ostacolo come la stampante, il telefono e la scrivania (e relative stringhe) e non si vuole rendere come ostacoli gli altri agenti, cioè si vuole permettere ad essi di sostare su posizioni della mappa già occupate da altri agenti.

Listing 4.28: OfficeModel - isPositionFree

```
private boolean isPositionFree(int x, int y) {
  return  isFree(OfficeModel.OBSTACLE, x, y)     &&
      isFree(OfficeModel.DESK, x, y)       &&
      isFree(OfficeModel.PHONE, x, y)      &&
      isFree(OfficeModel.PRINTER, x, y)    &&
      isFree(OfficeModel.DESKSTRING, x, y)   &&
      isFree(OfficeModel.PHONESTRING, x, y) &&
      isFree(OfficeModel.PRINTERSTRING, x, y);
}
```

Il *chief* ha a disposizione un ufficio dei task all'interno del quale può depositare dei post-it corrispondenti al compito che i *clerk* devono svolgere. In realtà, all'interno dell'ufficio dei task, si è deciso di limitare a 14 lo spazio disponibile per apporre dei post-it; lo stato di queste posizioni è mantenuto all'interno di una *HashMap*, denominata *taskPlaceFree*, che ha come chiave un numero che va da 0 a 13 e come valore il booleano che indica se tale posto è libero oppure no. Di conseguenza è necessario un metodo che sia in grado di restituire una posizione nella quale poter apporre un post-it. Il metodo in questione, denominato *getFirstTaskOfficePlaceFree* non fa altro che restituire la prima posizione libera in cui poter inserire il nuovo post-it, contestualmente a ciò modifica il valore corrispondente nella *HashMap* ponendolo a *false*.

Listing 4.29: OfficeModel - getFirstTaskOfficePlaceFree

```
public Location getFirstTaskOfficePlaceFree(){
  Set<Entry<Integer, Boolean>> temp =
            taskPlaceFree.entrySet();
  for (Entry<Integer, Boolean> entry : temp) {
    if(entry.getValue()){
      taskPlaceFree.put(entry.getKey(), false);
      return taskPlace[entry.getKey()];
    }
  }
  return null;
}
```

Quando il *chief* ha queste informazioni può procedere con l'inserimento di un nuovo task tramite il metodo *addTask*. Per evitare che vi siano problemi di concorrenza si è deciso di creare un semaforo (*taskListLock*) con un solo permesso, perciò un qualsiasi agente che voglia aggiungere o rimuovere un task deve prima di tutto acquisire tale permesso sul semaforo.

I task presentano un identificativo univoco in quanto per far sì che i *clerk* rimuovano il giusto task devono essere a conoscenza della posizione in cui si trova, quindi per semplificare il tutto si è deciso di aggiungere i task ad una *HashMap* ogni qualvolta ne venga aggiunto uno nuovo nell'ufficio dei task. L'*HashMap*, denominata *taskLocation*, ha come chiave l'identificativo univoco del task e come valore la posizione in cui si trova.

Listing 4.30: OfficeModel - addTask

```
public void addTask(int id,int taskType,Location taskPlace){
  try {
    this.taskListLock.acquire();
    this.add(taskType, taskPlace);
    taskLocation.put(id, taskPlace);
    this.taskListLock.release();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

Analogamente al metodo utilizzato per aggiungere nuovi task, si ha anche quello impiegato per rimuoverli che, dato l'identificativo del task da eliminare e ottenuto il lock sul semaforo, recupera la posizione del task, rimuove il task corrispondente ed infine pone nuovamente a *true* il valore nell'*HashMap taskPlaceFree* in modo che il *chief* possa riutilizzare quel posto per inserire un nuovo post-it.

Listing 4.31: OfficeModel - removeTask

```
public void removeTask(int id, int taskType) {
  try {
    this.taskListLock.acquire();
    Location pos = taskLocation.get(id);
    this.remove(taskType, pos);

    for (int i = 0; i < taskPlace.length; i++) {
      if(taskPlace[i].x ==
          pos.x && taskPlace[i].y == pos.y) {
        taskPlaceFree.put(i, true);
      }
    }
    this.taskListLock.release();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

Poiché risulta necessario definire dei punti bloccati sulla mappa, è stato utile creare un metodo, denominato *initPathFinder*, tramite il quale viene inizializzata una matrice di interi (*isTileBlocked*) avente la stessa dimensione del modello

dell'ambiente. Più nello specifico, in corrispondenza delle coordinate degli strumenti da ufficio (e relative stringhe) e dei muri delineanti le stanze, nella matrice viene inserito il valore 2, rappresentante il fatto che quella specifica posizione è bloccata. In questo modo si ottiene una rappresentazione delle coordinate bloccate, mappata esattamente sul modello a griglia utilizzato, dalla quale si parte per inizializzare il motore di *pathfinding* per la ricerca del percorso minimo.

Listing 4.32: OfficeModel - initPathFinder

```
public void initPathFinder() {
  this.isTileBlocked = new int[data.length][data.length];
  for (int j = 0; j < data.length; j++) {
    for (int i = 0; i < data.length; i++) {
      if( data[i][j] == OfficeModel.OBSTACLE     ||
        data[i][j] == OfficeModel.DESK       ||
        data[i][j] == OfficeModel.PHONE      ||
        data[i][j] == OfficeModel.PRINTER    ||
        data[i][j] == OfficeModel.DESKSTRING     ||
        data[i][j] == OfficeModel.PHONESTRING ||
        data[i][j] == OfficeModel.PRINTERSTRING ) {

        this.isTileBlocked[i][j] = 2;
      }
    }
  }
  this.pfm = new PathFindingMap(this.isTileBlocked);
}
```

Infine, il metodo *world* è stato utilizzato per definire gli elementi che compongono l'ambiente, come gli ostacoli, gli agenti, la stampante, il telefono, le scrivanie e relative stringhe, definendone le posizioni che devono avere. Inoltre si occupa di inizializzare l'*HashMap* utilizzata per tener traccia di quali sono le posizioni libere nelle quali poter inserire nuovi post-it.

