Universitá degli studi di Padova
Facoltá di ingegneria

Corso di laurea in Ingegneria Informatica

Agent Oriented Programming, programmazione AgentSpeak in Jason

Relatore: Prof. Michele Moro

Laureando: Marco Bergantin

Matricola: 610098 IF

Anno accademico 2012 - 2013

Indice

Introduzione

Nei primi anni '70 l'intelligenza artificiale (AI) era basata sul modello di Von Neumann e sui concetti della psicologia tradizionale, includendo solo l'idea di un comportamento individuale: alla fine del decennio peró fu chiara la necessitá di avere almeno un controllo distribuito, prevedendo concetti come cooperazione e comunicazione.
I sistemi software convenzionali erano, e sono tuttora, sviluppati prevalentemente per mondi statici in cui il contenuto informativo é certo e completo, mentre i sistemi moderni devono fronteggiare con risorse limitate un contesto variabile secondo modalitá non prevedibili e per il quale si dispone di conoscenza locale, dunque parziale.
Per questi motivi la ricerca inizió a rivolgersi verso lo studio dell'interazione fra sistemi e della soluzione dei problemi in una prospettiva piú sociale: i primi passi verso la nascita di un nuovo paradigma di programmazione, la AOP, basata sulla figura dell'agente.
Questa tesi tratta l'argomento della programmazione orientata agli agenti (AOP), esaminando il concetto di agente e l'implementazione secondo l'architettura BDI di questo tipo di entitá; inoltre verrá presentato un linguaggio di programmazione per realizzare un sistema ad agenti (MAS, Multi-Agent System) ovvero AgentSpeak: essendo questo solo un linguaggio astratto, esso verrá introdotto grazie all'ausilio di Jason, che é una piattaforma basata su Java che lo implementa.

Capitolo 1

Agent Oriented Programming (AOP)

1.1 Gli Agenti

1.1.1 Cos'é un agente

Il termine 'agente' é talmente usato che é difficile dare una definizione univoca, o almeno una che sia adatta in tutti i contesti in cui viene usato: nel linguaggio comune, ad esempio, con 'agente' si intende qualcuno che agisce per conto di qualcun altro. Nell'AI invece l'agente viene definito come un'entitá che funziona continuamente ed autonomamente in un ambiente dove altri processi sono in corso ed altri agenti sono presenti. In tale definizione, piuttosto ampia, non é ben delineato il concetto di autonomia per l'agente, ma si puó evincere che si intenda che le attivitá degli agenti non richiedono costante guida umana o interventi esterni.
Una chiara definizione di agente viene data da Shoham in [1], in cui scrive: 'un agente é un'entitá il cui stato consiste in componenti mentali ben definiti, come credenze o opinioni (beliefs), capacitá (capabilities), scelte (choices) ed impegni (committments), piú o meno fortemente in corrispondenza con le rispettive controparti umane'.
Attenendosi a quest'ultima definizione, peró, qualsiasi entitá potrebbe essere modellata come un agente, anche se non é sempre un approccio vantaggioso: ad esempio si potrebbe usare tale rappresentazione per un interruttore della luce, i cui belief sono "l'utente (non) vuole accendere la luce" e con la capability di lasciare o meno scorrere la corrente (attivitá in cui sceglie di impegnarsi a seconda della sue credenze rispetto l'utente, choice e commitment). É evidente che, siccome il sistema é semplice, non é conveniente una sua descrizione come agente, in quanto, essendo noto il suo reale funzionamento, tale rappresentazione risulterebbe in un modello inutilmente pesante: lo stesso non si puó dire di sistemi piú complessi come robot, persone o anche sistemi operativi. Affinché le descrizioni di entitá sotto forma di agente siano realizzabili ed uniformi, Shoham nello stesso articolo definisce tre linee guida per questo scopo:

1. presenza di una teoria precisa riguardante ogni categoria mentale con una semantica chiara

