ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA IN INGEGNERIA DEI SISTEMI E DELLE
TECNOLOGIE DELL'INFORMAZIONE

COORDINAZIONE DI AGENTI LOGICI GUIDATA DA ISTRUZIONI OPERATIVE

Elaborato in

FONDAMENTI DI INFORMATICA LA

Relatore:
Prof. MIRKO VIROLI

Presentata da:
ENRICO OLIVA

Correlatore:
Prof. ALESSANDRO RICCI

Sessione II
Anno Accademico 2003–2004

A Giacomo e Adriana
che hanno un posto speciale
nel mio cuore

Indice

Introduzione

Un agente è concepibile come una entità autonoma, che incapsula le regole di controllo, 'situata', che risiede in un ambiente, e che persegue proattivamente il suo goal, interagendo con l'esterno. Un insieme di agenti, che interagisce, coopera, si coordina, compete per il raggiungimento di uno scopo globale o sociale, forma quella che viene chiamata una comunità di agenti o Multi Agent System (MAS). Nell'ambito dei MAS la comunicazione diretta è la forma standard di interazione tra agenti, ma non sempre è il miglior modo di ottenere un comportameto organizzato del sistema. Recenti ricerche hanno centrato la loro attenzione su una coordinazine basata sull'ambiente come la stigmergy e più in generale sull'interazione mediata [1, 2].

I sistemi naturali sono spesso fonte di ispirazione per i progetti MAS. Molte forme di insetti sociali o animali come colonie di insetti, termiti, vespe, lupi, stormi di uccelli mostranno delle complesse forme di coordinazione, senza una diretta comunicazione tra loro e senza una particolare intelligenza, se non quella minima di un comportamento razionale, limitata a certo numero di azioni. Il sistema evolve da solo, in autonomia in uno stato di equilibrio auto organizzato. La forma di comunicazione indiretta – mediata attraverso l'ambiente, tra le varie entità del sistema – prende il nome di 'stigmergy'. I sistemi naturali presentano molte analogie con i MAS in quanto sono composti da molte entità autonome in grado di coordinarsi attraverso l'ambiente per raggiungere uno stato di equilibrio. Anche gli agenti come gli insetti possono utilizzare l'ambiente per il raggiungimento del loro scopo sociale. Chiedersi come l'ambiente possa essere di supporto alla coordinazione e cooperazione delle entità presenti nel sistema è un punto fondamentale nella costruzione di un MAS. La tesi si occupa di esplorare la costruzione di un MAS costituito da agenti razionali che interagiscono in maniera mediata.

Un agente razionale è un'entità che interagisce con l'ambiente in cui vive, può percepire l'ambiente attraverso dei sensori e agire su di esso modificando-

lo attraverso degli attuatori. La scelta dell'azione da compiere ne determina la sua razionalità cioè un agente è razionale quando sceglie, attraverso la propria conoscenza del mondo, di eseguire un'azione che nel miglior modo soddisfa il suo scopo.

Lo scenario che si delinea è di stigmergia per agenti intelligenti: cioè il riuscire a sfruttare l'ambiente, in maniera simile ai sistemi naturali, per ottenere delle proprietà del sistema molto interessati quali la decentralizzazione del controllo, l'auto organizzazione e la flessibilità. Riuscire a capire quali caratteristiche debba avere l'ambiente e che comportamento debba avere l'agente per raggiungere tale risultato sono due delle principali domande nella progettazione di un MAS.

In un recente articolo [1] si propone di utilizzare degli *artefatti di coordinazione*, ispirati all'Activity Theory, per realizzare l'interazione mediata. Un artefatto di coordinazione è una infrastruttura che ha lo scopo di migliorare la coordinazione per il conseguimento del task sociale della comunità di agenti [1, 3]. Nella società umana ci sono molti esempi di artefatti di coordinazione come: semafori, segnali stradali, uffici, e lavagne, i quali sono costruiti con l'intenzione di semplificare l'interazione. Dal punto di vista dell'agente, l'artefatto è visto come una entità dell'ambiete su cui può compiere delle azioni ed avere delle percezioni. Un artefatto è caratterizzato da una interfaccia d'uso che è definita in termini di operazioni, da delle istruzione operative che contengono le informazioni su come usare l'artefatto e da un comportamento di coordinazione dell'artefatto che è espresso in termini di regole all'interno dell'infrastruttura. Un agente intelligente ha la capacità di seguire le istruzioni operative e così utilizzare l'artefatto e prendere parte a complessi scenari di coordinazione.

Nell'ambito della tesi consideriamo MAS costituiti da agenti logici basati su inferenza Prolog per supportare minime capacità razionali, in particolare, per gestire la base di conoscenza locale sulla quale inferire.

Utilizziamo inoltre le istruzioni operative (Operating Instructions, OI) [4], una tecnica studiata per consentire ad agenti di utilizzare in modo razionale un artefatto. Queste infatti specificano quali azioni, quale dinamica e quale effetto sulla base di conoscenza. Permettono di collegare lo stato mentale dell'agente espresso dalla base di conoscenza con le azioni da eseguire sull'artefatto.

Come primo contributo, in questa tesi si è studiato un framework realizzativo per questo tipo di MAS basato su TuCSoN come infrastruttura di

coordinazione e tuProlog come motore inferenziale Prolog.

TuCSoN (Tuple Centres Spread over the Network) ci permette di realizzare l'artefatto come tulpe centre – uno spazio di tuple programmabile attraverso la programmazione ReSpecT e che esprime il comportamento di coordinazione del sistema [5]. Il motore tuProlog, che è un inteprete Prolog interfacciabile a Java, ci ha consentito di individuare un'architettura generale per agenti logici in Java in grado di supportare le OI collegate con la propria base di conoscenza. Si è inoltre sviluppato un interprete di istruzioni operative rappresentandone la semantica operazionale, formalizzata attraverso un Labeled Transition System (LTS), in termini di un predicato Prolog. Attraverso questa struttura gli agenti logici sono in grado di leggere ed interpretare le istruzioni operative.

Come secondo contributo di questa tesi, si è applicato questo framework relativo al Minority Game (MG), che rappresenta un semplice modello di sistema dove gli agenti in modo simultaneo e adattativo competono per limitate risorse. Il sistema evolve attraverso l'interazione di agenti che costruiscono scelte su un ambiente che agisce per coordinarne il comportamento. Tali agenti hanno strategie eterogenee, beliefs e uno scopo da realizzare. Il MG consiste nel far vincere la scelta minoritaria su due scelte possibili. Tale gioco serve a mettere in evidenza delle proprietà di coordinazione emergenti del sistema, conseguenza di una interazione mediata tra i giocatori.

L'implementazione del sistema che abbiamo realizzato permette di simulare gli andamenti di giocate del MG con un'architettura basata su TuCSoN. La variazione dei parametri generali del gioco può avvenire online, ossia modificando le regole di coordinazione del MAS senza dover fermare e far ripartire il sistema.

Gli agenti logici che giocano al MG costruiscono la loro scelta su una propria strategia di gioco basata sulla memoria dei casi precedenti. Gli agenti possiedono da 1 a N strategie con N un parametro di sistema, alle quali viene attribuito un punteggio virtuale, incrementandolo se tale strategia risulta vincente. Viene sempre giocata la strategia con punteggio più alto. La lunghezza delle strategie è determinata dalla memoria m sui risultati del sistema perchè rappresenta una scelta rispetto a tutti i possibili m stati in cui il sistema si può trovare.

La realizzazione del MG in termini di un MAS si è composta delle seguenti fasi. Si è realizzata la scrittura delle specifiche OI degli agenti giocatori del MG, utilizzando la definizione di vari processi. La specifica ReSpecT è stata

realizzata in modo da garantire la coordinazione di un numero N di agenti per più giochi consecutivi e per più sessioni. L'ambiente di simulazione è realizzato in Java e permette la modifica dei parametri di configurazione ed esegue durante il gioco il ruolo di osservatore dell'artefatto graficando i risultati real time. L'ambiente grafico costruito permette di lanciare simulazioni ottenendo via via i risultati medi e di varianza, che troviamo corrispondere a quelli riportati in letteratura. E' anche possibile modificare i parametri del sistema osservando il cambio di equilibrio di sistema, ottenuto modificando i parametri del behaviour di coordinazione dell'artefatto TuCSoN realizzato.

Capitolo 1

Agenti logici e artefatti

Nell'ambito della ricerca sui Multi Agent System (MAS), nuova disciplina scientifica che studia le proprietà e le caratteristiche dei sistemi formati da agenti, il nostro studio riguarda la coordinazione di agenti logici guidata da istruzione operative. Il primo problema è quello di creare un'entità autonoma in grado di compiere delle scelte razionali, il secondo è quello di mettere in condizione una società di agenti di cooperare e coordinarsi tra loro.

Nella prima sezione descriviamo i sistemi naturali che hanno molte analogie con i Multi Agent System (MAS), introdotti nella seconda sezione, che sono fonte di ispirazione per nuove architetture. Nella terza esaminiamo l'importanza della conoscenza in un agente e come questa ne determini caratteristiche di intelligenza. Nella quarta sezione consideriamo l'agente logico e nella quinta gli artefatti di coordinazione.

1.1 Sistemi Naturali

I sistemi naturali presentano molte analogie con i MAS in quanto sono composti da molte entità autonome in grado di comunicare attraverso l'ambiente e coordinarsi per raggiungere uno stato di equilibrio. Il sistema evolve da solo, in autonomia senza sollecitazioni esterne in uno stato auto organizzato dal quale riusciamo a ricavarne le leggi e le regole emergenti. Come avviena nel sistema solare, negli ecosistemi, nelle galassie, nei pianeti, nelle cellule e negli atomi .... Può sembrare che tali sistemi siano in contrasto con il secondo principio della termodinamica perchè tendono ad ordinarsi senza nessuna sollecitazione esterna. Ma non è cosi, perchè non si tratta di sistemi chiusi

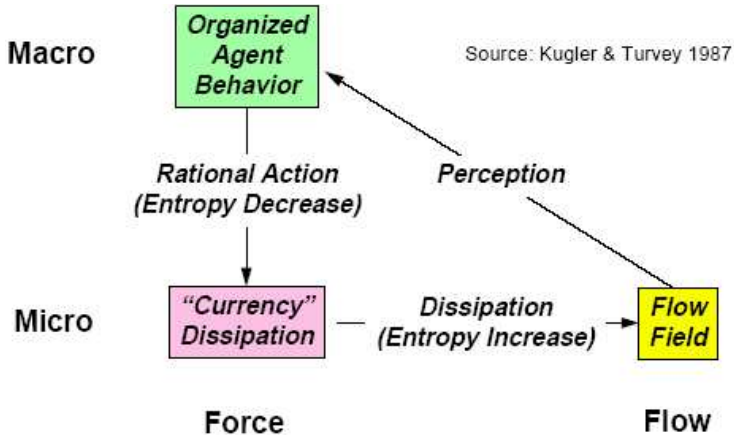

Figura 1.1: Macro organizzazione attraverso Micro dissipazione

e in equilibrio ma essi usano un flusso di energia per guidare i loro processi di crescita d'ordine, dissipando calore o altri modi che conducono ad un aumento di entropia.

Molte forme di insetti sociali o animali come colonie di insetti, termiti, vespe, lupi, stormi di uccelli mostranno delle complesse forme di coordinazione senza una diretta comunicazione tra loro e senza una particolare intelligenza, se non quella minima di un comportamento razionale limitata a certo numero di azioni. Per esempio le formiche, seguendo semplici regole di interazione con l'ambiente si coordinano fra loro facendo emergere una società complessa e organizzata. Il comportamento emergente della colonia di formiche è una rete di percorsi che collegano il nido alle sorgenti di cibo, nel minor percorso possibile. Nelle termiti osserviamo la costruzione di strutture molto complesse, negli uccelli la formazione di stormi, nei lupi l'emersione di strategie di attacco complesse e molto efficaci tutte queste forme di coordinazione avvengono senza interazioni dirette tra i membri della società [2].

Nel modello di tale sistema si ha un macro livello dove gli agenti si auto organizzano perchè le loro azioni sono in relazione ad un processo dissipativo disorganizzato al micro livello. Quindi la diminuzione di entropia al livello superiore è compensata da un aumento al micro livello. L'effetto dissipativo nell'ambiente genera un campo che gli agenti possono percepire e rinforzare e con il quale possono orientarsi.

Nel caso delle formiche il deposito dei feromoni sull'ambiente produce il campo che tutte possono percepire ed utilizzare per orientarsi. L'ambiente deve provvedere a tre operazioni:

1. *aggregare* i feromoni lasciati dai singoli agenti cioè fondere l'informazione;

2. *evaporare* i feromoni dopo un certo tempo che non contengono più informazione;

3. *diffondere* i feromoni nelle vicinanze cioè distribuire l'informazione localmente.

Cosi la nuova informazione è facilmente integrata nel campo e la vecchia viene eliminata attraverso il meccanismo dell'evaporazione.

Un elemento fondamentale di tali sistemi è l'ambiente attraverso il quale le varie entità si coordinano per il raggiungimento del *task*(scopo) globale o sociale. L'ambiente quindi funge chiaramente da artefatto di coordinazione vedi sezione 1.5.

Un altro aspetto che emerge dall'osservazione dei sistemi naturali è la forma di comunicazione indiretta, mediata attraverso l'ambiente, tra le varie entità del sistema che prende il nome di 'stigmergy'. Con la stigmergy ci si riferisce a quel processo di coordinazione mediata attraverso l'ambiente che può essere modificato dalle azioni degli agenti.

1.2 Sistemi multiagente

Si può definire un agente in maniera astratta come un'entità che interagisce con l'ambiente in cui vive. L'agente può percepire l'ambiente attraverso dei sensori e agire su di esso modificandolo attraverso degli attuatori. La scelta dell'azione da compiere determina la razionalità dell'agente cioè un agente è razionale quando sceglie, attraverso la propria conoscenza del mondo, di eseguire un'azione che nel miglior modo soddisfa il suo scopo. Ad esempio se il mio scopo è di rimanere asciutto e credo che stia piovendo io massimizzo il mio interesse prendendo un ombrello; questo è un esempio di comportamento razionale [6].