Listing 4.33: OfficeModel - world

```
static OfficeModel world() throws Exception {
  OfficeModel model = OfficeModel.create(30, 30, 4);

  model.add(OfficeModel.OBSTACLE, 0, 0);
  ...
  model.add(OfficeModel.OBSTACLE, 29, 1);

  //Printer
  model.add(OfficeModel.PRINTER, 0, 17);
  model.add(OfficeModel.PRINTER, 1, 16);
  model.add(OfficeModel.PRINTER, 1, 17);
  model.add(OfficeModel.PRINTERSTRING, 0, 16);

  //Phone
  model.add(OfficeModel.PHONE, 28, 19);
  model.add(OfficeModel.PHONE, 29, 18);
  model.add(OfficeModel.PHONE, 29, 19);
  model.add(OfficeModel.PHONESTRING, 28, 18);
```

```
  //Desks
  model.add(OfficeModel.DESK, 6, 27);
  model.add(OfficeModel.DESKSTRING, 5, 27);
  model.add(OfficeModel.DESK, 15, 27);
  model.add(OfficeModel.DESKSTRING, 14, 27);
  model.add(OfficeModel.DESK, 24, 27);
  model.add(OfficeModel.DESKSTRING, 23, 27);

  //Agents
  model.setAgPos(0, 27, 2);
  model.setAgPos(1, 5, 28);
  model.setAgPos(2, 14, 28);
  model.setAgPos(3, 23, 28);

  taskPlace = new Location[14];
  for (int i = 0; i < 14; i++) {
    int x = 1 + i;
    int y = 1;
    taskPlaceFree.put(i, true);
    taskPlace[i] = new Location(x, y);
  }
  return model;
}
```

La seconda classe che si trova nel package *office* è *OfficeView* che si occupa di inizializzare i componenti grafici e di mettere a disposizione i metodi che permettono di disegnare i vari elementi presenti nella scena come gli agenti e gli strumenti di lavoro. In particolare, si possono distinguere otto diversi metodi relativi ad elementi differenti.

Il metodo *draw*, che va a sovrascrivere il metodo della classe parent di *OfficeView* (cioè *GridWorldView*), viene utilizzato per discriminare se l'oggetto che si desidera disegnare è una scrivania, il telefono, la stampante oppure le stringhe relative e, a seconda del caso, viene richiamato il metodo adeguato.

Listing 4.34: OfficeView - draw

```
@Override
public void draw(Graphics g, int x, int y, int object) {
  switch (object) {
  case OfficeModel.PRINTER:
       drawPrinter(g, x, y); break;
  case OfficeModel.PRINTERSTRING:
       drawPrinterString(g, x, y); break;
  case OfficeModel.PHONE:
       drawPhone(g, x, y); break;
  case OfficeModel.PHONESTRING:
       drawPhoneString(g, x, y); break;
  case OfficeModel.DESK:
       drawDesk(g, x, y); break;
  case OfficeModel.DESKSTRING:
       drawDeskString(g, x, y); break;
  }
}
```

Anche il metodo *drawAgent* sovrascrive il metodo già definito nella classe *GridWorldView* in quanto si desiderano visualizzare gli agenti con due immagini che rappresentino il *chief* (Figura 4.6) e i *clerk* (Figura 4.7). Poiché il *chief* ha l'identificativo pari a 0, questa informazione discrimina a quale agente assegnare l'immagine del *chief*.

Figura 4.6: Immagine del chief

Figura 4.7: Immagine dei clerk

Listing 4.35: OfficeView - drawAgent

```
@Override
public void drawAgent(  Graphics g, int x, int y,
            Color c, int id) {
  File img;
    if( id==0){
      img = new File("resources/images/chief.png");
    }else{
      img = new File("resources/images/clerk.png");
    }
      BufferedImage bimg = null;
      try {
        bimg = ImageIO.read(img);
      } catch (IOException e) {
        e.printStackTrace();
      }
      g.drawImage(bimg, x * cellSizeW + 2,
      y * cellSizeH + 2, cellSizeW - 4, cellSizeW - 4, null);
}
```

Infine, per disegnare gli strumenti di lavoro e le relative stringhe sono stati utilizzati dei metodi molto semplici che si occupano di assegnare un colore e di disegnare un rettangolo oppure una stringa a seconda del caso.

Listing 4.36: OfficeView - Metodi draw

```
public void drawDeskString(Graphics g, int x, int y) {
  Color idColor = Color.CYAN;
  g.setColor(idColor);
  g.fillRect(x * cellSizeW + 1, y * cellSizeH+1,
        cellSizeW-1, cellSizeH-1);
  g.setColor(Color.BLACK);
  drawString(g, x, y, myFont, "Desk");
}
```

```
public void drawDesk(Graphics g, int x, int y) {
  Color idColor = Color.CYAN;
  g.setColor(idColor);
  g.fillRect(x * cellSizeW + 1, y * cellSizeH+1,
        cellSizeW-1, cellSizeH-1);
}

public void drawPhoneString(Graphics g, int x, int y) {
  Color idColor = Color.PINK;
  g.setColor(idColor);
  g.fillRect(x * cellSizeW + 1, y * cellSizeH+1,
        cellSizeW-1, cellSizeH-1);
  g.setColor(Color.BLACK);
  drawString(g, x, y, myFont, "Phone");
}

public void drawPhone(Graphics g, int x, int y) {
  Color idColor = Color.PINK;
  g.setColor(idColor);
  g.fillRect(x * cellSizeW + 1, y * cellSizeH+1,
        cellSizeW-1, cellSizeH-1);
}

public void drawPrinterString(Graphics g, int x, int y) {
  Color idColor = Color.green;
  g.setColor(idColor);
  g.fillRect(x * cellSizeW + 1, y * cellSizeH+1,
        cellSizeW-1, cellSizeH-1);
  g.setColor(Color.BLACK);
  drawString(g, x, y, myFont, "Printer");
}

public void drawPrinter(Graphics g, int x, int y) {
  Color idColor = Color.green;
  g.setColor(idColor);
  g.fillRect(x * cellSizeW + 1, y * cellSizeH+1,
        cellSizeW-1, cellSizeH-1);
}
```