2. dimostrabilitá che il comportamento dell'entitá segue la teoria
3. dimostrabilitá che la teoria formale ha un ruolo importante nell'analisi e il progetto

Nel contesto della AOP quando si parla di intelligenza ci si riferisce al fatto che i comportamenti dell'agente sono basati su scelte di azioni da compiere a seconda delle proprie conoscenze e sulla distinzione dello scopo del software.
A differenza di Shoham, Woolridge e Jennings nel 1995 danno due definizioni di agente: una debole e una forte. Quella debole si limita ad elencare un insieme di proprietá che un' entitá deve avere per essere considerata un agente; queste sono:

- autonomia, ovvero la capacitá di scelta e di azione senza alcun intervento esterno;
- abilitá sociale, ossia la possibilitá di comunicare con altre pari entitá attraverso un linguaggio specifico (ACL, Agent Communication Language);
- reattivitá, intesa come presenza di un meccanismo di percezione dell'ambiente e di risposta agli stimoli presentati;
- pro-attivitá, cioé la capacitá di prendere iniziative autonomamente.

La definizione forte continua ad includere queste quattro proprietá, ed aggiunge che per l'entitá si deve essere in grado di individuare delle caratteristiche di derivazione umana, quali conoscenze, opinioni, intenzioni e obblighi.
Definizioni a parte, l'agente puó essere visto come un oggetto avente uno strato di intelligenza che include un numero di capacitá come un protocollo uniforme di comunicazione, percezione, reazione e deliberazione; ha regole individuali e obiettivi che lo fanno sembrare un oggetto con iniziative. Per fare in modo che gli agenti agiscano con intelligenza, si devono sviluppare come entitá complesse e fornire di opinioni e conoscenze, in modo che essi siano in grado di perseguire i loro desideri.
In un contesto di AI si possono classificare gli degli agenti in reattivi e cognitivi.

1.1.2 Agenti Reattivi

Quando si parla di agenti reattivi, si intende una classe di sistemi che esibiscono un'intelligenza, ma sono strutturalmente abbastanza semplici e non hanno una esplicita rappresentazione dell'ambiente in cui si trovano; inoltre questi non sono in grado di eseguire ragionamenti logicamente sofisticati per cui i loro comportamenti sono basati solo sulle risposte agli stimoli. Nonostante ció, una societá di agenti reattivi in gruppo esibisce un'intelligenza superiore rispetto a quella del singolo, grazie alla cooperazione. Un esempio di questo si ha nel mondo reale con le popolazioni animali (formicai, alveari, stormi...) o con il sistema degli anticorpi umano.
In una societá di agenti reattivi, la comunicazione ha luogo indirettamente attraverso l'ambiente esterno, il che é sensato in quanto essi prendono decisioni basandosi solo sullo stato corrente dell'ambiente, senza avere memoria storica o pianificare operazioni future.

1.1.3 Agenti Cognitivi

Questa categoria di agenti si ispira alle interazioni sociali umane; essi hanno modelli espliciti del mondo esterno e strutture mnemoniche che permettono loro di mantenere la storia delle azioni passate per risolvere i problemi attuali; comunicano tra loro direttamente, usando i loro sistemi percettivi (per conoscere l'ambiente) e scambiandosi messaggi. Le caratteristiche prinicipali degli agenti cognitivi sono:

- le societá sono solitamente formate da pochi agenti;
- le comunicazioni avvengono in maniera diretta;
- hanno modelli espliciti del mondo esterno, memoria del passato e sono in grado di fare previsioni sul futuro;
- abilitá di decidere in materia di interazioni basandosi sulle loro conoscenze e di creare ed eseguire piani di azione;
- capacitá di gestire sistemi di cooperazione e coordinazione.

Questo tipo di entitá presenta una certa complessitá strutturale e sono in grado di comportarsi intelligentemente sia in societá, che sepratamente; con le piú avanzate tecniche della AI si puó addirittura dare all'agente la capacitá di pensare e di imparare. Esistono diverse architetture per implementare un agente cognitivo, ma il modello piú utilizzato é l'architettura BDI.