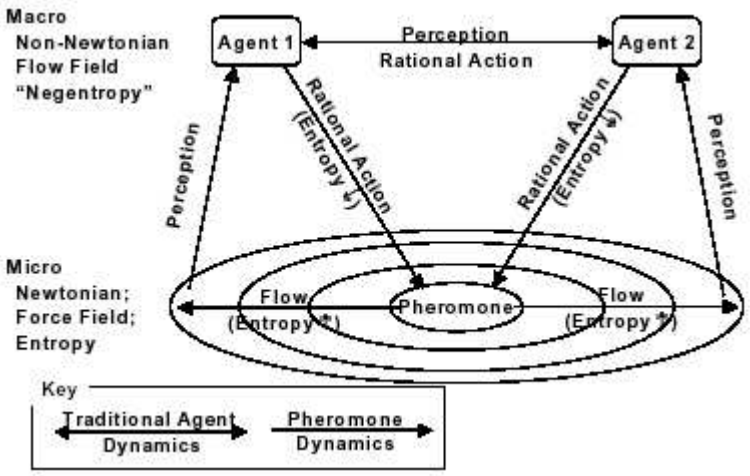

Figura 1.2: Stigmergy

La razionalità è una caratteristica tipica degli esseri viventi che sono degli ottimi costruttori di scelte razionali, perchè vivono in un ambiente straordinarimente ricco, complesso e in continuo cambiamento. Possiamo osservare la razionalità sia in un uomo che esegue sequenze di azioni e interazioni con molto complesse con l'ambiente, ma anche nel comportamento di una formica nell'azione di deposito di un feromone.

Le decisioni dell'agente e quindi le sue azioni sono sotto il suo controllo, e non guidate da altri tale caratteristica ne delinea l'autonomia. Un agente è dunque concepibile come una entità autonoma, cioè che incapsula le regole di controllo, 'situata' e che persegue proattivamente il suo goal, interagendo con l'esterno. L'agente non solo reagisce a stimoli esterni ma agisce in maniera autonoma per perseguire il suo scopo. Riepilogando le proprietà che caratterizzano un agente sono:

- **Autonomia** - capacità di agire senza interventi esterni diretti. L'agente deve avere il controllo totale sui suoi stati interni e sulle azioni che può eseguire.

- **Interatività** - capacità di comunicare con l'ambiente che lo circonda.

- **Adattatività** - capacità di percepire l'ambiente che lo circonda e

rispondere in modo tempestivo ai cambiamenti che avvengono in esso; modificando anche il proprio comportamento.

- **Reattività** capacità dell'agente di rispondere ad eventi.
- **Mobile** - capacità di trasportare se stesso da un ambiente ad un altro.
- **Proattività** - l'agente non risponde semplicemente in relazione a sollecitazioni provenienti dall'ambiente, ma deve essere capace di attivarsi per il conseguimento del proprio scopo (goal-driven).
- **Abilità sociale** - capacità degli agenti di vivere in un contesto sociale cioè collaborare e cooperare con altri agenti per il conseguimento di un comune obiettivo, o per un la realizzazione di uno scopo sociale. **Rational** - capacità di un agente di agisce per soddisfare i suoi goal interni.

Un insieme di agenti, che interagisce, coopera, si coordina, compete per il raggiungimento di uno scopo globale o sociale, forma quella che viene chiamata una comunità di agenti o MAS. La progettazione di un sistema MAS parte dalla costruzione dei singoli agenti per i quali non è decisivo il linguaggio con cui vengono implementati ma l'architettura utilizzata. Le tre architetture classiche di costruzione di un agente sono:

- logic-based architectures
- reactive architectures
- belief-desire-intention architectures

1.2.1 Architettura BDI

L'architettura BDI prende spunto da studi sull'osservazione del comportamento umano attraverso esperimenti scientifici studiati nell'ambito delle scienze cognitive. Il modello BDI vuole imitare il comportamento umano, in particolare il fatto che gli esseri umani siano guidati da tre 'attitudini':

- Beliefs (credenze): corrispondono alle informazioni che l'agente ha sul mondo;

- Desires (goal): rappresentano lo stato del mondo che l'agente vuole ottenere;
- Intentions (intenzioni): rappresentano le azioni che l'agente vuole eseguire;

Tale modello è basato sulla teoria del ragionamento pratico di Bratman, in cui per ragionamento pratico si intende la capacità di una persona di decidere ogni giorno della sua vita e momento per momento quale azione eseguire al fine di raggiungere determinati obiettivi. Si distinguono due distinte fasi: la prima è quella di decidere qual è l'obiettivo che si vuole ottenere (deliberation); la seconda è decidere il processo (o sequenza delle azioni)attraverso cui ottenere il l'obbiettivo (means-ends reasoning). Si può osservazione è che non tutti i desideri dell'agente, anche se consistenti, potranno essere realizzati perchè non compatibili con le sue intenzioni. Le intenzioni definiscono lo stato mentale dell'agente e sono importanti nel modello del ragionamento pratico, perchè vincolano e limitano il ragionamento da compiere per la scelta dell'azione. Ad esempio se è nostra intenzione mangiare un panino, quando dovremo decidere cosa fare sceglieremo solo le azioni compatibili con la nostra intenzione, aprendo il frigo prenderemo solo in considerazione gli affettati scartando il resto degli oggetti presenti. Una prima versione di un modello di agente BDI esegue il seguente ciclo:

```
ripeti true
    osservo il mondo;
    aggiorno il mio stato interno;
    delibero sulle prossime intenzioni;
    faccio un piano delle intenzioni;
    eseguo il piano;
fine ripeti
```

1.2.2 Architettura logic-based

Una tipica architettura per un agente MAS è basata sulla logica. Un agente logico ha una propria rappresentazione del mondo e usa un processo di inferenza per derivarne una nuova ed utilizzarla per dedurre cosa fare (quali azioni compiere). La rappresentazione della propria conoscenza e del proprio goal è espressa attraverso una logica del primo ordine o equivalente. Una particolare forma di agente logico è il BDI.

1.3 La base di conoscenza

Un agente intelligente ha bisogno di conoscenza sul mondo per raggiugere delle buone decisioni e determinarne un comportamento di successo. Si potrebbe affermare che un agente è intelligente se ha una propria base di conoscenza, altrimenti l'intelligenza sarebbe solo un'elaborazione di input. L'interpretazione delle percezioni e l'elaborazione di risposte intelligenti richiede conoscenza pregressa e tale conoscenza serve a interpretare le percezioni e a correlarle con quelle antecedenti [7]. Il comune concetto di esperienza cattura esattamente l'idea di una conoscenza pregressa alla quale si fa riferimento per l'elaborazione di risposte. Un agente può memorizzare nuove sentenze ma anche ricavarne di nuove attraverso un meccanismo di inferenza ed utilizzarle per decidere quali azioni compiere. Va definita la semantica cioè il valore di verità o falsità di ogni sentenza e verificata la consistenza della base di conoscenza. La Knowledge Base (KB) ha un ruolo fondamentale anche in quei sistemi con un ambiente parzialmente osservabile: perchè un agente è in grado di combinare la conoscenza pregressa con quella acquisita correntemente e inferire così degli aspetti nascosti. Si pensi ad esempio ad un medico che formula la diagnosi deducendola da tutti i sintomi del paziente. La capacità di apprendere nuove competenze e di addattarsi ai cambiamenti dell'ambiente sono altre caratteristiche per le quali inseriamo la KB negli agenti. La conoscenza e il ragionamento sono caratteristiche molto importati per gli agenti perchè ne determinano un comportamento di successo difficilmente ottenibile senza tali proprietà. Attraverso un'operazione di *Tell* si aggiunge conoscenza dalle percezioni. Mentre con la *Ask* facciamo interrogazioni e inferiamo nuova conoscenza.

1.4 Agenti Logici

La logica è quella scienza che studia le leggi che si suppone governino il funzionamento della mente. Aristotele è stato uno dei primi a definire il 'giusto pensiero' attraverso un processo di ragionamento irrefutabile. Il suo sillogismo definisce un modello che produce sempre corrette conclusioni quando le premesse sono corrette. Nel 1958 Piaget afferma: 'il ragionamento non è niente altro che la logica proposizionale'.

Un agente logico è basato su una qualche forma di logica. Ha la propria base di conoscenza formata da un insieme sentenze in termini di un linguag-

gio logico. Anche le percezioni e la capacità di fare inferenza deve essere espressa nei termini di tale linguaggio. Un modo per agire razionalmente è quello di ragionare logicamente. L'inferenza è una maniera per ottener un comportamento razionale. Per esprimere la KB potremo utilizzare una logica proposizionale dotata di regole di inferenza che risulta però poco espressiva, perchè assume un mondo di puri fatti e non si riesce a distiguere tra relazioni e funzioni o a fare discorsi 'intensionali'[8]. Molto performante è la logica dei predicati del primo ordine attraverso la quale possiamo esprimere oggetti, relazioni tra oggetti e funzioni [9, 10]. I ragionamenti ottenibili con una logica proposizionale possono essere sviluppati anche in quella dei predicati. In questo modo aumentiamo l'espressività del nostro linguaggio di rappresentazione anche per esprimere meglio la base di conoscenza. Un agente logico comunque si basa sui fondamentali concetti della rappresentazione logica e del ragionamento logico. Le sentenze che formano la base di conoscenza dell'agente devono rispondere alla sintassi del linguaggio logico che ci dice se queste sono ben formate. Ragionare significa generare e manipolare questa configurazione. Una logica deve anche definire la semantica del linguaggio cioè che significato dare alle sentenze. La semantica del linguaggio definisce il valore di verità di ogni sentenza rispetto ad ogni possibile stato del mondo (o modello). Il ragionamento logico deriva dalla relazione di conseguenza logica :

$$\alpha \models \beta \tag{1.1}$$

significa che α è conseguenza logica di β se e solo se in ogni modello in cui α è vera anche β è vera.

L'applicazionde del meccanismo di conseguenza logica conduce a derivare delle conclusioni quindi all'inferenza logica. L'algoritmo per il calcolo dell'inferenza prova tutti i possibili modelli in cui α è vera in tutti i modelli nei quali KB risulta essere vera. Esprimiamo l'inferenza o la deduzione logica come:

$$KB \vdash \alpha \tag{1.2}$$

cioè α è derivato dalla KB. Il model checking è un algoritmo che calcola l'inferenza provando tutti i possibili modelli. Funziona se lo spazio dei modelli è finito. Per tale algoritmo valgono due proprietà: soundness e completezza in quanto esso preserva la verità delle sentenze ottenendole solo come conseguenza logica, e ha la capacità di derivare tutte le sentenze richieste.

Quello che abbiamo descritto è un processo di ragionamento in cui le

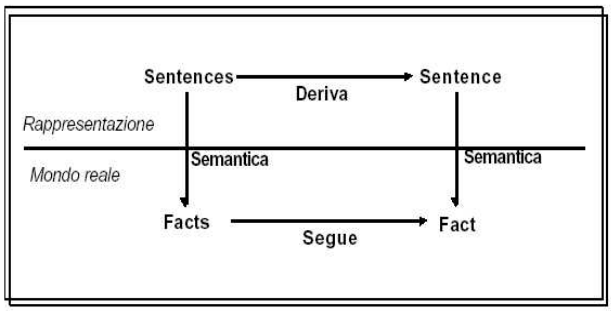

Figura 1.3: Conseguenza logica

conclusioni sono garantite essere vere se le premesse ovviamnente sono vere. Questo implica che, se la nostra base di conoscenza è una vera espressione del mondo reale, ogni sentenza derivata dalla KB attraverso un processo di inferenza è anche vera nel mondo reale!

Occorre esaminare il problema della connessione tra il ragionamento logico dell'agente e il mondo reale nel quale l'agente esiste. In particolare come possiamo determinare la correttezza della KB rispetto al mondo? La verità di tutte le deduzioni legata all'ipotesi di verità della KB. Una prima risposta è che le percezioni legano il nostro agente al mondo reale e la verità della KB è garantita da un processo di apprendimento [6]. Le informazioni memorizzate nella KB devono avere una attendibilità non minore del 100%.

Basare i nostri agenti su una forma di logica e aggiungere una KB, li porta ad essere intelligenti e dotati di un comportamento razionale.

```
Ripeti
    <percezione>,
    <aggiornamento KB>,
    <inferenza/pianificazione>, (scelta dell'azione)
    <eseguo l'azione>,
Fine ripeti
```

L'ontologia definita per la logica del primo ordine è: dominio del discorso, relazioni tra oggetti e funzioni tra oggetti. La sintassi del linguaggio è formata da un alfabeto composto da cinque insiemi: costanti, funzioni, predicati(relazioni), variabili e connettivi logici ($\leftrightarrow, \vee, \wedge, \exists, \forall, \neg, ()$). Le costanti rappresentano le entità del dominio del discorso. Le variabili possono rapp-

resentare diverse entità del dominio. Una funzione individua univocamente un oggetto, mediante la relazione tra oggetti.

1.5 Artefatti di Coordinazione

Lo studio dell' interazione tra agenti è un punto chiave per la creazione di sistemi multi agente in cui gli agenti cooperano per eseguire un task globale o sociale. A tale scopo è necessario che gli agenti comunichino o meglio interagiscano tra loro. Una via ormai consolidata di interazione tra agenti è la comunicazione diretta basata sulla teoria delle *Speech Acts*. Gli speech act sono una particolare classe del linguaggio naturale dove la comunicazione ha la stessa caratteristica di una azione cioè cambia lo stato del mondo proprio come le azioni fisiche. Da tale teoria sulla comunicazione diretta ne è nato un Agent Communication Languages (ACLs) standard FIPA.[6]

Sia nel mondo naturale, ad esempio nelle colonie di insetti, che nel mondo umano si hanno degli esempi di interazione mediata cioè forme di collaborazione senza l'utilizzo di una comunicazione diretta tra le entità ma la comunicazione è mediata attraverso l'ambiente. Tale può anche essere definita indiretta. Ad esempio le formiche comunicano fra loro depositando e percependo feromoni dall'ambiente formando una società auto organizzata e molto efficiente. E' una forma di coordinazione che avviene attraverso una comunicazione indiretta chiamata stigmergy.

Anche gli uomini spesso collaborano non parlandosi direttamente ma attraverso vari tipi di strutture di mediazione: semafori, segnali stradali, uffici, lavagne i quali sono costruiti con l'intenzione di semplificare l'interazione.

La nozione di artefatto di coordinazione, ispirato alla Activity Theory [3, 1], è una struttura per i sistemi MAS che media l'interazione tra agenti realizzando un determinato task di coordinazione.

Dal punto di vista dell'agente, l'artefatto è visto come una entità dell'ambiete su cui può compiere delle azioni ed avere delle percezioni. Un' azione va a modificare lo stato del mondo e attraverso una percezione leggiamo il nuovo stato. Ad esempio considerando l'artefatto di coordinazione ascensore, premendo il pulsate di chiamata eseguiamo una azione sull'artefatto questa va a modificare lo stato del mondo facendo arrivare l'ascensore al nostro piano e la nostra percezione sarà di poter salire.