Infine, nel package *office* si trova la classe *OfficeEnv* che estende la classe *Environment* e, oltre ad inizializzare Model e View, si occupa anche di definire alcune azioni tramite il metodo *executeAction*. In particolare vengono definite le azioni associate al movimento classico verso un punto, quelle che impostano il percorso verso gli strumenti di lavoro e le azioni che definiscono la procedura effettuata per inserire o rimuovere un task.

Listing 4.37: OfficeEnv - executeAction

```
@Override
public boolean executeAction(String agName, Structure act) {
  boolean result = false;

  if(this.agentsFlags.get(agName) == null){
```

```
    this.agentsFlags.put
        (agName, new HashMap<String, Integer>());
  }
  HashMap<String, Integer> af =
              this.agentsFlags.get(agName);
  if(this.agentsPaths.get(agName) == null){
    this.agentsPaths.put
          (agName, new HashMap<String, Path>());
  }
  HashMap<String, Path> ap = this.agentsPaths.get(agName);

    try {
      int agId = getAgIdBasedOnName(agName);

        if(act.getFunctor().equals("goToPrinter")){
          Location agL = this.model.getAgPos(agId);
        Path path =
          this.model.findPathTo(agL.x, agL.y, 2, 16);
        ap.put("pathToDestination", path);
        result = true;
      }else if(act.getFunctor().equals("goToPhone")){
            Location agL = this.model.getAgPos(agId);
        Path path =
          this.model.findPathTo(agL.x, agL.y, 27, 18);
        ap.put("pathToDestination", path);
        result = true;
      }else if(act.getFunctor().equals("goToTaskOffice")){
        Location agL = this.model.getAgPos(agId);
        Path path =
          this.model.findPathTo(agL.x, agL.y, 7, 4);
        ap.put("pathToDestination", path);
        result = true;
      }else if(act.getFunctor().equals("removeTask")){
        int id =
          Integer.parseInt(act.getTerm(0).toString());
        String c = act.getTerm(1).toString();
        if(c.equals("green")){
          this.model.removeTask(id,OfficeModel.PRINTER);
      } else if(c.equals("pink")){
          this.model.removeTask(id,OfficeModel.PHONE);
        } else if(c.equals("cyan")){
          this.model.removeTask(id,OfficeModel.DESK);
        }
        result = true;
      }else if(act.getFunctor().equals("insertNewTask")){
        int id =
          Integer.parseInt(act.getTerm(0).toString());
        String c = act.getTerm(1).toString();
        Location taskPlace =
          this.model.getFirstTaskOfficePlaceFree();
        if (taskPlace!=null){
          if(c.equals("green")){
              this.model.addTask
                (id, OfficeModel.PRINTER, taskPlace);
```

```
            } else if(c.equals("pink")){
              this.model.addTask
                (id, OfficeModel.PHONE, taskPlace);
            } else if(c.equals("cyan")){
              this.model.addTask
                (id, OfficeModel.DESK, taskPlace);
            }
          }else{
            af.put("fullWall", 1);
          }
          result = true;

        }else if(act.getFunctor().equals("moveTo")){
          int x = Integer.parseInt(act.getTerm(0).toString());
          int y = Integer.parseInt(act.getTerm(1).toString());
              Location agL = this.model.getAgPos(agId);
          Path path =
            this.model.findPathTo(agL.x, agL.y, x, y);
          ap.put("pathToDestination", path);
          result = true;

        } else if (act.getFunctor().equals("move")) {

          Location agL = this.model.getAgPos(agId);
          Path path = ap.get("pathToDestination");
          if(path != null) {
            if(!path.isEmpty()) {
              if( path.getStep(0).getX() == -1 &&
                path.getStep(0).getY() == -1) {
                af.put("arrived", 1);
                result = true;
              } else {
                Step coord = path.getStep(0);
                int x = coord.getX();
                int y = coord.getY();
                if(this.model.isTileBlocked(x,y)) {
                  Step destination = path.getLast();
                  if(!this.model.isTileBlocked
                      (destination.getX(),
                      destination.getY())) {
                    path =
                      this.model.findPathTo
                      (agL.x, agL.y,
                      destination.getX(), destination.getY());
                    coord = path.removeStep(0);
                    x = coord.getX();
                    y = coord.getY();
                    ap.put("pathToDestination", path);
                    result = this.model.move(agId, x, y);
                  } else {
                    af.put("arrived", 1);
                    result = true;
                  }
                } else {
```

```
                path.removeStep(0);
                ap.put("pathToDestination", path);
                result = this.model.move(agId, x, y);
              }
            }
          } else {
            af.put("arrived", 1);
            result = true;
          }
        } else {
          af.put("noway", 1);
          result = true;
        }
      }

       } catch (InterruptedException e) {
       } catch (Exception e) {
            logger.log(Level.SEVERE, "error executing "
              + act + " for " + agName, e);
       }

       if (result) {
          this.agentsFlags.put(agName, af);
          this.agentsPaths.put(agName, ap);

      updateAgPercepts(agName, act.getFunctor(), af);

      return true;
    }

       return result;
}
```

Infine il metodo *updateAgPercepts* viene richiamato per aggiornare tutte le percezioni dell'agente che possono essere inserite nella sua base di conoscenza. Tali percezioni sono:

arrived viene utilizzata quando l'agente raggiunge la destinazione verso la quale si sta muovendo.

fullWall viene utilizzata quando il muro dei task è pieno e quindi non vi sono spazi per ulteriori post-it.

noway viene utilizzata per indicare che non vi è un percorso per giungere alla destinazione desiderata.