1.1.4 Architettura BDI

Si tratta di un modello per la realizzazione di un agente cognitivo, che si basa sulla teoria di Bratman del ragionamento pratico umano. Mentre il ragionamento teorico riguarda le credenze (ad es. se x<y e y<z allora x<z), nel ragionamento pratico invece si possono distinguere due fasi: deliberation, cioé decisione di quale stato si vuole raggiungere e means-end reasoning, ovvero la scelta di come conseguire gli obiettivi prefissati. Per modellare questo tipo di ragionamento, tale teoria prevede tre atteggiamenti mentali: Belief, Desire e Intention (BDI) corrispondenti agli stati informativi, motivazionali e intenzionali dell'agente. Piú precisamente:

- Beliefs: insieme delle informazioni riguardanti l'ambiente e il contesto; possono contenere regole di inferenza utili per la deduzione di nuove informazioni. Possono essere parziali, incompleti ed addirittura errati.
- Desires: stato motivazionale dell'agente, obiettivi che vuole raggiugere, stato finale desiderato. É con questi che ci si stacca ulteriormente dall'informatica convenzionale, che é task-oriented, ovvero in cui ogni processo viene eseguito senza che si sappia o si ricordi il perchá della sua esecuzione: questo implica l'incapacitá di recuperare l'elaborazione in caso di fallimento.
- Intentions: stato deliberativo dell'agente, ovvero i desires per cui ha pianificato delle azioni da eseguire. Insieme di piani completi che corrispondono ad un insieme di thread eseguiti in un processo che puó essere interrotto

a seguito di un cambiamento di contesto. Queste componenti sono quelle che garantiscono all'agente la pro-attivitá (tendenza a compiere azioni), la persistenza e che vincolano il suo ragionamento, in quanto l'aver adottato un'intenzione esclude degli stati futuri (ammissibilitá delle intenzioni future).

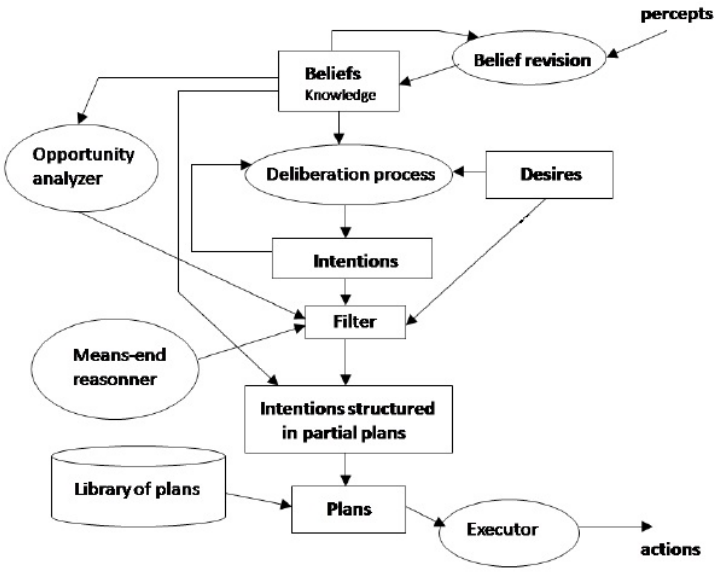

Figura 1.1: Schema Architettura BDI

Per progettare un agente BDI, é necessario specificare le sue opinioni e desideri: l'agente avrá il permesso di scegliere le sue intenzioni, basandosi su una auto-analisi degli stati inizialmente disponibili.
Nel contesto della AOP la realizzazione di queste componenti avverrá attraverso processi computazionali, che saranno eseguiti da agenti con risorse limitate. Questo ha due importanti implicazioni: calo di prestazioni in un ambiente real-time e obbligo della scelta di un altro stato da raggiungere, se questo si rivelasse non ottimale solo in corso di elaborazione.