Con più persone che eseguono richieste è l'ascensore che coordina il rag-

giungimento del task globale cioè che ogni persona raggiunga il piano desiderato; questo avviene senza che gli utenti si accordino fra loro su che piano andare.

Nell'esempio dell'ascensore occorre che l'agente conosca le regole con cui utilizzare l'artefatto: premere il pulsante chiamata, attendere l'arrivo al piano, salire, premere pulsante di destinazione... sono tutte azioni che devono essere eseguite secondo una certo ordine, dettato dallo stato in cui si trova l'artefatto. Con la percezione possiamo determinare lo stato dell'artefatto e quindi inferire in maniera corretta la nuova azione da compiere. In questa maniera operiamo una scelta razionale sull'insieme delle azioni possibili. La sequenza di istruzioni di cui deve essere dotato un agente per utilizzare l'artefatto è chiamata *operating instructions*. Queste specificano il protocollo di azione e percezione consentito dall'artefatto.

L'artefatto è caratterizzato da un insieme di operazioni consentite all'agente come richiesta informazioni, notifica di risultati, che ne definiscono l'interfaccia d'uso. L'agente agisce sull'artefatto con un azione e può percepirne il risultato ma non può avvenire il contrario cioè l'artefatto è un componente non proattivo che non agisce se non in seguito ad una richiesta.

Il comportamento di coordinazione è insito nell'artefatto ed è descritto dall'insieme di regole che gestiscono l'interazione mediata. Nel nostro esempio alla pressione del pulsante di chiamata l'ascensore sa che deve recarsi al piano questa è un comportamento di coordinazione dell'artefatto.

Tale comportamento può subire delle modifiche nel tempo in base alle esigenze di coordinazione del sistema. Questa è una caratteristica molto utile per l'ingegnerizzazione dei sistemi e perchè si può cosi portare autonomamente il sistema da uno stato di equilibrio in un altro, rendere l'artefatto adattativo alle condizioni della società. Ad esempio l'ascensore potrebbe aumentare la propria velocità di salita se il numero di richieste è in forte aumento.

Nel caso della stigmergia è l'ambiente che funge da artefatto di coordinazione, incapsula in se le regole di coordinazione della società e il sistema nel complesso risulta avere delle proprietà molto interessanti: flessibilità, robustezza e auto organizzazione.

Questo significa che se poniamo degli ostacoli, imprevisti nell'ambiente il sistema riesce autonomamente a trovare un nuovo stato di equilibrio. Ad esempio nelle formiche se gettiamo un sasso in un percorso già consolidato e ne interrompiamo il flusso di formiche queste riescono a trovare percorsi

alternativi utilizzando solo le semplici azioni di cui sono dotate.

Nell'esempio dell'ascensore invece l'ostacolo aggiunto deve essere comunque previsto nella sequenza delle azioni dell'agente o nel comportamento dell'artefatto per far si che il sistema continui la realizzazione del suo task.

Potremo affermare che le formiche e l'ambiente sono programmate per superare determinati imprevisti ma non è cosi sono le semplici regole di interazione che danno la possibilità al sistema nel suo complesso di adattarsi ai cambiamenti.

1.5.1 Istruzioni operative

L'agente deve conoscere le regole per l'utilizzo dell'artefatto di coordinazine. Ad esempio il semaforo è una entità di coordinazione fin tanto che tutti gli agenti conoscono le regole di utilizzo dell'artefatto cioè con il rosso ci ferma e con il verde si passa. Se gli agenti non rispettassero queste semplici regole o non le conoscessero non si raggiungerebbe più il risultato finale di coordinazione nel caso del semaforo l'incrocio sarebbe sempre pieno di automobili incidentate. Le istruzioni operative (Operating Istructions OI) hanno il compito di fornire all'agente le regole di uso dell'artefatto, sono come un manuale di istruzioni che l'agente segue per ottenere il proprio scopo. Specificano essettamente la sequenza delle azioni e percezioni che l'agente deve compiere o percepire per utilizzare il servizio di coordinazione fornito dall'artefatto.

Le OI formano un protocollo di comunicazione molto flessibile perchè è sossibile specificare l'esecuzione delle azioni in sequenza, in alternativa o in parallelo in base allo stato dell'agente o alla percezione avvenuta dall'infrastruttura di coordinazione. L'albero delle possibili sequenze di istruzioni esprime la varietà di comportamenti diversi che l'agente può mostrare.

Le OI, in una visione di agente BDI, esprimono l'intezione dell'agente cioè l'insieme delle azioni che vuole eseguire. Tali istruzioni vengono completate specificando delle precondizioni alle azioni e degli effetti alle percezioni che vengono espressi in termini della base di conoscenza dell'agente questo per collegare l'interazione con lo stato mentale dell'agente. La precondizione deve essere verificata sulla base di conoscenza, per consetire l'azione a cui è collegata mentre l'effetto, associato a una percezione, va a modificare lo stato interno dell'agente. Le istruzioni in un dato momento forniscono un insieme di azioni possibili di cui possiamo utilizzare solo quelle con le precondizioni valide, riducendo cosi le possibilità di scelta dell'agente. Se l'insieme risul-

tante è uguale a uno l'unica azione risultante verà immediatamete eseguita. E' possibile costruire un agente che utilizzi le OI e che attraverso il meccanismo delle precondizioni e effetti riesca in maniera autonoma a interagire con l'artefatto e coordinarsi. Il collegamento delle istruzioni allo stato mentale dell'agente è molto importane perchè ne va a modificare le credenze su cui base le scelte future ottenendo cosi un comportamento diverso.

Il protocollo di interazione espresso dalle OI deve essere noto all'agente prima di utilizzare l'artefatto. Potrebbe essere incluso nella sua base di conoscenza o scaricato attraverso una infrastruttura di supporto o risiedere sull'artefatto di coordinazione. Tutte queste possibili soluzioni sono equivalenti e dipendeti dall'architettura del sistema.

Capitolo 2

Tecnologie di riferimento

Nel presente capitolo vengono presentate le tecnologie di riferimento per la costruzione di agenti logici e degli artefatti di coordinazione: tuProlog come motore inferenziale, TuCSoN come infrastruttura di coordinazione e Respect come linguaggio per esprimere le leggi di coordinazione. Queste tecnologie innovative permettono l'ingegnerizzazione e la creazione di sistemi multi agente basati su agenti logici.

2.1 tuProlog

Nella programmazione logica in Prolog un problema da risolvere diventa un quesito sulla base di conoscenza espressa come sequenza di clausole. L'interprete Prolog funziona come un dimostratore di teoremi, che cerca di derivare la conclusione dalle premesse attraverso le regole di inferenza con una strategia di ricerca depth-fist con backtracking [9, 10]. Dal punto di vista dell'utente l'interprete si assume il compito di verificare che il goal sia o meno una *conseguenza logica* delle assunzioni rappresentate nel programma. In caso di successo la risposta conterra il binding delle variabili presenti nel goal. Ad esempio un goal viene presentato come un predicato del tipo: `:- temperatura(X).` su una base di conoscenza `temperatura(50).` che ha il seguente significato: esiste un qualche elemento (un X) per cui il predicato temperatura(X) è vero. Questo viene chiesto alla base di conoscenza da cui si inferirà la soluzione. Il goal in questo caso banale ha successo e nella soluzione la variabile X sarà legata il valore 50 (X/50).

tuProlog è un motore inferenziale Prolog scritto in Java che può essere

integrato facilmente in applicazioni Java. I componenti tuProlog in Java possono essere usati sia in maniera dichiarativa logica che imperativa. Importando i `packages alice.tuprolog.*` abbiamo la visibilità di tutte le classi necessarie all'utilizzo del motore inferenziale. Attravero tuProlog riusciamo a dotare un agente Java della capacità di inferire e quindi eseguire dei ragionamenti razionali. E' evidente come ogni linguaggio abbia il suo particolare potere espressivo e che vada utilizzato nel suo giusto settore. Cioè non possiamo chiedere al Prolog di gestire grandi insieme di oggetti in maniera efficiente nè al Java di eseguire dell'inferenza logica. Ogni paradigma ha le sue caratteristiche e nell'ingegnerizzazione dei sistemi vanno considerate accuratamente.

2.1.1 Utilizzo di tuProlog da Java

Tutti i tipi di dati in Prolog sono mappati su oggetti Java. La classe Term è la classe base per esprimere gli atomi e i termini composti rappresentati dalla classe Struct. Le strutture dati di base del Prolog in Java sono: `Term`, che è una classe astratta che rappresenta un generico termine Prolog e definisce le basilari operazioni sui termini; `Var` che deriva dalla classe Term e rappresenta le variabili tuProlog. Può essere definita anonima o costruita con un nome che inizi con la lettera maiuscola; `Struct` che deriva da Term e rappresenta i termini tuProlog non tipizzati come atomi, liste e termini composti; essi sono caratterizzati da un funtore e da una lista di argomenti.
Esempio di creazione di una struttura:

```
Term tempOggi= new Struct("temperatura",
new Struct("lunedi"),
new Int(16));
temperatura(lunedi,16)
```

Le principali classi utilizzate in Java sono: `Prolog`, che rappresenta il motore inferenziale Prolog; da tale classe si può settare o prelevare la teoria logica, caricare le librerie e dimostrare i goal sotto forma di termini rappresentati come oggetti termine o come stringhe; `Theory` questa classe rappresenta le teorie Prolog. Una teoria è formata da una serie di clausole o direttive seguite da un '.' e uno spazio. L'istanza della classe è costruita passandogli o la striga della teoria logica o un input stream;

`SolveInfo` questa classe rappresenta il risultato della dimostrazione. L'istanza di questa classe è data dal metodo solve della classe Prolog, utilizzato per la dimostrazione del goal. In particolare la classe testa il successo della dimostrazione e accede al termine di soluzione con tutte le sue variabili.

Il seguente codice crea il motore inferenziale prolog, crea un nuovo oggetto Theory leggendo la teoria da un file di testo e settandola nel motore inferenziale.

```
try {
  FileInputStream f=new FileInputStream("senderTheory.pl");
  Prolog Pengine=new Prolog();
  Pengine.setTheory(new Theory(f));

} catch (InvalidTheoryException e1) {
  System.out.println("Errore nella teoria" );
  e1.printStackTrace();
}
```

Esecuzione di un goal e stampa di tutte le possibili soluzioni.

```
SolveInfo sol=null;
try {
  SolveInfo sol = Pengine.solve("firststate(X).");
  while (sol.isSuccess()){
    System.out.println("solution: "+sol.getSolution());
    if (Pengine.hasOpenAlternatives()){
      sol=Pengine.solveNext();
    } else {
    break;
  }
 }
}catch (MalformedGoalException e1) {
   System.out.println("Errore sintassi del Goal" );
   e1.printStackTrace();
}catch (NoSolutionException e) {
   e.printStackTrace();
}
```

2.1.2 Invocare Java da tuProlog

Dal punto di vista di tuProlog si riescono ad utilizzare oggetti, classi, packages di Java come se fossero dei termini Prolog. Si possono usare da Prolog tutte le librerie Java come ad esempio Swing e JDBC migliorando cosi le capacità del motore tuProlog con grafica e accesso al database. Possiamo anche creare dei nostri oggetti java e richiamarli in tuProlog.

Ad esempio il seguente codice mostra come la classe contatore venga utilizzata in una teoria logica:

```
public class Counter {
  private int value;
  public  String name;
  public Counter(){}
  public Counter(String n){ name=n; }
  public void setValue(int val){ value=val; }
  public int  getValue() { return value; }
  public void inc()      { value++; }
  public void setName(String s) {
  name = s;
  }
  static public String getVersion() { return "1.0"; }
}

test :-
  java_object('Counter',['Counter'],myCounter),
  myCounter <- setValue(5),
  myCounter <- inc,
  myCounter <- getValue returns Value,
  write(Value),
  class('Counter') <- getVersion returns Version,
  [class('java.lang.System')|out] <- get(StdOut),
  StdOut <- println(Version),
  [myCounter|name] <- get(MyName),
  StdOut <- println(MyName),
  [myCounter|name] <- set('OkCounter'),
  java_object('Counter[]',[10],ArrayCounters),
  java_array_set(ArrayCounters,3,myCounter).
```

La libreria JavaLibrary di tuProlog offre la possibilità al Prolog di accedere direttamente ai componenti Java. I principali predicati built-ins sono [11]: `java-object(ClassName,ArgumentList,ObjectRef)` , che è il predicato attraverso il quale si creano nuovi oggetti Java, dove ClassName è il nome della classe, ArgumentList la lista degli argomenti da passare al costruttore della classe e ObjectRef il riferimento al nuovo oggetto creato; `ObjectRef` <− `MethodName(Arguments) returns Term` che è il predicato usato per invocare i metodi degli oggetti; dove ObjectRef è il riferimento all'oggetto, MethodName è il nome del metodo che si vuole invocare, Arguments è lista di argomenti e Term è il valore di ritorno del metodo; Esempio di utilizzo di predicati buit-in:

```
java_object('Counter',['Counter'],myCounter)
myCounter <- getValue returns Value
```

Se l'oggetto ha dei metodi statici questi possono essere invocati attraverso il termine composto **class(ClassName)** ad esempio:

```
class('Counter') <- getVersion returns Version
```

.: è un operatore infisso usato con i pseudo metodi get e set per accedere ai campi pubblici della classe. Ad esempio:

```
myCounter.name <- get(MyName),
myCounter.name <- set('OkCounter')
```

2.2 TuCSoN

Tuple Centres Spread over Networks TuCSoN è una infrastruttura di coordinazione per per sistemi multi agente. Un artefatto di coordianzione ha lo scopo di mediare l'interazione tra gruppi di agenti evitando una comunicazione diretta.

TuCSoN è in grado di fornire una molteplicità di artefatti di coordinazione, è ispirato alla coordinazione basata su lavagne che utilizza un condiviso spazio dei dati per la comunicazione. L'idea è quella di avere uno spazio proprio come una lavagna su cui gli agenti depositano e leggono dei messaggi,

in questo modo gli agenti riescono a interagire fra loro senza conoscersi con una forma di comunicazione indiretta.

I messaggi lasciati sullo spazio dei dati sono scritti in forma di tuple logiche e il recupero delle tuple, in seguito alle richieste degli agenti, avviene attraverso il meccanismo dell'unificazione. I tuple centre forniti da un nodo TuCSoN sono delle lavagne programmabili il cui comportamento può essere programmato in risposta ad eventi di comunicazione, cioè un tuple centre ha la capacità di definire le regole di coordinazione e farle evolvere dinamicamente con l'andamento del sistema.