Listing 4.38: OfficeEnv - updateAgPercepts

```
private void updateAgPercepts(String agName,
        String action, HashMap<String,Integer> af) {
  this.agsRepo.clearPercepts(agName);

  if(af.get("arrived") != null && af.get("arrived") == 1) {
    this.agsRepo.addPercept
          (agName, Literal.parseLiteral("arrived"));
    af.put("arrived", 0);
    this.agentsFlags.put(agName, af);
  }

  if( af.get("fullWall") != null &&
    af.get("fullWall") == 1) {
    this.agsRepo.addPercept
          (agName, Literal.parseLiteral("fullWall"));
    af.put("fullWall", 0);
    this.agentsFlags.put(agName, af);
  }

  if(af.get("noway") != null && af.get("noway") == 1) {
    this.agsRepo .addPercept
            (agName, Literal.parseLiteral("noway"));
    af.put("noway", 0);
    this.agentsFlags.put(agName, af);
  }
}
```

Package office.agent

All'interno di questo package si trovano le classi *AbstractTucsonJasonAgent* e *TucsonJasonAgent* di cui si è discusso nella Sezione 3.1. In questo package si trova anche la classe *AgentsRepository* di cui si è discusso nella Sezione 3.2.

Nella classe *TucsonJasonAgent* è stato definito un solo centro di tuple, denominato *blackboardTC*. Ogni volta che il *chief* depone un post-it, viene inserita una nuova tupla del tipo *task(Id, Colore)* nel centro di tuple.

Listing 4.39: TucsonJasonAgent - Centro di tuple

```
blackboardTC =
  new TucsonTupleCentreId("blackboardTC",this.ip,this.port);
```

Inoltre, in questa classe, sono state implementate solo alcune delle primitive di coordinazione discusse nella Sezione 3.1: `out`, `in`, `rdp` e `inp`.

Package office.internalaction

Nel package denominato *office.internalaction* sono state inserite le classi che definiscono delle azioni interne che vengono utilizzate dall'agente Jason.

La prima internal action, denominata *chooseTask*, viene utilizzata dall'agente Jason *chief* per decidere che tipo di task inserire nell'ufficio dei task. Tale operazione viene svolta tramite la definizione di un numero casuale e, a seconda del quale, viene scelto un colore diverso che rappresenta un task specifico.

Listing 4.40: Internal action - chooseTask

```
public class chooseTask extends DefaultInternalAction {
  @Override
  public Object execute(TransitionSystem ts,
          Unifier un, Term[] terms) throws Exception {
    try {

      switch (r.nextInt(3)) {
        case 0:
          return un.unifies(terms[0], new Atom("green"));

        case 1:
          return un.unifies(terms[0], new Atom("pink"));

        case 2:
          return un.unifies(terms[0], new Atom("cyan"));
      }

      return un.unifies(terms[0], new Atom("cyan"));
    } catch (Throwable e) {
      e.printStackTrace();
      return false;
    }
  }

  private int getAgIdBasedOnName(String agName) {
    if(agName.equals("chief")) {
      return 0;
    } else {
      return (Integer.parseInt(agName.substring(5)));
    }
  }
}
```

La seconda ed ultima internal action, denominata *getMyPosition*, viene utilizzata dall'agente Jason per conoscere la propria posizione attuale.

Listing 4.41: Internal action - checkInsideCrossroad

```
public class getMyPosition extends DefaultInternalAction {
  @Override
  public Object execute(TransitionSystem ts,
          Unifier un, Term[] terms) throws Exception {
    try {

      OfficeModel model = OfficeModel.get();

      int agId =
        this.getAgIdBasedOnName(terms[0].toString());

      Location l = model.getAgPos(agId);
      int x = l.x;
      int y = l.y;

      return
```

```
        un.unifies(terms[1],
                   new Structure(Literal
                      .parseLiteral("pos("+x+","+y+")")));
    } catch (Throwable e) {
      e.printStackTrace();
      return false;
    }
  }

  private int getAgIdBasedOnName(String agName) {
    if(agName.equals("chief")) {
      return 0;
    } else {
      return (Integer.parseInt(agName.substring(5)));
    }
  }
}
```

Package office.pathfinder

Nel package *office.pathfinder* sono state inserite le classi utilizzate per determinare il percorso che l'agente deve effettuare per andare da un punto ad un altro.

La mappa del mondo nella quale tutti gli agenti "vivono" è una griglia di forma quadrata che comprende anche la presenza di ostacoli come i muri e gli strumenti di lavoro utilizzati dagli agenti. Per garantire che un qualsiasi agente riesca a muoversi agevolmente in una mappa di questo tipo, è stato implementato un algoritmo per il calcolo dei percorsi che tiene conto di quali zone della mappa sono bloccate oppure accessibili.

Per l'implementazione dell'algoritmo è risultato necessario generare una nuova mappa, associata al modello a griglia utilizzato, denominata *PathFindingMap* ed avente come unici attributi le dimensioni in larghezza e altezza e una matrice di valori interi contenente:

- *0*: se la posizione i,j non è bloccata
- *2*: se la posizione i,j è bloccata

Partendo da questa nuova mappa si può procedere con la valutazione del percorso più breve tramite l'utilizzo del ben noto *algoritmo A**.