Esempio

Premettendo che, a patto di usare un formalismo appropriato e sotto determinate condizioni, qualsiasi entitá potrebbe essere modellata secondo l'architettura BDI, qui viene presentato un esempio di come si potrebbero scegliere le tre componenti fondamentali per un robot calciatore.

Per quanto riguarda i belief dell'agente, l'ossatura del suo stato informativo potrebbe essere composta da delle entry che rappresentano le posizioni di sé stesso, dei suoi compagni di squadra, degli avversari e della palla. Queste informazioni probabilmente saranno ottenute con uno dei seguenti metodi: sensori di posizione, telecamere interne, sistemi di posizionamento (tipo GPS), calcolo interno a partire da una posizione 0 con un'equazione del moto del corpo in

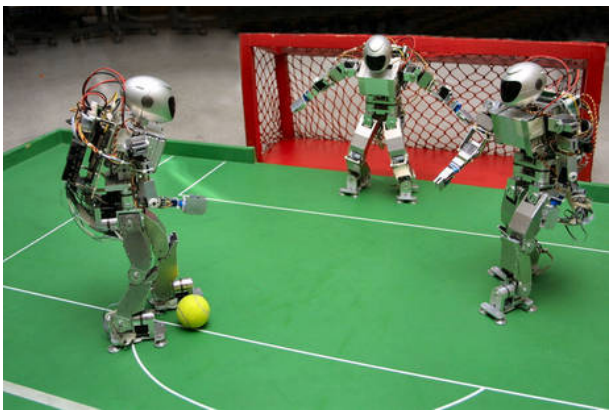

Figura 1.2: Robot calciatori in azione

questione, comunicazione con un compagno di squadra (un altro agente) o anche una combinazione dei precedenti modi. Supponendo che il robot conosca il campo da gioco (grazie ad un modello statico precaricato in memoria), l'insieme dei belief di ciascun agente, potrebbe essere composto dai seguenti elementi:

- positionSelf(x,y), la propria posizione
- positionTeamMate(x,y,tmNum), posizione di un compagno di squadra (identificato da un numero)
- positionOpponent(x,y,OpNum), posizione di un avversario (sempre identificato con un numero)
- positionBall(x,y), posizione della palla

Queste informazioni servono al robot per decidere cosa fare in ogni momento: per esempio nel momento in cui l'agente crede di avere la palla (ovvero la posizione di questa é entro il suo raggio d'azione) questo puó valutare la propria posizione nel campo, dei suoi compagni e degli avversari per scegliere se effettuare un passaggio o tentare un tiro in porta.

I desire dell'agente saranno verosimilmente diversi a seconda del ruolo che il robot ha nella squadra: ad esempio l'attaccante non avrá il desire di effettuare una parata, come il portiere non avrá quello di tentare il goal. Inoltre i vari desire sono 'attivati' a seconda dello stato dell'ambiente (e quindi in base ai belief dell'agente), infatti, a seconda della posizione degli avversari rispetto ai propri compagni di squadra, il robot con la palla puó decidere a chi passarla, scartando dalla decisione quelli marcati. Possibili desire per i robot:

- passTo(num), passaggio al compagno di squadra indicato, la cui posizione sará memorizzata fra i belief, e sará recuperata all'occorrenza
- shoot(), tiro in porta, la cui posizione é nota, quindi non servono parametri
- catch(), parata (ovviamente solo per il portiere), necessita la valutazione del belief positionBall(x,y)

- move(x,y), per far muovere il robot dalla posizione corrente a quella specificata

Prendendo il desire 'catch()' come esempio, il funzionamento é il seguente: quando il robot portiere si accorge che la palla é in una posizione 'pericolosa', ovvero vicino alla porta e in avvicinamento, viene attivato l'obiettivo di pararla, quindi al momento opportuno (deciso in base alla velocitá della palla ed alla capacitá di movimento del robot) l'agente deciderá di bloccarla in un modo coerente con l'hardware dell'agente (tuffo, semplice spostamento, allungando un arto).