Uno spazio di condivisione dei dati così definito potrebbe richiamare ad una similitudine con la nozione di stigmergia dove l'ambiente incapsula le regole di coordinazione come un tuple centre. Ad esempio l'ambiente delle formiche che ha semplici regole di coordinazione quali la somma ed evaporazione dei feromoni, sembrerebbe un tuple centre programmato che elimina l'informazione vecchia e somma l'informazione localmente vicina, sia nel tempo che nello spazio.

TuCSoN utilizza i tuple centres come artefatti di coordinazione dove un tuple centre è un tuple space aumentato della nozione di comportamento programmabile. L'agente comunque percepisce il tuple centre come un normale tuple space, invece chi è responsabile della comunicazione può definirgli all'interno le leggi di coordinazione.

2.2.1 Spazio di coordinazione

Lo spazio di coordinazione di TuCSoN è realizzato da una molteplicità di artefatti di coordinazine chiamati tuple centres. Ogni tuple centres ha un proprio nome univoco sul nodo TuCSoN a cui è associato. Per riferirsi ad un certo tuple centre dovremo indicare:

<Tuple Centre Name> @ <IP Address>

Ad esempio:

```
resource @ localhost
```

2.2.2 Linguaggio di comunicazione

TuCSoN definisce anche delle primitive di comunicazione che l'agente deve inglobare per utilizzare l'artefatto. Gli agenti interagiscono scambiando tu-

ples attraverso il tuple centres utilizzando le primitive di comunicazione (out,in,rd,inp,rdp).

Il linguaggio di coordinazione di TuCSoN ha la seguente forma:

out	scrivi una tupla nel tuple centres.
in	invia una tupla template e aspetta la risposta dal tc con una tupla che unifica con il template e questa viene eliminata dal tc.
rd	spedisce una tupla template e aspetta la risposta dal tc con una tupla che unifica con il template.
inp	leggi e rimuovi se presente una tupla che unifica con il tupla template inviata altrimenti fallisci.
rdp	leggi se presente una tupla che unifica con il tupla template inviata altrimenti fallisci.
set_spec	setta il comportamento dello specifico tuple centre; inviando una valida teoria ReSpecT.
get_spec	preleva il corrente comportamento del tuple centre.

Tabella 2.1: Primitive di comunicazione

TCName @ TCAddress ? op (Tuple)

- TCName è il nome del tuple centre: può essere usata ogni striga con spazi per istanziare un correto tuple centre.
- TCAddress è l'indirizzo del tuple centre: un valido host name o un indirizzo IP.
- op è una delle operazioni elencate nella tabella 2.1.
- Tuple è l'informazione contenuta nell'operazione: deve esser una tupla logica valida (un termine Prolog). Esempio temperatura(50), temperatura(X), sender(ID,temperatura,Costo)

2.2.3 Utilizzo di TuCSoN da Java

I pacchetti fondamentali per accedere a TuCSoN da Java sono are alice.tucson.api e alice.logictuple. alice.tucson.api nel package sono incluse le classi che abilitano l'accesso all'infrastruttura TuCSoN e che rappresentano i tuple centres; alice.logictuple nel package sono incluse le classi che rappresentano il linguaggio di comunicazione: tuple logiche. Le classi principali che costituiscono il linguaggio di comunicazione sono:

- LogicTuple questa classe rappresenta le tuple logiche: è caratterizzata dal nome e da una lista di argomenti logici (TupleArgument).
- TupleArgument è una classe astratta che rappresenta la classe base per gli argomenti logici.
- Value questa è una clesse che deriva dal TupleArgument è che dà il valore agli argomenti logici.
- Var questa classe rappresenta una variabile logica, caratterizzata dal nome.

Esempio di creazione di un tuple centre chiamato 'resource' posizionato in localhost:

```
try {
   TucsonContext cn = Tucson.enterDefaultContext();
   TupleCentreId tid = new TupleCentreId("resource");
} catch (InvalidTupleCentreIdException e) {
   System.out.println("Errore nella creazione del tuple centre");
   e.printStackTrace();
```

Esempio di creazione di un tupla logica e di scrittura sul tuple centre:

```
LogicTuple t1;
TupleCentreId tid;
TucsonContext cn;
 ...
try {
   t1 = new LogicTuple("newCFP",
   new Value("cont",
```

```
   new Value("Initiator1"),
   new Value("temperatura")),
   new Value("prop",new Value(indice), new Value(temp)),
   new Value(temp));
   cn.out(tid,t1);
} catch (OperationNotAllowedException e) {
   e.printStackTrace();
} catch (UnreachableNodeException e) {
   e.printStackTrace();
}
```

2.2.4 Utilizzo di TuCSoN da tuProlog

Per utilizzare TuCSoN da tuProlog occorre che la libreria alice.tuprologx.lib.TucsonLibrary sia caricata nel motore inferenziale. Tutte le operazioni di comunicazione diventano disponibili come predicati buit-in nella forma:

TCName @ TCAddress ? op (Tuple)
TCName ? op (Tuple)
op (Tuple)

op è una delle seguenti operazioni (in,rd,out,inp,rdp,get_spec,set_spec) Esempio di agente prolog che utilizza TuCSoN:

```
:- load_library('alice.tuprologx.lib.TucsonLibrary').
main(Address):-
    write(Address),nl,
    text_from_file('coordinazione.rsp',Text),
    resource @ Address ? set_spec(Text),
    write('ok.').
```

Per lanciare l'agente tuProlog scriviamo la teoria in un file di testo test.pl ed eseguiamo con il seguente comando:

```
java -cp tucson.jar alice.tuprolog.Agent test.pl main(localhost).
```

2.3 ReSpecT

La fondamentale nozione di comportamento programmabile porta un tuple space a diventare un tuple centre e definirlo come artefatto di coordinazione. Si possono implementare nel tuple centres le leggi che regolano la coordinazione del sistema di agenti.

La percezione da parte degli agenti del tuple centre rimane uguale a quella dei tuple space, perchè l'agente interagisce con l'artefatto solamente leggendo e scrivendo tuple. E' chi programma l'artefatto che ne stabilisce le regole. Si potrebbe pensare che questa programmazione venga fatta da una entità superiore che crea le cose e ne descrive le regole di funzionamento.

Nel tuple centre programmabile si ha una transizione in un nuovo stato in risposta ad eventi di comunicazione, attraverso questa transizione noi cambiamo il comportamento del centro di tuple. Per ottenere un comportamento programmato occorre definire la reazione ad un evento di comunicazione attraverso un linguaggio di specifica delle reazioni.

Più precisamente ReSpecT Reaction Specification language crea la possibilità di associare ad un evento di comunicazione di TuCSoN (out, in, rd, inp, rdp) delle specifiche attività di computazione o anche delle azioni sull'artefatto chiamate reazioni. Ogni reazione può liberamente modificare i dati del tuple centre, e far scattare un evento di comunicazione che a sua volta può essere intercettato da un'altra reazione. Una specifica tupla ReSpecT è una clausola Prolog nella forma:

```
reaction(op,R)
```

dove op è il termine logico che rappresenta l'operazione e R è la reazione. Esempio:

```
reaction(out(play(X)),(
    in_r(count(Y)),
    Z is Y+1,
    in_r(sum(M)),
    V is M+X,
    out_r(sum(V)),
    out_r(count(Z))
)).
```

Le reazioni alle azioni compiute in seguito ad eventi di comunicazione vengono intercettate con un'altra sintassi: out_r, in_r, rd_r. Esistono per le in e rd due

predicati pre e post, che stanno ad indicare quando eseguire l'azione, se nella fase di domanda pre o nella fase di risposta dopo il tuple matching post.

Capitolo 3

Framework realizzativo

In questo capitolo presentiamo lo scenario di un MAS basato sulla nozione di artefatto di coordinazione con agenti logici che per pianificare il loro comportamento e servirsi dell'artefatto utilizzano le istruzioni operative che ne specificano il protocollo di utilizzo. Ispirandoci al mondo naturale dove la coordinazione è basata sull'ambiente come la stigmengia, esempio le formiche, ci poniamo l'obiettivo di avere una moltitudine di agenti che si coordinano fra loro attraverso l'ambiente e perseguendo il loro goal individuale anche con un comportamento emergente del sistema. Ad esempio una singola formica è un agente razionale che conosce ed esegue le istruzioni operative del proprio artefatto di coordinazione che è l'ambiente.

Il nostro obiettivo è quello di strutturare un agente logico in grado di eseguire le istruzioni operative collegate con la propria base di conoscenza. E di costruire OI che siano in grado di modellare l'interfaccia d'uso dell'artefatto di coordinazione.

3.1 Interpretazione delle istruzioni operative

Le istruzioni operative sono una serie di regole per l'agente sull'utilizzo dell'artefatto di coordinazione. Il modello di interazione dell'agente con l'artefatto risponde allo schema azione/percezione. Le OI generano un albero di possibili comportamenti formati dalla possibilità di compiere azioni o una percezioni.

Un agente può compiere delle azioni sull'artefatto in accordo con il

linguaggio di comunicazione dei tuple centre può leggere o scrivere tuple (in,out,inp,rdp) e percepirne eventualmente il risultato.

Utilizziamo il motore inferenziale Prolog per l'interpretazione della sequenza di istruzioni operative; la teoria logica si compone con una serie di regole che esprimono la semantica degli operatori delle istruzioni operative. Lanciando la dimostrazione, sarà il Prolog che attraverso il meccanismo di inferenziale, troverà la successiva sequenza di IO o la lista di azioni possibili. Tutto questo accade grazie alla grande potenza espressiva di un linguaggio logico.

Le unità base per la definizione del protocollo d'uso dell'artefatto sono l'azione, espressa dal predicato `act(X)` dove X rappresenta l'azione da compiere sull'artefatto, e la percezione `per(X)` dove X è la percezione in seguito ad una azione.

La variabile X può essere sostituita con qualsiasi tupla che corrisponda ad una valida azione o percezione definita dalla seguente sintassi in acccordo con il linguaggio di comunicazione del tuple centre:

```
validAction(out(_)).
validAction(in(_)).
validAction(rd(_)).

validPerception(out(_)).
validPerception(in(_)).
validPerception(rd(_)).
```

Quindi prima dell'esecuzione di una azione X occorrere controllare che si un'azione valida. validAction(X),act(X).

Nel nostro scenario abbiamo ridotto le azioni possibili sul tuple centre rispetto a quelle definite dal linguaggio di comunicazione cioè non consideriamo le azioni che potrebbero rispondere attraverso il fallimento come *rdp(_)* e *inp(_)*. I seguenti predicati completano la sintassi delle istruzioni operative con gli operatori di: sequenza, percezione, scelta, parallelo, ricorsione.

```
validState(zero).
validState(act(A,Pre)):-validAction(A).
validState(per(P,Eff)):-validPerception(P).
validState(seq(I1,I2)):-validState(I1),validState(I2).
validState(cho(I1,I2)):-validState(I1),validState(I2).
```

```
validState(par(I1,I2)):-validState(I1),validState(I2).
validState(agent1(Name,ListOfVar)).
```

Attraverso alla grammatica sopra illustrata riusciamo a definire in maniera corretta le istruzioni operative per l'utilizzo dell'artefatto. Gli operatori mes-

zero	Indica lo stato finale delle OI.
act(A,Pre)	A/azione deve essere una azione valida e Pre la sua precondizione verificata sulla base di conoscenza.
per(P,Eff)	P/percezione deve essere una percezione valida e Eff sono gli effetti sulla base di conoscenza.
seq(I1,I2)	Indica l'operatore di sequenza, dove I1 e I2 devono essere azioni o percezioni valide.
cho(I1,I2)	Indica l'operatore di scelta, dove I1 e I2 devono essere azioni o percezioni valide.
par(I1,I2)	Indica l'operatore di parallelo, dove I1 e I2 devono essere azioni o percezioni valide.
agent(Name, ListOfVar)	Name/nome del processo e ListOfVar/lista di variabili associate al processo. Dove il processo è una sequenza di stati validi (istruzioni operative).

Tabella 3.1: Grammatica delle OI

si a disposizione dal linguaggio per la costruzione di istruzioni operative ci permettono di esprimere una qualunque sequenza di azioni. L' operatore `seq(I1,I2)` ci permette di indicare la sequenzialità delle azioni cioè esprimiamo l'ordine di esecuzione delle istruzioni prima I1 e poi I2, mentre l'operatore `cho(I1,I2)` indica la possibilità di scelta tra due possibili azioni. Il parallelo tra due istruzioni è indicato dall'operatore `par(I1,I2)` e, dato che non gestiamo processi paralleli, l'operatore garantisce l'esecuzione in sequenza di entrambe le istruzioni senza un ordine non prefissato, la ricorsione è garantita attraverso l'operatore `agent1(Name,ListOfVar)` indica il passaggio all'esecuzione del processo *Name* con la lista di variabili *ListOfVar*.

Labeled Transition System LTS è la formalizzazione della semantica delle OI. In generale si tratta di una terna formata da un insieme degli stati S, dalle azioni possibili A e da una relazione di transizione $\rightarrow$:

$$\langle S, A, \rightarrow \rangle \, dove \rightarrow \subseteq S \times A \times S$$

nel nostro caso la LTS è rappresenta dalla tupla `transition(State,Action,NewState)` dove State è lo spazio numerabile formato da tutte le istruzioni operative, Action è l'insieme delle azioni possibili e NewState è il risultato della transizione.

Transition è un sottoinsieme dello spazio formato da $State \times Action \times State$ ed è quella relazione di transizione che verifica le regole operazionali, che definiscono la semantica degli operatori, espresse dalla teoria logica del predicato transition. La semantica dell'operatore di sequenza seq(X,I) è banalmente definita da:

$$seq(act(A), I) \xrightarrow{act(A)} I$$

cioè eseguendo l'azione X mi porta nello stato I. In Prolog esprimiamo la semantica dell'operatore seq attraverso i seguenti predicati inseriti nella nostra base di conoscenza:

```
transition(seq(act(A,Pre),I),act(A,Pre),I).
transition(seq(per(P,Eff),I),per(P,Eff),I).
transition(seq(act(A),I),act(A),I).
transition(seq(per(P),I),per(P),I).
```

La semantica degli operatori di scelta, di parallelo e di ricorsione sono espresse dai seguenti regole:

```
transition(cho(I1,_),A,I1B):- transition(I1,A,I1B).
transition(cho(_,I2),A,I2B):- transition(I2,A,I2B).

transition(par(I1,I2),A,par(I1B,I2)):- transition(I1,A,I1B).
transition(par(I1,I2),A,par(I1,I2B)):- transition(I2,A,I2B).

transition(agent1(Name,ListOfVariables),A,Q):-
    definitions(L),
    search(L,Name,ListOfVariables,P),
    transition(P,A,Q).

search([def(Name,ListOfVariables,P)|_],Name,ListOfVariables,P)
      :-!.
search([_|L],Name,ListOfVariables,P):-
                search(L,Name,ListOfVariables,P)
```

La trasizione di ricorsione presuppone che le istruzioni operative vengano rilasciate come una sequenza di processi formati da un nome, una lista di variabili e da un insieme di istruzioni operative contenute all'interno del predicato def(Name,ListOfVariable,P). Il meccanismo di transizione dell'agent1 ricerca nella definizione generale il processo che unifica con il nome, che si suppone univoco, e con la lista di variabili. Tale processo diventerà il nuovo processo in esecuzione.