Prima di tutto si controlla che il nodo target non sia bloccato poiché, nel caso lo fosse, l'algoritmo terminerà immediatamente non restituendo alcun percorso.

Listing 4.42: AStarPathFinder - findPath

```
sourceX = tx;
sourceY = ty;
distance = 0;

if (map.blocked(this, tx, ty)) {
  return null;
}
```

Se il nodo target non è bloccato, l'algoritmo comincia partendo dal nodo selezionato. Per ogni nodo è definito un costo di entrata (zero per il nodo iniziale). A* allora valuta la distanza dal nodo target a partire da quello corrente. Questa stima ed il costo formano l'euristica che sarà assegnata al percorso passante per questo nodo che viene aggiunto ad una lista chiamata *open*.

Listing 4.43: AStarPathFinder - findPath

```
nodes[sx][sy].cost = 0;
nodes[sx][sy].depth = 0;
closed.clear();
open.clear();
addToOpen(nodes[sx][sy]);
```

L'algoritmo rimuove il primo nodo dalla lista (perché avrà valore della funzione euristica più basso). Se la lista è vuota, non ci saranno percorsi dal nodo iniziale a quello target e l'algoritmo si arresterà.

Se il nodo corrisponde a quello target, A* ricostruisce e pone in output il percorso ottenuto e si arresta. Questa ricostruzione del percorso a partire dai nodi più vicini significa che non è necessario memorizzare il percorso in ogni nodo. Se il vicino che si sta valutando coincide con il nodo target e se tale vicino è una posizione valida [1] allora l'algoritmo termina poiché il percorso è stato trovato.

Listing 4.44: AStarPathFinder - findPath

```
if (current == nodes[tx][ty]) {
  if (isValidLocation(sx,sy,tx,ty)) {
    break;
  }
}
```

L'algoritmo gestisce anche una lista di *closed*, un elenco di nodi che sono già stati controllati. Quindi a partire dal primo nodo, questo viene rimosso dalla lista *open* e inserito nella lista *closed*.

Se il nodo non è quello target, verranno controllati tutti i vicini ammissibili. Per ciascuno di essi, A* calcola il *costo* di entrata nel nodo e lo salva con esso. Questo costo viene calcolato dalla somma cumulativa dei pesi immagazzinati negli antenati, più il costo dell'operazione per raggiungere il nodo correntemente analizzato.

Listing 4.45: AStarPathFinder - findPath

```
for (int x=-1;x<2;x++) {
    for (int y=-1;y<2;y++) {
        if ((x == 0) && (y == 0)) {
            continue;
        }
        int xp = x + current.x;
        int yp = y + current.y;

        if (isValidLocation(current.x,current.y,xp,yp)) {
```

[1] *isValidLocation* è un metodo che ritorna *true* a patto che gli argomenti in ingresso *tx* e *ty* (nodo target) non siano esterni alla mappa e che non corrispondano rispettivamente a *sx* e *sy* (posizione corrente)

```
            float nextStepCost = current.cost +
                getMovementCost(current.x, current.y,
                                        xp, yp);
```

Se il nuovo costo determinato per tale nodo è inferiore a quello precedentemente calcolato, il nodo non deve essere scartato. Quindi, potendovi essere un percorso a costo minore per raggiungerlo, esso deve essere valutato nuovamente.

Successivamente, viene aggiunta al costo una stima della distanza dal nodo corrente a quello target per formare il suo valore euristico. Tale nodo viene aggiunto, quindi, alla lista *open*, a meno che non esista un nodo identico con valore euristico minore o uguale.

Listing 4.46: AStarPathFinder - findPath

```
if (!inOpenList(neighbour) && !(inClosedList(neighbour))) {
    neighbour.cost = nextStepCost;
    neighbour.heuristic = getHeuristicCost(xp, yp, tx, ty);
    maxDepth = Math.max(maxDepth, neighbour.setParent(current));
  addToOpen(neighbour);
}
```

Il metodo *getHeuristicCost* invoca a sua volta il metodo *getCost*, definito nella classe *ClosestHeuristic*, che rappresenta l'euristica omonima grazie alla quale si trova la soluzione ottima, quindi il percorso minimo

L'algoritmo viene applicato ad ogni nodo vicino, fatto ciò il nodo originale viene inserito nella lista di *closed* e il prossimo nodo verrà recuperato dalla lista di *open* e su di esso si ripeterà il processo appena descritto.

Terminato tale processo, il percorso minimo è stato calcolato e si procede con la sua ricostruzione partendo dal nodo target fino al nodo di partenza e inserendo un nodo alla volta sempre in testa alla lista.

Listing 4.47: AStarPathFinder - findPath

```
Path path = new Path();
Node target = nodes[tx][ty];
while (target != nodes[sx][sy]) {
    path.prependStep(target.x, target.y);
    target = target.parent;
}
path.prependStep(sx,sy);

return path;
```

Il parametro di ritorno di questo algoritmo è un oggetto di tipo *Path* che rappresenta il percorso come lista di passi da effettuare. Tale oggetto, infatti, incapsula un'*ArrayList* di oggetti di tipo *Step* che rappresentano il singolo passo in termini di coordinate cartesiane.

Per sfruttare l'algoritmo appena descritto, nella classe *OfficeModel* è stato definito il metodo *findPathTo*, il quale prese in ingresso le coordinate di partenza ed arrivo, crea un nuovo *AStarPathFinder* invocando su di esso il metodo *findPath*.

Listing 4.48: OfficeModel - findPathTo

```
public synchronized Path
      findPathTo(int fromX, int fromY, int toX, int toY) {
  if(fromX == toX && fromY == toY) {
    Path p = new Path();
    p.appendStep(-1, -1);
    return p;
  }

  AStarPathFinder pathFinder =
    new AStarPathFinder(model.getPathFindingMap(),
                        10000,
                        false);
  Path path = pathFinder.findPath(fromX, fromY, toX, toY);
  return path;
}
```

Nel caso in cui le coordinate di partenza ed arrivo coincidano, viene creato e restituito un path contenente un unico step con valori di coordinate pari a *-1*. Tale percorso sta a significare che l'agente ha già raggiunto la destinazione richiesta.

Agenti Jason

Per quanto riguarda gli agenti Jason, ne sono stati definiti due tipi: *chief* e *clerk*.

L'agente *chief* rappresenta il capo ufficio che ha il compito di inserire dei post-it che rappresentano i task nella lavagna preposta all'interno dell'ufficio dei task. Il *chief* deve, quindi, scegliere il colore del post-it (in maniera casuale). Poiché vi sono tre *clerk*, si è pensato di far inserire al *chief* due post-it alla volta in modo da impegnare il più possibile gli impiegati.

Nella propria base di conoscenza, il *chief*, ha una convinzione *Id(0)* che rappresenta l'identificativo del primo task da inserire nella lavagna dei task.

Appena creato, questo agente applica il piano *!start*, tramite il quale si occupa di recuperare la propria posizione iniziale e, poiché il *chief* viene posizionato ad inizio simulazione all'interno del proprio ufficio, egli inserisce nella propria base di conoscenza una convinzione del tipo *myOffice(posX, posY)*. In seguito l'agente applica il piano *!chiefLoop*.

Listing 4.49: Chief - !start

```
+!start : true <-
  .my_name(N);
  office.internalaction.getMyPosition(N,pos(X,Y));
  +myOffice(X,Y);
  !chiefLoop.
```

Il piano *!chiefLoop* rappresenta l'azione ciclica compiuta dall'agente. Innanzitutto si dirige verso l'ufficio dei task, dopodiché inserisce due nuovi post-it/task e ritorna al proprio ufficio, nel quale attende 3 secondi prima di ricominciare la propria azione ciclica. Queste operazioni vengono svolte tramite dei piani che verranno descritti in seguito.

Listing 4.50: Chief - !chiefLoop

```
+!chiefLoop: true <-
  goToTaskOffice;
  !doMove;
  .print("I'm going to the task office");
  !insertTwoNewTask;
  ?myOffice(X,Y);
  moveTo(X,Y);
  .print("I'm coming back to my office");
  !doMove;
  .wait(3000);
  !chiefLoop.
```

Tramite il piano *!doMove*, l'agente si dirige verso la destinazione impostata tramite l'azione d'ambiente *goToTaskOffice*, definita nella classe *OfficeEnv*, nella quale si imposta anche il percorso (path) che è necessario intraprendere per raggiungere quel punto. Il piano *!doMove* viene richiamato ciclicamente finché l'agente non giunge a destinazione.

Listing 4.51: Chief - !doMove

```
+!doMove:   arrived.

+!doMove :  not arrived <-
      move;
      !doMove.
```

Una volta raggiunto l'ufficio dei task, il *chief* procede con l'inserimento di due task tramite il piano *!insertTwoNewTask* col quale recupera dalla base di conoscenza l'identificativo che deve avere il prossimo task da inserire, sceglie in modo casuale il colore che deve avere il prossimo post-it grazie all'internal action *chooseTask* di cui si è parlato precedentemente e richiama l'azione d'ambiente *insertNewTask* che, se vi è almeno un posto libero nella lavagna per inserire un nuovo post-it, lo inserisce, altrimenti aggiunge alla base di conoscenza dell'agente una percezione chiamata *fullWall*. Se quest'ultima non risulta presente, viene inserita una nuova tupla corrispondente al nuovo task tramite il piano *!insertPostIt*.

Il procedimento appena descritto viene ripetuto un'altra volta in quanto si desidera che il *chief* inserisca due task per volta.

Listing 4.52: Chief - !insertTwoNewTask

```
+!insertTwoNewTask : true <-
  -id(Id);
  office.internalaction.chooseTask(Color1);
  insertNewTask(Id, Color1);
  !insertPostIt(Id, Color1);
  office.internalaction.chooseTask(Color2);
  insertNewTask(Id+1, Color2);
  !insertPostIt(Id+1, Color2);
  +id(Id+2).

+!insertPostIt(Id, Color): fullWall.
```

```
+!insertPostIt(Id, Color): not fullWall <-
  out(blackboardTC, task(Id, Color));
  .print("A new task is now available!").
```

L'agente *clerk* rappresenta il dipendente che periodicamente si dirige verso l'ufficio dei task alla ricerca di un post-it e, se lo trova, a seconda del colore, mette in pratica un comportamento differente:

- A post-it *verde* corrisponde un'azione che prevede l'utilizzo della stampante.
- A post-it *rosa* corrisponde un'azione che prevede l'utilizzo del telefono.
- A post-it *azzurro* corrisponde un'azione che prevede l'utilizzo della propria scrivania.

Nel caso in cui non siano presenti post-it all'interno dell'ufficio dei task, si è deciso di far tornare il *clerk* alla propria postazione e di fargli riordinare le scartoffie. In seguito tornerà all'ufficio per verificare se vi sono nuovi compiti da svolgere.

Appena creato, l'agente *clerk* applica il piano *!start*, tramite il quale si occupa di recuperare la propria posizione iniziale e, poiché anche i *clerk* vengono posizionati ad inizio simulazione alla propria scrivania, egli inserisce nella propria base di conoscenza una convinzione del tipo *myDesk(posX, posY)*. In seguito l'agente applica il piano *!workLoop*.