Occorre sottolineare che le transizioni avvengono con il passaggio del solo processo corrente, al termine del quale il collegamento tra le variabili si scioglie a meno che non si utilizzi il passaggio di parametri, questo evita pesanti problemi di unificazione delle variabili visto che le OI si presentano come unica stringa. Otteniamo l'esecuzione della transizione eseguendo il *goal* transition(State, A,NewState) se l'esecuzione avrà successo legato al valore della variabile newState avrò la successiva sequenza di istruzioni operative.

In Prolog abbiamo espresso in maniera molto chiara la semantica delle operatori. Ma per completare l'interpretazione delle istruzioni operative è necessario che dato uno stato valido ci venga restituito l'insieme delle azioni attualmente possibili. In questo ci viene in aiuto il Prolog con un predicato che risolve una query del secondo ordine findAll(Template,Goal, List) cioè ci restituisce l'insieme degli elementi Template che soddisfano il Goal.

La dimostrazione del seguente predicato findall(Azione,transition(Processo,Azione,_),Lista) ci restituisce la lista delle azioni possibili, il processo è una variabilie di input che va specificata con le istruzioni operative correnti, l'Azione è una variabile libera come pure lo stato futuro mentre la Lista è una variabile di uscita che conterrà se il predicato avrà successo la lista delle azioni possibili. Il motore inferenziale Prolog inferirà su tutte le transizioni trovando quali sono le azioni che verificano e quindi ammissibili.

Con la giusta combinazione dei predicati findall() e transition() si è creato un interprete di istruzioni operative in grado di guidare l'interazione con l'artefatto di coordinazione. La sequenza di istruzioni principali per l'interpretazione delle OI e l'interagire con l'artefatto è dato dal seguente codice:

```
main(P):- findall(A,transition(P,A,_),L),
<esegui una azione della lista L-A1>,
transition(P,A1,P1),
main(P1).
```

Rimane in sospeso la scelta dell'azione da eseguire nel caso ci fossero una lista con più di un elemento di azioni possibili. Questa scelta è legata allo stato mentale dell'agente, cioè alla sua base di conoscenza.

Ci viene in aiuto per delimitare le possibili azioni il meccanismo di precondizioni ed effetti che collegano la nostra interazione con il nostro stato mentale. Una azione possibile viene abilitata solo se verifica la precondizione, che è dimostrata sulla base di conoscenza, in generale la scelta viene fatta attraverso un meccanismo di IA.

Le IO vengono fornite in forma di una unica stringa e inserite come fatto nell'agente. Esempio di istruzioni operative:

```
firststate( seq(
                act(out(p(Id,Phi))),
                seq(
                    per(out(p(Id,Phi))),
                    seq(
                        act(in(q(Id,Phi,_))),
                        cho(
                            seq(per(in(q(Id,Phi,yes))),zero),
                            seq(per(in(q(Id,Phi,no))),zero)
                        )
                    )
                )
)).
```

Queste istruzioni mostrano una semplice sequenza di richiesta di verità di un'informazione Phi `out(p(Id,Phi))` e di sucessiva attesa di una risposta `in(q(Id,Phi,_)` che potrà avere un esito negativo o positivo.

3.2 Agenti Logici

Gli agenti logici si basano su una qualche forma di logica, (vedi 1.2.2). Nel nostro scenario costruiamo agenti basati su tuProlog quindi in grado sia di utilizzare il motore inferenziale Prolog che le api Java come spiegato nella sezione 2.1. L'agente logico nel nostro scenario fa parte di una società di agenti che utilizza una forma di comunicazione mediata, attraverso attraverso degli artefatti di coordinazione. La comunicazione si basa sulla scrittura

e lettura di tuple in un tuple centre, che consente una serie di azioni che sono espresse con un linguaggio di comunicazione (in, out, inp, rdp, set_spec, get_spec) 2.2.2. L'artefatto di coordinazione realizzato attraveso un tuple centre, è la struttura che la società di agenti usa per coordinarsi. L'artefatto ingloba in sè le regole della coordinazione ed associato ha la nozione di interfaccia d'uso cioè di come l'agente può utilizzare l'artefatto(vedi 1.5).

Ispirandoci al mondo naturale dove esistono molteplici esempi di sistemi che utilizzano questo tipo di coordinazione come le formiche il nostro obiettivo è di realizzare un agente razionale in grado di interagire con l'artefatto/ambiente. La logica ci garantisce un modello di ragionamento razionale che utilizziamo per ottenere l'azione da compiere in accordo con la base di conoscenza attraveso una operazione di inferenza (vedi 1.4).

Le OI come già viste in precedenza nella sezione 1.5 e l'interprete di istruzioni operative sono lo strumento che l'agente possiede per interagire con l'artefatto e compongono una parte dell'agente logico. In una visione BDI dell'agente le OI rappresentano le intenzioni dell'agente cioè quali saranno le prossime azioni da compiere. Ma la decisione dell'azione deve avvenire anche considerando la base di conoscenza dell'agente non solo lo stato in cui si trova l'OI, occorre collegare le OI con lo stato mentale dell'agente. Questo collegamento viene dalla meccanismo di precondizione effetto. Ad ogni azione sull'artefatto è collegata una precondizione che deve essere verificata prima dell'esecuzione dell'azione, abilitando o meno l'azione stessa, o anche valorizzando delle variabili presenti nell'azione e nella precondizione. Gli effetti sono una conseguenza delle percezioni e vanno ad inferire nuova conoscenza sui believe dell'agente modificando predicati già presenti o aggiungendone di nuovi.

Si configura un agente logico composto da: una base di conoscenza, delle precondizioni, degli effetti, un iterprete di istruzioni operative, una definizione delle IO e un main loop che esegue la principale sequenza di azioni dove il goal dell'agente è l'esecuzione di IO. Un esempio di ciclo principale:

```
main_loop(S) :-  <cerca lista azioni in S>
<scelta azione act(A,Pre)>
<esegui precondizione Pre>
<esegui azione A>
<ricevi/crea percezione P>
<transizione (S,A,S1)>
<transizione (S1,per(P,Eff),S2)>
```

```
<esegui effetto Eff>
main_loop(S2)
```

Le precondizioni e gli effetti non sono solo dei predicati che vanno a verificare dei semplici fatti ma possono andare a verificare vere e procedure che agiscono sulla base di conoscenza. Un elemento cruciale dell'ingegnerizzazione del sistema è la progettazione della base di conoscenza: oltre ai belives che l'agente possiede che ne esprimono la conoscenza, l'agente logico che interagisce con gli artefatti deve possedere delle regole nella base di conoscenza di collegamento con le OI cioè delle procedure che vengono invocate al momento di verifica di precondizioni ed effetti. Per questo problema non si ha una soluzione univoca ma occorre valutare il goal dell'agente, la struttura della sua base di conoscenza e costruire queste procedure di collegamento. La struttura della base di conoscenza si presenta composta dalla sintassi e dalla semantica dell'OI, dai believes, dalle procedure di collegamento delle precondizioni e degli effetti. Per funzionare l'agente deve possedere tutti questi componenti che per loro natura hanno caratteristiche diverse: l'iterprete delle OI è uguale per tutti i componenti del sistema, mentre i believes, le OI, le procedure di precondizione ed effetti potrebbero risultare diverse da agente ad agente del sistema.

Ripensando ai sistemi naturali come le formiche dove tutte le entità hanno il medesimo comportamento razionale e si rapportano alla stessa maniera con l'ambiente potremmo immaginare che tutte queste entità abbiano le stesse istruzioni operative, gli stessi believes e le stesse precondizioni ed effetti. La conseguenza di questa osservazione è che il nostro una modello di agente è 'like ant' che se valorizzato nella sua base di conoscenza e replicato costituisce un sistema del tutto simile a quelli naturali. Con facili modifiche all'artefatto di coordinazione e alle OI si riescono a modellare tantissime forme di comportamento e cosi è possibile studiare la relazione delle semplici regole che guidano il comportamento degli agenti e le proprietà emergenti presenti al macro livello del sistema.

Ci si pone l'obiettivo di studiare i pattern di comportamento emergenti dovuti all'interazione tra molti individui e capire quando le entità sono in grado di formare uno stato regolare e sotto quali circostanze questo stato è stabile. L'agente logico cosi strutturato più l'artefatto di coordinazione fornisce una l'infrastruttura molto valida per lo studio della dinamica dei sistemi con l'interazione di molti individui.

Capitolo 4

Minority Game

Il Minority Game rappresenta un semplice modello di sistema dove gli agenti in modo simultaneo e adattativo competono per limitate risorse. Il sistema evolve attraverso l'interazione di agenti che costruiscono scelte su un ambiente che agisce per coordinarne il comportamento. Tali agenti hanno strategie eterogenee, beliefs e uno scopo da realizzare. Un agente che partecipa al MG può essere implemetato attraverso un Agente Logico 1.4 e l'ambiente su cui gioca e si coordina può essere implementato come tuple centre 2. La costruzione di agenti reali che giocano il MG, mette in evidenza i principali problemi di coodinazione società di agenti e le principali strutture con cui si risolvono [12].

4.1 Il problema del bar di 'El Farol'

Il problema del bar di El Farol, da cui ha origine il Minority Game, è proposto nella seguente maniera: ci sono N persone indipendenti che ogni ogni week end decidono se andare al bar ad ascoltare della musica oppure no. Lo spazio del bar è limitato e si riesce ad ascoltare con piacere della musica se il numero di persone presenti è minore di aN con $a \leq 1$, dove N è il numero massimo di persone. Una persona va nel locale solo se una persona si aspetta di trovarvi nel locale un numero di persone inferiore a aN [13]. Ma pochè l'unica informazione disponibile è il numero di persone che sono andate al locale nel week end precedente è impossibile dare una deduttiva razionale soluzione al problema anche allargando lo spazio dell'informazione disponibile nel passato.

4.2 Il Minority Game

Ispirandosi al problema 'El Farol' Challet and Zhang [14]ne ha dato una precisa definizione matematica chiamata Minority Game (MG). Il modello di sistema è formulato per una società di agenti evolutiva e adattativa. Si tratta di un toy model, dove gli agenti applicano un pensiero induttivo piuttosto che deduttivo e dove la società di agenti con una limitata razionalità genera un comportamento cooperativo. Il gioco consiste in N(dispari) agenti che decidono ad ogni time step di giocare una certa azione $a_i(t)$ con $i = 1, .., N$ che esprime una certa scelta binaria. Ad esempio se rappresentassimo la scelta di andare al bar con $a_i(T) = 1$ e la scelta di stare a casa con $a_i(t) = -1$. L'agente che ha scelto l'azione minoritaria vince mentre la maggioranza perde dopo ogni turno di gioco il totale delle azioni è dato da:

$$A(t) = \sum_{i}^{N} a_i(t)$$

Ad ogni agente viene dato un premio: $-a_i(t)g[A(t)]$ con $g(x) = sign(X)$ dalla quale però non dipende in maniera significativa l'andamento del gioco. L'informazione sul gruppo vincente viene data agli agenti attraverso la funzione $W(t+1) = sign(A(t))$. La scelta dell'azione $a_i(t)$ successiva avviene attraverso un ragionamento induttivo su una conoscenza parziale del mondo data dalla memoria degli m passati risultati. Ad ogni agente è associato un set di strategie $s \geq 2$, dove una strategie mappa tutte le possibili sequenze dei passati risultati con la futura azione possibile. La dimensione della strategia dipende da m, ci sono 2^m possibili valori a cui deve essere associata la predizione. Lo spazio delle strategie ha dimensione 2^{2^m} ed indica il numero di strategie possibili. Nel caso di $m = 3$ una strategia possibile è ad esempio $(+1, +1, -1, +1, +1, +1, +1, -1)$. L'azione predetta dalla strategia è in relazione con i passati gruppi vincenti espressi in una rappresentazione binaria associando al -1 uno 0. $\mu(t) \in (1..2^m)$ è il numero decimale convertito dalla rappresentazione binaria degli m gruppi vincenti passati e rappresenta l'elemento da selezionare, nel vettore della strategia, come prossima azione da eseguire. Il sistema presenta caratteristiche adattative perchè tra le possibili s strategie scegliamo quella che ha avuto un punteggio virtuale più alto. Il punteggio è calcolato alla fine di ogni turno, quando si conosce il gruppo vincente, assegnando un punto ad ogni strategia con la predizione corretta. Il nuovo stato della memoria andrà aggiornato con il risultato del turno di

gioco e si otterà un valore $\mu(t+1)$ dato da:

$$\mu(t+1) = [2\mu(t) + (W(t+1) - 1)/2] mod P$$

I paramentri fondamentali del sistema sono: s numero di strategie, m che è la dimensione della memoria, N numero di agenti e *step* il numero di giocate. Queste variabili rappresentano il dominio del sistema. Ogni agente inizia il gioco generandosi casualmente un insieme di strategie s e uno stato random di memoria m, poi inizia a giocare e percepire i risultati.

Per capire l'andamento che dovremo aspettarci consideriamo i casi estremi del gioco: il primo caso è rappresentato da un solo giocatore che gioca su un lato e tutti gli altri giocano sul lato opposto; il giocatore è molto fortunato e prende un punto mentre tutti gli altri prederanno zero. L'altro caso estremo è che in un lato giochino $(N-1)/2$ giocatori e nell'altro giochino $N + 1/2$ giocatori. In questo caso la metà della popolazione prende un punto ed è il caso preferibile dal punto di vista della società, perchè rappresenta la perfetta coordinazione mentre la precedente situazione esprime il fallimento totale del sistema. Ci aspettiamo che la nostra popolazione si muova con una variabilità tra i due estremi.

Il valore di $A(t)$ varia attorno al valore di ottimalità che è zero e presenta nel casi di m piccolo un comportamento periodico che tende a svanire all'aumentare di m. La media su un lungo periodo risulta essere proprio $\overline{A(t)} = 0$ per valori di $m > 1$ ed $N >> 1$ [14].

4.3 L'efficienza del sistema

La varianza σ^2 è la principale grandezza per la misura dell'efficienza del sistema ed è definita da: $\sigma^2 = \overline{[A(t) - \overline{A(t)}]^2}$; essa indica la variabilità delle giocate attorno al valor medio $\overline{A(t)} = 0$, quindi maggiore è la varianza peggiore è il comportamento di coordinazione del sistema. Un'alta varianza indica che non stiamo utilizzando in maniera efficiente la risorsa. Il comportamento di σ^2 è una funzione dei parametri m, s e N. Si è verificato da estensive simulazioni che la varianza normalizzata al numero di agenti $\rho = \sigma^2/N$ dipende unicamente dal valore $\alpha = 2^m/N$ per ogni valore di s. Per grandi valori di α il ρ tende al valore di scelta random $\rho = 1$ cioè con agenti che eseguono la loro scelta casualmente e con uguale probabilità. Per piccoli valori di α il ρ tende ad essere molto alto esprimendo una pessima coordinaizone tra

gli agenti del sistema, mentre per valori intermedi di $\alpha \cong 1/2$ la varianza è inferiore rispetto al caso random. In questa regione il gruppo degli agenti che perdono è di dimensione $N/2$ che è la minima dimensione possibile.

Tale comportamento ci mostra che un aumento progressivo di m cioè della storia passata non produce come ci potevamo aspettare un costante miglioramento della coordinazione sistema, ma ad un certo punto conduce ad un peggioramento, ciò accade perchè in questo tipo di struttura allargando lo spazio delle strategie oltre un certo limite le scelte diventano molto vicine al random.

4.4 Irrilevanza della memoria

Studi effettuati dimostrano come il risultato del gioco possa essere sostituita da una sequenza random senza alterare il comportamento del sistema. La storia passata è sostituita da una sequenza di m bit, che prende il ruolo di una falsa memoria. Questa è l'informazione che tutti gli agenti utilizzano con le loro migliori strategie per la scelta da giocare. Tale modello del gioco produce gli stessi risultati dell'originale, dimostrando che il ruolo di m è puramente geometrico [15]. Da un punto di vista tecnico il parametro m non ha significato di memoria, ma indica la dimensione dello spazio delle strategie. Lo studio della geometria del sistema e della distribuzione delle strategie nel loro spazio sono elementi chiave per la comprensione corretta del problema [15]. La proprietà principale del MG non è la memoria della storia passata, ma piuttosto il fatto di condividere tutti la stessa informazione.

4.5 Il comportamento adattativo ed evolutivo

I giocatori del MG, costruiti con la strategia sopra citata, presentano un comportamento adattativo, perchè il giocatore utilizza sempre la strategia che ha il punteggio più alto. Nel tempo l'agente sceglie le strategie con cui giocare che meglio si adattano all'andamento del sistema. Si può affrontare la soluzione del MG con un approccio genetico, eliminando regolarmente gli agenti con uno scarso punteggio e introducendone di nuovi che rimpiazzino quelli eliminati. Si è applicato così il principio evolutivo di Darwin: i peggiori

giocatori vengono eliminati dopo un certo numero di interazioni e rimpiazzati con dei nuovi giocatori clonati dai migliori, cioè caricandogli le stesse strategie ma azzerando i punteggi virtuali. Tale evoluzione è analoga all'evoluzione umana nel senso che un bambino che nasce eredita le stesse potenzialità, dei genitori ma non ha la stessa conoscenza.

Otteniamo una certa diversità se introduciamo la mutazione come possibilità di clonazione. Una delle strategie del miglior giocatore viene rimpiazzata da una nuova disponibile nello spazio delle strategie. Ci aspettiamo che il sistema in questo modo sia capace di apprendere, auto distruggendo i peggiori giocatori e replicando i migliori. Ne risulta che l'apprendimento emerge nel tempo; le fluttuazioni si riducono e si saturano ma non si raggiunge il comportamento ottimo del sistema.

4.6 Resource Allocation Games

Si può considerare il MG come un punto nello spazio del dominio dei Resource Allocation Game (RAG) studiato nell'articolo [16]. Possiamo considerare MG come N agenti che competono ognuno per una unità di risorsa fornita da due fornitori. Ogni fornitore ha disponibile $N - 1/2$ di una certa risorsa e un agente deve scegliere da quale fornitore andare. Se un fornitore viene sovraccaricato non riuscirà a soddisfare le richieste. Una naturale estensione del gioco è di disaccopiare i fornitori di risorse disponibili dal numero degli agenti, avremo quindi N agenti, G fornitori che avranno a disposizione una certa quantità di risorsa C/G. Il minority game diventa un punto nel domino dei RAG rappresentato dalla configurazione $N, G = 2, C = N - 1$. Successivi esperimenti su tale dominio hanno evidenziato una transizione di fase, il sistema cambia stato di equilibrio velocemente, muovendosi lungo il parametro N da un valore $N < C$ ad un valore $N > C$. In una rappresentazione tridimensionale dove sull'asse delle x mettiamo gli agenti sull'asse delle y le risorse disponibili e la sull'asse z la varianza del sistema si osserva che lungo la diagonale $N = C$ si ha un passaggio veloce da uno stato di sistema con bassa varianza con una allocazione di risorse efficiente ad uno con alta varianza.

E' molto importante studiare sistemi dove il che hanno una variabilità tra domanda e offerta molto vicina alla configurazione del MG. In generale ci aspettiamo che molti sistemi nel mondo reale evolveranno attraverso uno stato

in cui le risorse saranno bilanciate con le domande. Per esempio ad un' alta domanda di personal computer le compagnie risponderanno aumenteranno la loro produzione. Quando la domanda diminuisce, le compagnie andranno fuori dal buisness e sopravivveranno solo quelle capaci di ridurre la produzione, guidando il sistema in uno stato in cui la fornitura sia rapportata alla domanda. Nel MG la domanda e l'offerta sono bloccate obbligando il sistema ad una competizione per limitate risorse. Nella realtà questo bilanciamento non è imposto ma generalmente si ottiene dinamicamente.

Capitolo 5

Realizzazione del Minority Game

La realizzazione del gioco è stata ottenuta utilizzando il framework realizzativo, e le tecnologie tuProlog e TuCSoN. La struttura del gioco è formata da un agente esecutore di OI, introdotto nella prima sezione, da un tuple centre programmato in ReSpecT introdotto nella seconda sezione, nella terza si esamina il comportameto dell'agente dovuto alla propria gestione delle strategie, nella quarta sezione si verifica la possibilità di spostare il sistema in un altro stato di equilibrio attraverso il tuning di alcuni parametri. Nella quinta sezione descriviamo come lanciare il gioco e nella sesta sezione mostriamo i grafici ottenuti di varianza e andamento del gioco.

5.1 Le istruzioni operative dei giocatori

L'agente, seguendo il modello precedentemente dettagliato nella sezione 3 inizia l'esecuzione delle istruzioni operative a partire dal processo di partenza indicato nella teoria logica: `firststate(agent1(primo,[]))`. La definizione delle istruzioni operative rappresenta il cuore dell'agente, ne determina il comportamento. Il primo processo da cui parte l'esecuzione è `agent1(primo,[])` che esegue la lettura dei parametri del minority game e li memorizza attraverso l'effetto nella base di conoscenza. La percezione e gli effetti eseguiti dal primo processo in esecuzione sono:

```
pre(firstAction(S)):-write('eseguo first'),firstAction(S).
```

```
eff(formulaTrue(Phi)):-retract(Phi), assert(Phi).
```

Definizione del `primo` processo:

```
def(primo,[],
  seq(act(rd(mem(M))),
  seq( per(rd(mem(M)),
    eff(formulaTrue(mem(M)))),
  seq(act(rd(spazStr(M1))),
  seq( per(rd(spazStr(M1)),
    eff(formulaTrue(spazStr(M1)))),
  seq(act(rd(numStr(Num))),
  seq( per(rd(numStr(Num)),
    eff(formulaTrue(numStr(Num)))),
  seq(act(rd(numsession(Num1))),
  seq( per(rd(numsession(Num1)),
    eff(formulaTrue(numsession(Num1)))),
  seq(act(out(ok(config))),
  seq(per(out(ok(config)),pre(firstAction(S))),
    agent1(ciclo,[S])
))))))))))),
```

La precondizione pre(firstAction(S)) si occupa dell'inizializzazione dell'agente cioè della creazione del proprio insieme di strategie, sul quale si deciderà l'azione da compiere e la generazione del proprio stato iniziale. Finita la fase di inizializzazione si passa all'esecuzione del processo `agent1(ciclo,[S])` dove si esegue la giocata mettendo una tupla `play(X)` con $X = \pm 1$. La precondizione dà valore alla variabile X prima dell'esecuzione dell'azione, chiedendo il valore della giocata alla base di conoscenza rappresenta dalla strategia S passata come parametro. Poi l'esecuzione passa al `agent1(ciclo2,[S])` dove si aspetta la tupla `winner(Ris,NG,CS,last)` dall'artefatto di coordinazione, che contiene il risultato del gioco. I parametri NG e CS sono due contatori uno di sessione e uno di gioco per evitare che l'agente legga tuple legate a giochi precedenti. Se nella tupla letta è presente `last` significa che la corrente è l'ultima giocata e l'agente transiterà in uno stato di fine `agent1(fine,[S,CS])`. In questo stato l'agente resta in attesa di una ulteriore conferma per la chiusura o nel caso opposto transita alla reinizializzazione dell'agente per una ulteriore sessione di gioco. Se l'artefatto viene letto `more` allora si realizza il ciclo perchè dopo aver aggiornato

lo stato con il risultato del gioco attraverso l'effetto si passa a `ciclo` per eseguire una nuova giocata.

```
firststate(agent1(primo,[])).
definitions([
def(primo,[],(...),
def(ciclo,[S],
       seq(act(out(play(AzF)),pre(scegli(S,AzF))),
         seq( per(out(play(Sc)),eff(s)),
           agent1(ciclo2,[S])
    ))),

def(ciclo2,[S],
    seq(
        act(rd(winner(X,NG,CS,L)),pre(agg(NG,CS))),
        cho(
            seq(per(rd(winner(Ris,NG,CS,last)),
            eff(s)),agent1(fine,[S,CS])),
            seq(per(rd(winner(Ris,NG,CS,more)),
            eff(nuovoStato(Ris,S,NG))),agent1(ciclo,[S]))
    ))),
def(fine,[S,CS],
    seq(
        act(rd(continue(X))),
        cho(
            seq(per(rd(continue(yes)),eff(azzera,CS)),
            agent1(primo,[])),
            seq(per(rd(continue(no)),eff(s)),zero)
    )))
]).
```

L'agente continua a giocare fino a quando l'artefatto decide di fermarlo.

5.2 La programmazione ReSpecT dell'artefatto

Il tuple centre è programmato inviando una teoria ReSpecT valida attraverso la primitiva di comunicazioe `set_spec()`. La teoria viene inviata dall'agente `resource.pl` al momento della configurazione dell'artefatto. La programmazione ReSpecT è dettagliata nella sezione 2.3.

La reazione principale che genera tutte le altre a catena avviene al momento dell'arrivo delle tuple di gioco `play()` inviate dai giocatori.

```
reaction(out(play(X)),(
  in_r(count(Y)),
  Z is Y+1,
  in_r(sum(M)),
  V is M+X,
  out_r(sum(V)),
  out_r(count(Z))
)).
```

Vengono contate le giocate attraverso la lettura e scrittura della tupla `count()` incrementata di uno. Si può determinare anche il risultato parziale, sommando le giocate: ± 1 degli agenti e memorizzandole nella tupla `sum()`.

Intercettando la scrittura di `count()` e confrontandola con il numero di agenti, che è un parametro di configurazione del sistema `numag()`, determiniamo se il gioco è finito e se può produrre la nuova tupla di risultato `winner()`. Se il confronto tra il numero di agenti e il numero di giocate ha successo, si incrementa il conteggio dei turni di gioco `totcount()`, si azzera il risultato del gioco contenuto in `sum()` e il contatore delle giocate `count()`. Inoltre si mantengono aggiornati la somma totale dei risultati `totsum()` e la somma totale dei risultati al quadrato `qsum()` per il calcolo della media e della varianza. La reazione al `out(count(X))` è la seguente:

```
reaction(out_r(count(X)),(
  rd_r(numag(Num)),
  X=:=Num,
  in_r(totcount(T)),
  P is T+1,
```

```
  rd_r(game(G)),
  in_r(sum(A)),
  out_r(sum(0)),
  rd_r(countsession(CS)),
  in_r(count(Y)),
  out_r(count(0)),
  %%per il calcolo della varianza
  in_r(qsum(SQ)),
  NSQ is A*A+SQ,
  out_r(qsum(NSQ)),
  %%per il calcolo della media
  in_r(totsum(R)),
  NewS is R+A,
  out_r(totsum(NewS)),
  rd_r(tuning1(T1)),
  rd_r(tuning2(T2)),
  out_r(winner(A,P,CS,T1,T2,G)),
  out_r(totcount(P))
)).
```

Lo scopo degli agenti è quello di giocare e attendere il risultato, la terminazione del gioco o della sessione è un compito di coordinazione dall'artefatto. Il numero di turni e il numero di sessioni sono un parametro di sistema noto solo all'artefatto. Per realizzare il comportamento di coordinazione di fine gioco, l'artefatto conta il numero di turni e sessioni fino ad arrivare al valore di configurazione e scrivere le tuple `game(last)` per i turni e `continue(no)` per le sessioni. Leggendo la tupla `game(G)` dove `G/more;last` e inserendo il parametro nella tupla `winner(_,_,_,_,G)` comunichiamo all'agente la fine del turno. L'agente andrà in seguito a leggere il valore `continue(yes/no)` e saprà se riiniziare il gioco o finire. La fine gioco può anche essere ottenuta attraverso la console di comando del gestore della simulazione pulsante 'Stop'. La tupla principale per la coordinazione è la tupla: `winner(Ris,numturno,numsessione,tuning1,tuning2,last/more)` Gli agenti internamente conservano la traccia del numero di turno e di sessione in modo da sapere quale nuovo messagio leggere per non avere una informazione sbagliata. I parametri di tuning vengono aggiornati ogni volta che si invia il messaggio. Il codice completo ReSpecT è il seguente:

```
%% Il compito del tuple center è sancire la vittoria del
%% gruppo di minoranza con il messaggio
%% winner(Risultato,NG,CS,T1,T2,more/last).
reaction(out(play(X)),(
  ...
)).
reaction(out_r(count(X)),(
  ...
)).

reaction(out_r(totcount(P)),(
  rd_r(numgame(N)),
  N1 is N-1,
  P=:=N1,
  in_r(game(G)),
  out_r(game(last))
)).

reaction(out_r(totcount(P)),(
  rd_r(numgame(N)),
  P=:=N,
  in_r(countsession(CS)),
  CS1 is CS+1,
  out_r(countsession(CS1))
)).

reaction(out_r(countsession(C)),(
  rd_r(numsession(NS)),
  NS=:=C,
  in_r(continue(X)),
  out_r(continue(no))
)).

reaction(out_r(countsession(C)),(
  rd_r(numsession(NS)),
  NS=\=C,
  in_r(continue(X)),
```

```
  out_r(continue(yes))
)).

reaction(out_r(winner(X1,Y1,_,_,last)),(
  rd_r(numgame(NG)),
  in_r(totsum(R)),
  rd_r(mem(Mem)),
  Media is R/(NG+1),
  out_r(media(Mem,Media)),
  in_r(qsum(QS)),
  Varianza is QS/(NG+1)-Media*Media,
  out_r(varianza(Mem,Varianza)),
  out_r(totsum(0)),
  out_r(qsum(0)),
  in_r(game(G)),
  out_r(game(more)),
  in_r(totcount(TC)),
  out_r(totcount(-1))
)).
```

Intercettando l'ultimo `winner(_,_,_,_,last)` chiudiamo il turno di gioco calcolando media e varianza dei risultati di gioco e reinizializiamo alcuni parametri pronti per ricominciare a giocare.