Listing 4.53: Clerk - !start

```
+!start : true <-
  .my_name(N);
  office.internalaction.getMyPosition(N,pos(X,Y));
  +myDesk(X,Y);
  .wait(5000);
  !workLoop.
```

Il piano *!workLoop* rappresenta l'azione ciclica compiuta dall'agente *clerk*; innanzitutto si dirige verso l'ufficio dei task, dopodiché ricerca un task da svolgere. Prima di ricominciare la propria azione ciclica attende 5 secondi. Queste operazioni vengono svolte tramite dei piani che verranno descritti in seguito.

Listing 4.54: Clerk - !workLoop

```
+!workLoop: true <-
  goToTaskOffice;
  !doMove;
  .print("I'm going to the task office");
  !checkForTask;
  .wait(5000);
  !workLoop.
```

Come per il *chief*, tramite il piano *!doMove* l'agente si dirige verso la destinazione impostata tramite l'azione d'ambiente *goToTaskOffice*, definita nella classe *OfficeEnv*, nella quale si imposta anche il percorso (path) che è necessario intraprendere per raggiungere quel punto. Il piano *!doMove* viene richiamato finché l'agente non giunge a destinazione.

Listing 4.55: Clerk - !doMove

```
+!doMove:    arrived.

+!doMove :   not arrived <-
       move;
       !doMove.
```

Col piano *!checkForTask* si richiama una primitiva non bloccante `inp` che va a cercare nel centro di tuple *blackboardTC* se è presente una tupla del tipo *task(Id, Color)*; tale primitiva, nel caso non vi sia alcuna tupla che faccia match con il template richiesto, fa unificare *Id* col valore numerico *-1* e *Color* con la stringa *none* in modo che sia possibile, tramite il piano *!doJob*, mettere in pratica il comportamento opportuno. Infatti nel caso in cui non vi sia nessuna tupla che faccia match col template specificato, l'agente deve agire tornando alla propria scrivania per riordinare le proprie scartoffie.

Se viene trovata la tupla corrispondente al template, a seconda del colore ritornato, che corrisponde a quello del post-it, l'agente agirà in modi differenti. Ad ogni modo, però, l'agente segue determinati passi: rimuove il task che corrisponde al post-it letto, tramite l'utilizzo dell'azione d'ambiente *removeTask*, definita nella classe *OfficeEnv*, si dirige verso lo strumento che deve utilizzare per compiere il proprio task ed infine, ritorna alla propria scrivania nel caso non vi si trovi già.

Listing 4.56: Clerk - !checkForTask

```
+!checkForTask: true <- inp(blackboardTC, task(Id, Color));
             !doJob(Id,Color).

+!doJob(-1,none):true <-.print("Nothing to do!");
           ?myDesk(X,Y);
           moveTo(X,Y);
           !doMove;
           .print("I'm organizing my paperwork").

+!doJob(Id,green):true <-removeTask(Id, green);
              goToPrinter;
              .wait(1000);
              !doMove;
              .print("I'm using the printer!");
              ?myDesk(X,Y);
              moveTo(X,Y);
              !doMove.

+!doJob(Id,pink):true <-removeTask(Id, pink);
              goToPhone;
              .wait(1000);
              !doMove;
              .print("I'm using the phone!");
              ?myDesk(X,Y);
              moveTo(X,Y);
              !doMove.

+!doJob(Id,cyan):true <-removeTask(Id, cyan);
              ?myDesk(X,Y);
```

```
        moveTo(X,Y);
        !doMove;
        .wait(1000);
        .print("I'm working at my desk!").
```

4.3.3 Risultati dell'esperimento

In questo secondo esperimento gli impiegati (*clerk*) sono chiamati a mettere in atto determinate azioni in relazione al tipo di post-it recuperato dall'ufficio dei task, precedentemente inserito dal capo (*chief*).

Osservando per un periodo sufficientemente prolungato la simulazione, in modo che vengano coperte tutte le casistiche, cioè che gli agenti siano chiamati a svolgere tutti i compiti previsti, si nota che gli agenti reagiscono opportunamente in base al colore del post-it che recuperano dall'ufficio.

In questo caso non vi è una regola universalmente riconosciuta, ma è una convenzione che si è adottata all'interno di quell'ambiente lavorativo. Perciò si può osservare che, se l'agente recupera dall'ufficio dei task un post-it di colore verde, esso si dirigerà immediatamente alla stampante per effettuare una stampa. Il comportamento aspettato si ottiene anche nel caso in cui l'agente recuperi post-it di colore rosa oppure azzurro.

In questo secondo esperimento, quindi, si può osservare come gli agenti, recuperando post-it di colori differenti, agiscano in maniera diversa ma sempre appropriata in accordo con i significati che il capo ufficio ha assegnato ad ogni colore.

Conclusioni e sviluppi futuri

Nei precedenti capitoli è stato intrapreso un "viaggio" che ci ha portati dai concetti teorici sulla stigmergia cognitiva, fino alla realizzazione di due esperimenti che mettessero in luce le peculiarità di questo approccio.

La stigmergia, che dapprima è stata desunta dal comportamento di animali semplici come le formiche, acquista in questi anni l'appellativo di cognitiva per sottolineare il desiderio degli studiosi di applicare questo concetto ad esseri con capacità cognitive e di ragionamento.