5.3 La gestione delle strategie

Le strategie dell'agente sono implementate come array di bit, dove l'entry per ogni elemento è data dallo stato passato. La lunghezza della strategia è data dalla dimensione della memoria: se $m = 3, 5, 7, 15$ la dimensione dell'array $dim = 2^m$ sarà: $dim = 8, 32, 128, 32768$. Ogni agente ha un numero s di strategie, dove s è parametro di sistema, prese a caso dall'insieme totale delle strategie. Ad ogni strategia è assegnato un punteggio e viene utilizzate per giocare la strategia con il punteggio più alto. Il punteggio di una strategia, in un determinato time step, è incrementato di uno se ha predetto correttamente il risultato del gioco. Abbiamo impletmentato tutte queste funzionalità di gestione delle strategie dell'agente in una classe Java `Strategie` richiamata

dall'agente tuProlog. Lo stato interno della classe è rappresentato da un array di BitSet che rappresenta le strategie dell'agente che non possono essere modificate.

Il costruttore della classe è: `public Strategie (int numS,int mem)` vuole in ingresso il numero di strategie e la dimensione della memoria; crea un insisme `numS` di strategie di lunghezza 2^{mem} generate random. I metodi pubblici sono: `public void configura(int mem)` si usa per generare un nuovo spazio delle strategie di dimensione 2^{mem}; `public void calcPunteggio(int ris, int stato)` si usa per aggiornare il punteggio delle strategie prendendo in ingresso lo stato del sistema rappresetato dalla memoria passata convertita in decimale e il risultato del gioco; `public int getValore (int n)` restituisce l'elemento n-esimo della strategia di punteggio massimo; `public void stampaPunt()` stampa i risultati.

L'agente al momento della configurazione esegue come effetto di fine lettura dei parametri la creazione delle strategie e della lista che rappresenta la memoria del sistema richiamando la procedura:

```
firstAction(S):-
    mem(M),genera(St,M),
    numStr(N),
    java_object('tuning.Strategie',[N,M],S),
    S <- stampaPunt.
```

La determinazione della scelta avviene attraverso la precondizione all'azione di `out(play(X))` valorizzando il valore della variabile X. La procedura eseguita è:

```
scegli(S,NextAction):-
    stat(NS),stato(NS,I),
    S <- stampaPunt,
    S <- getValore(I) returns A,
    (A>0->NextAction = 1; A=<0->NextAction = -1).
```

Invocando il metodo `S <- getValore(I) returns A` si ottiene il valore da giocare determinato dalla migliore strategia. Come effetto della percezione della tupla `winner()` viene aggiornato lo stato della memoria, inserendo il risultato del gioco in testa alla lista e eliminando l'ultimo elemento. Vengono aggiornati i punteggi con `S <- calcPunteggio(Ris,I)`, viene tenuto traccia

del numero consecutivo di sconfitte e verificato con i parametri di tuning se sia necessaria o meno una rigenerazione delle strategie `S <-configura(T2)`. La procedura seguente ne è l'implementazione:

```
nuovoStato(Ris,S,T1,T2):-
        (Ris>0->Ris1 = 0;Ris=<0->Ris1=1),
        ...
        stat(St),
        nextAction(Ris1,St,NS),
        retract(stat(St)),assert(stat(NS)),
        %%Aggiorno i punteggi, NS Nuovo Stato
        stato(St2,I),
        S <- calcPunteggio(Ris,I),
        S <- stampaPunt.
```

5.4 Tuning

Il programma di simulazione permette di simulare gli andamenti di giocate del minority game, dove la variazione dei parametri generali del gioco può avvenire online, ossia modificando le regole di coordinazione del MAS e non cambiando il singolo behaviour degli agenti. Riuscire a portare un sistema da uno stato di equilibrio all'altro in maniera on line, senza doverlo fermare, riprogrammare gli agenti poi farlo ripartive è uno degli obiettivi che può essere ottenuto, utilizzando una architettura basata su TuCSoN.

Il MG giocato con un numero non molto alto di partecipanti dopo poche mosse si stabilizza in un comportamento periodico di stretto periodo proporzionato al numero di agenti. L'ampiezza di queste oscillazioni determina la qualità della coordinazione del sistema: più si discostano dal valore atteso *zero* e maggiore sarà la varianza. Una volta che il sistema è entrato nel regime periodico, si è stabilizzato e continuerà tale comportamento all'infinito; si tratta di una situazione ottimale per piccole oscillazioni dell'andamento del gioco attorno al valore zero, mentre in caso contrario avremo una pessima utilizzazione delle risorse del sistema. Il sistema nel caso di pochi agenti presenta anche dei comportamenti degeneri dove tutti eseguono la stessa azione.

Vogliamo ottenere una regolazione on line svolta ad esempio o da un agente manager o da un utente umano oppure programmata direttamente

sull'artefatto di coordinazione mediante una teoria ReSpecT 2.3, che porti il sistema ad evitare situazioni degeneri e di scarso utilizzo delle risorse, e lo porti ad uno stato di maggiore efficienza. Appenna si verifica l'evento di alta varianza, $\sigma^2/N > 1$ dove uno è il valore di varianza ottenuto da un agente che si comporta in modo random o una situazione degenere, interveniamo sui parametri di tuning per guidare il sistema. Si sono individuati due parametri che possono determinare un cambio di equilibrio del sistema: uno è un parametro standard del minority game, che è la dimensione dello spazio delle strategie chiamato anche memoria del sistema, perchè indica quante azioni passate devo essere considerare per determinare la scelta corrente vedi 4, l'altro è il parametro soglia che indica il numero di eventi di perdita tollerati dagli agenti prima di rigenerare la propria strategia.

5.4.1 Parametro di Tuning: soglia delle perdite

Il parametro di soglia delle perdite è inserito all'interno dell'agente ed indica il numero di volte consecutive che può verificarsi l'evento di perdita, superato il quale, l'agente rigenera la propria strategia in modo da rompere la simmetria creatasi nel sistema ed evolvere così in un nuovo stato di equilibrio migliore del precedente. Tale parametro non è presente nella versione originale del MG ma è utile per migliorare le prestazioni del sistema. Giocando al Minority con pochi giocatori ad esempio 3, 5,7,11..., il sistema, oltre a comportamenti periodici degenera facilmente in casi patologici in cui tutti si comportano sempre allo stesso modo, per questi casi il parametro, che indica la soglia delle perdite, permette all'agente *perdente* di 'rivendicare i propri diritti di accesso alla risorsa', ricalcolandosi la strategia e quindi rompendo l'equilibrio precedentemente raggiunto. Si tratta di un metodo empirico che porta il sistema a spostare il suo punto di equilibrio, si potrebbe pensare di inserire o modificare il parametro all'occorrenza, ipotizzano un agente manager che osservando il sistema e rilevando il comportamento anomalo, utilizzi il parametro per portare il sistema ad uno stato più efficiente.

5.4.2 Realizzazione del tuning on line

Il tuning viene realizzato da un agente umano che, dopo aver osservato il sistema, comunica all'artefatto di coordinazione attraverso un'interfaccia utente, vedi figura 5.1, il valore dei parametri di tuning. Tali valori verranno

Figura 5.1: Interfaccia di inserimento parametri di tuning.

percepiti dagli agenti che modificheranno i loro singoli comportamenti, portando il sistema in un altro stato di equilibrio. Capire quali semplici regole di modifica dei parametri portino il sistema a un livello ottimale di utilizzo delle risorse è punto cruciale nella progettazione. Incapsulando queste regole sull'artefatto di coordianazione mediante ReSpecT, otterremo un sistema che si auto organizza attraverso l'ambiente cioè che evolve, interagendo con l'ambiente, verso uno stato di maggiore efficienza, in maniera analoga all'evoluzione dei sistemi naturali presentata nella sezione 1.1. Nell'interfaccia di inserimento dei parametri di tuning del programma di simulazione, premendo sul pulsante set inviamo all'artefatto il nuovo valore del parametro.

```
LogicTuple t1=null;
try {
  cn.inp(tid,new LogicTuple("tuning1",new Var("T")));
  cn.out(tid,new LogicTuple("tuning1",new Value(tun)));
}catch (OperationNotAllowedException e) {
  e.printStackTrace();
}
```

I parametri vengono letti dall'artefatto e inseriti nella tupla dei risultati di gioco. Programmazione ReSpecT per la gestione del tuning :

```
reaction(out_r(count(X)),(
  ...
  rd_r(tuning1(T1)),
```

```
  rd_r(tuning2(T2)),
  out_r(winner(A,P,CS,T1,T2,G)),
  out_r(totcount(P))
)).
```

Per il secondo parametro di tuning T2 l'agente controlla che la sua memoria sia uguale a quella richiesta dal sistema in caso contrario rigenera lo spazio delle strategie con il nuovo valore di memoria.

```
nuovoStato(Ris,S,T1,T2):-
    ...
    (M=\=T2->(S <-configura(T2),genera(St1,T2),
                retract(stat(_)),assert(stat(St1)),
                retract(mem(M1)),assert(mem(T2))
    ;M=:=T2)
```

Mentre il primo parametro viene utilizzato come soglia di confronto con il numero di perdite consecutive totalizzate dall'agente, se superiamo tale soglia l'agente rigenera la propria strategia.

```
nuovoStato(Ris,S,T1,T2):-
    ...
    lost(NumPerd),mem(M),
    (NumPerd@>T1->(S <- configura(M));NumPerd@=<T1)
```

5.5 Via al gioco

L'ambiente grafico che si è costruito permette di lanciare simulazioni, ottenendo via via i risultati medi e di varianza. Poi è anche possibile modificare i parametri del sistema, osservando il cambio di equilibrio di sistema, ottenuto modificando i parametri del behaviour di coordinazione dell'artefatto TuCSoN realizzato. Eseguendo il comando `java -cp ../lib/tucson.jar;../lib;./ tuning.Start` si fa partire l'esecuzione del gioco, con l'interfaccia di gestione. L'ambiente grafico come prima fase di configurazione lancia un thread per la creazione e configurazione del'artefatto di coordinazione eseguita attraverso il file bath `ConfigNode.bat` vedi la seguente sezione 5.5.2.

La seconda fase è di configurazione dell'applicazione di gestione e dell'interfaccia grafica. Una volta comparsa la finestra principale si possono

Figura 5.2: Finestra di settaggio parametri MG

configurare i parametri del MG relativi al numero di agenti, alla dimensione della memoria, al numero delle strategie, al numero di turni di gioco e il numero di sessioni direttamente da interfaccia, premendo il pulsante settings; la dialog che compare consente l'inserimento dati vedi figura 5.2. Una volta scelta la configurazione desiderata si lancia la creazione dei giocatori con il pulsante restar. Verà mandato in esecuzione un altro thread che eseguirà il file batch `multiCall.bat` (vedi seguente sezione 5.5.1).

L'applicazione a questo punto lancia un agente che monitorizza l'artefatto di coordinazione, tale agente ha il compito di leggere i risultati del gioco e elabora media e varianza mobili su un numero di time step prefissato. I risultati dell'elaborazione vengono graficati real time. Dall'interfaccia vedi figura 5.3 è inoltre possibile modificare i parametri di tuning durante l'esecuzione e seguire l'andamento del sistema verso il nuovo equilibrio dai grafici di media, varianza e andamento.

5.5.1 La creazione dei giocatori

L'agente implementato è in grado di giocare al MG attraverso un artefatto di coordinazione che implementa le regole del gioco. L'agente segue l'architettura descritta nel paragrafo 1.4. Attraverso il comando: `java -cp ../lib/tucson-full.jar;./ alice.tuprolog.Agent agente.pl start(localhost,1).` parte l'agente con l'esecuzione del goal

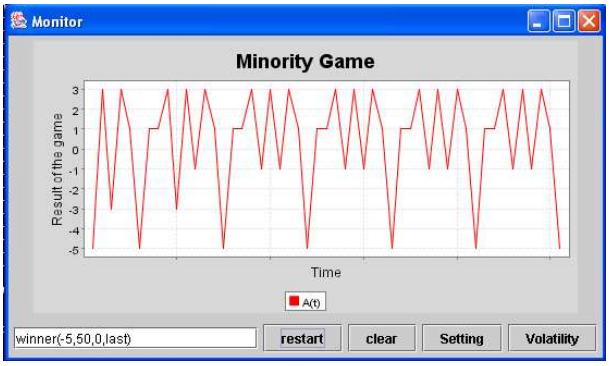

Figura 5.3: Interfaccia del programma di gestione del MG

start(localhost,1) sulla teoria logica contenuta nel file agente.pl. Per la invocazione multipla di più agenti si è creato un file bach `multiCall.bat NumAgent` che come primo parametro prende in ingresso il numero di agenti da lanciare in esecuzione. Contenuto del comando muticall.bat:

```
FOR /L %%I IN (1,1,%1) DO call :esegui %%I %2
goto :EOF
:esegui
java  -cp ../lib/tucson-full.jar alice.util.Sleep 500
start java -cp ../lib/tucson-full.jar;./ alice.tuprolog.Agent
      agenteMG4.pl  start(localhost,%1).
```

5.5.2 La creazione dell'artefatto

L'artefatto al momento della creazione viene inizializzato con tutti i parametri per la configurazione del minority game. Tale operazione è eseguita da un agente il cui unico scopo è la corretta cofigurazione dell'artefatto. Il comando batch per la creazione e configurazione dell'artefatto è `CofigNode.bat`:

```
start java -cp ../lib/tucson-full.jar alice.tucson.runtime.Node
java  -cp ../lib/tucson-full.jar alice.util.Sleep 5000
java -cp ../lib/tucson-full.jar;./ alice.tuprolog.Agent
resource.pl main(localhost).
```

L'agente che ha il compito di configurare esegue il goal main(localhost) sulla teoria resource.pl.

```
:- load_library('alice.tuprologx.lib.TucsonLibrary').
main(Address):-
    write(Address),nl,
    %tuple per l'inizializzazione dell'artefatto
    text_from_file('coordinazione.rsp',Text),
    resource @ Address ? set_spec(Text),
    resource @ Address ? out(count(0)),
    resource @ Address ? out(sum(0)),
    resource @ Address ? out(totcount(-1)),
    resource @ Address ? out(totsum(0)),
    resource @ Address ? out(qsum(0)),
    resource @ Address ? out(countsession(0)),
    resource @ Address ? out(continue(_)),
    resource @ Address ? out(game(more)),
    resource @ Address ? out(numag(11)),
    resource @ Address ? out(tuning1(100000)),
    resource @ Address ? out(tuning2(3)),
    %tuple per inizializzazione dell'agente MG
    resource @ Address ? out(numsession(2)),
    resource @ Address ? out(numgame(100)),
    resource @ Address ? out(mem(3)),
    resource @ Address ? out(spazStr(8)),
    resource @ Address ? out(numStr(2)),
    write('ok.').
```

Si carica sull'artefatto la teoria ReSpecT di specifica del comportamento del tuple centre, poi si scrivono sull'artefatto un insieme di tuple logiche di configurazione del comportamento di coordinazione dell'artefatto e la cofigurazione del gioco. Dall'esempio le tuple di configurazione del gioco sono: `out(numsession(2))` indica il numero di sessioni che l'agente giocherà, `out(numgame(100))` indica il numero di turni di gioco, `out(mem(3))` indica lo spazio di memoria dell'agente `out(spazStr(8))` indica la dimensione della strategia `out(numStr(2))` indica il numero di strategie.

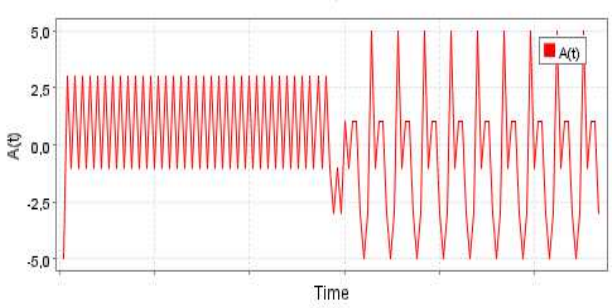

Figura 5.4: Andamento periodico del risultato del gioco con N=5 e m=3

5.6 Risultati

L'andamento delle MG può essere descritto attraverso l'uso di alcune variabili. Definiamo la media, la varianza e la varianza calcolata rispetto allo zero. In particolare sono interessanti le medie e le varianze mobili, calcolate su una finestra temporale di $NC = 40$ campioni e normalizzati sul numero di agenti N, che danno una informazione sull'evoluzione del sistema a meno dei comportamenti periodici. Infatti la caratteristica principale del sistema analizzato è che mostra un andamento periodico raggiunto dopo pochi passi di iterazione come in figura 5.4.

Osservando l'andamento della varianza mobile si nota con un numero ridotto di agenti il gioco può trovarsi in situazioni degeneri dove tutti i giocatori perdono ad ogni turno o dove tutte le scelte risultano fissate. Per rompere la simmetria creatasi, a causa della ridotta distribuzione delle strategie, modifichiamo i parametri di tuning, in particolare la soglia delle perdite viene settata a un numero simile al numero degli agenti. Cosi facendo i giocatori che stavano perdendo cercano nuove strategie e il sistema evolve in uno stato migliore, vedi figura 5.5.

Quando la varianza del sistema scende sotto al valore ottenuto con l'esperimento a giocate random $\sigma^2/N = 1$ (vedi figura 5.8), il sistema si è coordinato e ha una gestione delle risorse efficace. Il valore minimo di varianza per un turno è invece $\sigma^2/N = 1/N$. Come si vede dalla figura 5.5 il tuning permette di passare da esperimenti peggiori dell'esperimento casuale a varianze minori di uno.

E' possibile mostrare come all'aumentare della memoria m, e quindi del

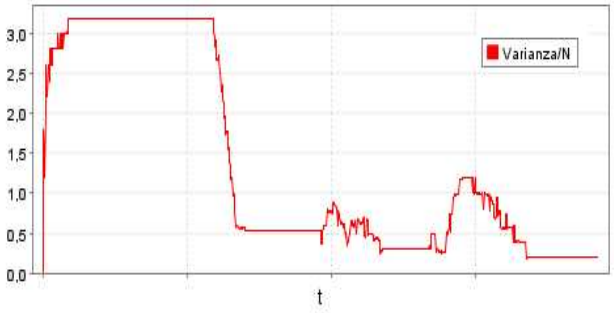

Figura 5.5: Varianza di un sistema con con N=5 e m=3. Alla prima fase di equilibrio segue una seconda fase ottenuta modificando il parametro di soglia (S=5). Segue una terza fase ottenuta cambiando la grandezza della memoria (m=5).

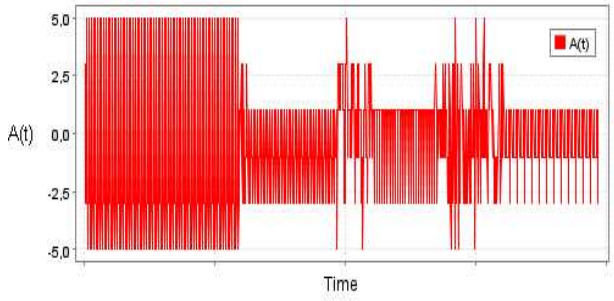

Figura 5.6: Andamento dei risultati di gioco per l'esperimento in figura 5.5.

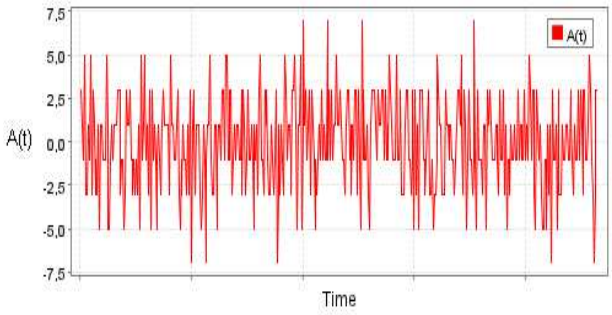

Figura 5.7: Andamento di una sessione di gioco con scelta random di 11 agenti.

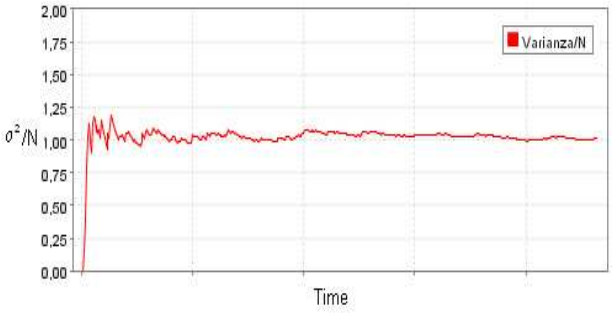

Figura 5.8: Varianza di una sessione di gioco con scelta random di 11 agenti.

parametro $\alpha = 2^m/N$, si ottengono valori di varianza che asintoticamente tendono a 1 che è la varianza del caso random. Questo concorda con quanto trovato in [13, 12].

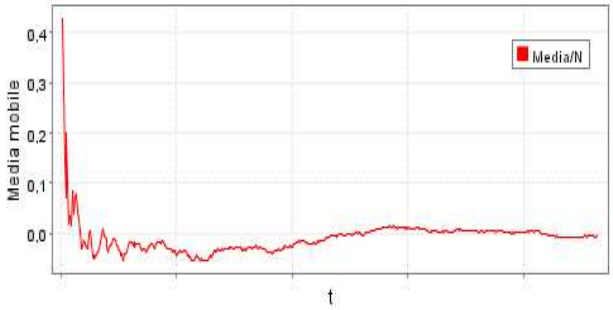

Figura 5.9: Media di una sessione di gioco con scelta random di 11 agenti.

Conclusioni

Nel lavoro di tesi, ispirandoci a forme di comunicazione indiretta – mediata attraverso l'ambiente, tra le varie entità del sistema – come la 'stigmergy', ci si è occupati di studiare un framework realizzativo per MAS utilizzando la nozione di artefatto di coordinazione, presentata in un recente articolo [1], e di istruzioni operative [1]. TuCSoN ci permette di realizzare l'artefatto come tulpe centre e programmarlo attraverso la programmazione ReSpecT per esprimerne il comportamento di coordinazione del sistema. Il motore tuProlog ci ha consentito di individuare una architettura generale per agenti logici in Java in grado di supportare le OI collegate con la propria base di conoscenza. L'agente utilizza un interprete di istruzioni operative sviluppato rappresentandone la semantica operazionale in termini di un predicato Prolog.

Il framework cosi realizzato permette l'implementazione di modelli MAS e di studiarne i comportamenti grazie anche alla flessibilità del sistema che consente di modificare in tempi brevi i comportamenti dell'agente e dell'artefatto.

Si è anche realizzata l'applicazione del framework al Minority Game (MG), che rappresenta un semplice modello di sistema dove gli agenti competono per limitate risorse interagendo con l'ambiente. Gli agenti logici, che giocano al MG, sono realizzati attraverso la scrittura delle specifiche OI e costruiscono la loro strategia di gioco in base alla memoria dei casi precedenti. La specifica ReSpecT dell'artefatto è stata realizzata in modo da garantire la coordinazione di un numero N di agenti per più giochi consecutivi e per più sessioni. Mentre l'ambiente di esecuzione, che permette di seguire gli andamenti di giocate del MG, ha una architettura basata su TuCSoN. La variazione dei parametri generali del gioco può avvenire online, ossia modificando le regole di coordinazione del MAS, senza dover fermare e far ripartire il sistema. Si è mostrato come, attraverso la possibilità di modifi-

care i parametri del behaviour di coordinazione dell'artefatto TuCSoN, sia possibile condurre il sistema in un altro stato di equilibrio più efficiente.

Lo studio delle regole che portano alla coordinazione di entità autonome, è un punto chiave per lo sviluppo dei MAS basati sugli artefatti di coordinazione. L'obiettivo è quello di capire quale interazione tra agente e artefatto generi un comportameto globalmente organizzato. Si potrebbe esplorare la possibilità di formalizzare l'utilizzo delle OI come strumento per creazione di agenti BDI che utilizzano artefatti di coordinazione. Un possibile sviluppo futuro dei MAS basati sugli artefatti di coordinazione, è la scrittura all'interno dell'artefatto delle regole che conducono il sistema ad uno stato di equilibrio efficente, ottenendo cosi un sistema auto organizzato del tutto simile agli esempi di stigmergy presenti in natura.

Bibliografia

[1] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 286–293, New York, USA, 19–23 July 2004. ACM.

[2] H. Van Dyke Parunak. Go to the ant: Engineering principles from natural agent systems. *Annals of Operations Research*, (101):75–69, 1997.

[3] A.Omicini A.Ricci and E.Denti. Activity theory as a framework of mas coordination. In *ESAW III*, volume 2577. Springer-Verlag, 2003.

[4] Mirko Viroli and Alessandro Ricci. Instructions-based semantics of agent mediated interaction. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 102–110, New York, USA, 19–23 July 2004. ACM.

[5] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999. Special Issue: Coordination Mechanisms for Web Agents.

[6] M. Wooldridge. *Rational Agents*. The MIT Press, first edition, 2000.

[7] Andrea Omicini. (homepage). url http://www.lia.deis.unibo/aomicini/.

[8] P. Norvig. *Artificial Intelligence*. Prentice Hall, second edition, 2002.

[9] E.Lamma L. Console and P.Mello. *Programmazione logica e Prolog*. Utet, 1992.

[10] G.Levi and F.Patricelli. *Prolog: linguaggio applicazioni ed implementazioni*. SSGRR, first edition, 1993.

[11] LIA. tuprolog (webpage). url http://www.lia.deis.unibo/tuProlog.html.

[12] Damien Challet. Minority game (webpage). url http://www.unifr.ch/econophysics/minorty/.

[13] E. Moro. The minority game: an intruductory guide. *Advances in Condensed Matter and Statistical Physics*, 2004.

[14] D.Challet and Y.-C. Zhang. Emergence of cooperation and organization in an evolutionary game. *Physica A*, 246:407, 1997.

[15] A. Cavagna. Irrelevance of memory in the minority game. *Physica review E*, 59:3783, 1999.

[16] S.A. Brueckner R. Savit and H. Van Dyke Paunak. General structure of resource allocation games.

[17] LIA. Tucson (webpage). url http://www.lia.deis.unibo/TuCSoN.html.

[18] H. Van Dyke Parunak and Sven Brueckner. Entropy and self organization in multi-agent system. 2001.

Ringraziamenti

Un grande, grandissimo ringraziamento alla mia famiglia: ai miei genitori Michele e Antonia che non mi hanno mai fatto mancare l'affetto e il sostegno necessario, a mia moglie Alessandra e mio figlio Giacomo che mi hanno sopportato in questo periodo di delirio, a mio fratello Alberto per le brillanti e utili conversazioni.

Un doveroso ringraziamento al professore Mirko Viroli mostratosi sempre disponibile e valida guida nel percorso di tesi, all'Università di Cesena che oltre ad avermi dato una valida formazione mi ha aiutato a crescere in tutti questi anni di studio.

Ringrazio anche la squadra di calcio Blue SGACC di Candelara che attraverso lo sport mi aiutato ad andare avanti, 'scaricandomi' dalle tensioni dell'università.

Enrico