Nella vita di tutti i giorni siamo sottoposti inconsapevolmente a processi cognitivi stigmergici, in quanto, come visto negli esperimenti del Capitolo 4, siamo chiamati ad interpretare dei segni inseriti da altri nell'ambiente in cui viviamo e a reagire conseguentemente ad essi. Perciò nel momento in cui un automobilista, percorrendo la strada, incontra un segnale triangolare bianco col profilo rosso sa benissimo come si deve comportare, infatti questo segnale attiva in lui un processo cognitivo che lo porta, in prossimità dell'incrocio, a fermarsi solo nel caso in cui provengano altre macchine. Risulta quindi evidente, da questo semplice esempio, come ognuno di noi svolga attività di stigmergia cognitiva nella vita di ogni giorno.

Questo elaborato si è posto come obiettivo quello di mostrare i concetti chiave della stigmergia cognitiva e come questa sia stata desunta dalla stigmergia classica (Capitolo 1). Successivamente è stato mostrato come poter utilizzare questi concetti in ambito di programmazione, integrando i linguaggi TuCSoN e Jason, permettendo di definire agenti che presentano delle peculiarità cognitive ed in grado di interagire e coordinarsi tra loro tramite l'infrastruttura TuCSoN (Capitolo 2 e Capitolo 3).
Infine, nel Capitolo 4 sono stati mostrati due esperimenti per meglio comprendere come il concetto di stigmergia cognitiva venga utilizzato nella vita di ogni giorno.

In futuro si potrebbero utilizzare questi concetti per realizzare robot o droni che siano in grado di vedere il mondo che li circonda, individuare eventuali segni che noi umani inseriamo nell'ambiente, interpretarli opportunamente ed agire di conseguenza.

Bibliografia

[1] R. H. Bordini, J. F. Hübner, and M. J. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, Ltd, Oct. 2007. Hardcover.

[2] M. Casadei and M. Viroli. Applying self-organizing coordination to emergent tuple organization in distributed networks. In S. Brueckner, P. Roberson, and U. Bellur, editors, *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'08)*, pages 213–222, Venice, Italy, 20–24 Oct. 2008. IEEE Computer Society.

[3] C. Castelfranchi, G. Pezzullo, and L. Tummolini. Behavioral implicit communication (BIC): Communicating with smart environments via our practical behavior and its traces. *International Journal of Ambient Computing and Intelligence*, 2(1):1–12, Jan.–Mar. 2010.

[4] P. P. Grasse. La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. *La theorie de la stigmergie: Essai d'interpretation du comportament des termites constructeurs*, (6):41–81, 1959.

[5] S. Mariani and A. Omicini. MoK: Stigmergy meets chemistry to exploit social actions for coordination purposes. In H. Verhagen, P. Noriega, T. Balke, and M. de Vos, editors, *Social Coordination: Principles, Artefacts and Theories (SOCIAL.PATH)*, pages 50–57, AISB Convention 2013, University of Exeter, UK, 3–5 Apr. 2013. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour.

[6] A. Omicini. Agents writing on walls: Cognitive stigmergy and beyond. In F. Paglieri, L. Tummolini, R. Falcone, and M. Miceli, editors, *The Goals of Cognition. Essays in Honor of Cristiano Castelfranchi*, volume 20 of *Tributes*, chapter 29, pages 543–556. College Publications, London, Dec. 2012.

[7] A. Omicini. Nature-inspired coordination for complex distributed systems. In G. Fortino, C. Bădică, M. Malgeri, and R. Unland, editors, *Intelligent Distributed Computing VI*, volume 446 of *Studies in Computational Intelligence*, pages 1–6. Springer, 2013. 6th International Symposium on Intelligent Distributed Computing (IDC 2012), Calabria, Italy, 24-26 Sept. 2012. Proceedings. Invited paper.

[8] A. Omicini. Nature-inspired coordination models: Current status, future trends. *ISRN Software Engineering*, 2013, 2013. Article ID 384903, Review Article.

[9] A. Omicini and S. Mariani. The tucson coordination model & technology: A guide. 2013.

[10] A. Omicini and F. Zambonelli. TuCSoN: a coordination model for mobile information agents. In D. G. Schwartz, M. Divitini, and T. Brasethvik, editors, *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187, Pisa, Italy, 8–9 June 1998. IDI – NTNU, Trondheim (Norway).

[11] A. Omicini and F. Zambonelli. Tuple centres for the coordination of Internet agents. In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 183–190, San Antonio, TX, USA, 28 Feb. – 2 Mar. 1999. ACM. Special Track on Coordination Models, Languages and Applications.

[12] A. Ricci, A. Omicini, M. Viroli, L. Gardelli, and E. Oliva. Cognitive stigmergy: A framework based on agents and artifacts. In M.-P. Gleizes, G. A. Kaminka, A. Nowé, S. Ossowski, K. Tuyls, and K. Verbeeck, editors, *3rd European Workshop on Multi-Agent Systems (EUMAS 2005)*, pages 332–343, Brussels, Belgium, 7–8 Dec. 2005. Koninklijke Vlaamse Academie van Belie voor Wetenschappen en Kunsten.

[13] A. Ricci, M. Viroli, and A. Omicini. Programming MAS with artifacts. In R. P. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *3rd International Workshop "Programming Multi-Agent Systems" (PROMAS 2005)*, pages 163–178, AAMAS 2005, Utrecht, The Netherlands, 26 July 2005.

[14] G. Theraulaz and E. Bonabeau. A brief history of stigmergy. *Artificial Life*, 5(2):97–116, Spring 1999.

[15] L. Tummolini, C. Castelfranchi, A. Ricci, M. Viroli, and A. Omicini. "Exhibitionists" and "voyeurs" do it better: A shared environment approach for flexible coordination with tacit messages. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for Multi-Agent Systems*, volume 3374 of *LNAI*, pages 215–231. Springer, Feb. 2005. 1st International Workshop (E4MAS 2004), New York, NY, USA, 19 July 2004. Revised Selected Papers.