A questo punto si possono definire le possibili intention per un agente di questo tipo. Queste componenti mentali rappresentano in sostanza dei piani di azione, attivati dalla decisione dell'entitá di perseguire un desire, quindi nel nostro esempio le intention dei robot potrebbero essere:

- shootKick(), per calciare la palla con l'obiettivo di tirarla in porta
- passKick(num), calcio alla palla, ma per un passaggio ad un compagno di squadra indicato
- catchDive(), serve al portiere per parare, tuffandosi verso la posizione in cui pensa la palla arriverá in porta
- runTowards(x,y), spostamento del robot, utile anche al portiere, che potrebbe decidere di parare un tiro semplicemente ostruendo la porta col proprio 'corpo'

Probabilmente l'intention 'shootKick()', per esempio, prevederá che venga eseguita un'azione (simile all'invocazione di un metodo) per calciare la palla: questa si occuperá internamente di determinare l'angolo con cui il robot deve girarsi per colpirla correttamente ed indirizzarla in porta, di chiamare la funzione per comandare l'hardware della 'gamba' affinché questa venga estesa per eseguire l'azione fisica del calcio e di regolare la tempistica di tutte queste sotto-azioni (e quindi evitando che il robot calci a vuoto).

1.2 Framework AOP

Per realizzare in pratica i principi della AOP, é necessario un framework con tre componenti fondamentali:

- un linguaggio formale per descrivere lo stato mentale,
- un linguaggio di programmazione per definire e programmare gli agenti e
- un agentificatore (agentifier).

Il primo componente deve essere soggetto a restrizioni, con una sintassi e una semantica chiare per descrivere lo stato mentale che sará definito unicamente da alcune componenti, ad esempio, se si sceglie di implementare l'agente secondo l'architettura BDI, questo linguaggio consentirá di specificare i belief, i desire e le intention.
Il linguaggio di programmazione per la definizione e la programmazione degli

agenti sará preferibilmente interpretato e prevederá delle primitive del tipo REQUEST e INFORM con una semantica coerente con quella dello stato mentale.
L' agentifier ´e un componente che Shoham introduce nel suo articolo con lo scopo di convertire dei dispositivi neutrali in agenti programmabili, ovvero instaurare una corrispondenza tra i costrutti dell'AOP e la macchina. Siccome la programmazione ad agenti vuole essere uno strumento versatile per controllare e coordinare diversi tipi di dispositivi in altrettante realtá, non sempre tale collegamento é presente o facilmente realizzabile: ad esempio se si volesse gestire con l'uso del paradigma ad agenti un insieme di elettrodomestici tipo un impianto home-theatre, il primo compito del progettista sarebbe quello di rappresentare ogni entitá presente come agente, trovando un modo per agganciare alla parte di linguaggio le funzioni che ognuno dei componenti é in grado di eseguire, in quanto quasi sicuramente la casa produttrice non fornirá il sistema con un'interfaccia adatta giá di fabbrica.
In sostanza, l'agentificatore dovrebbe colmare il divario tra la macchina a basso livello e la parte intenzionale ad alto livello degli agenti: un elemento fondamentale per ottenere questo risultato é l'interprete.

1.2.1 Agent Interpreter

Il ruolo di un programma ad agenti é quello di controllare l'evoluzione dello stato mentale degli agenti: le azioni vengono intraprese come effetto collaterale del fatto che un agente si impegna ad eseguire un certo compito, e che é giunto il momento di eseguirlo.
Un linguaggio di programmazione, per ottenere questo risultato, dovrá includere delle strutture dati che rappresentino le varie componenti logiche ed extralogiche, cioé su tali strutture dati che aderiscano alle proprietá delle varie componenti logiche, ad esempio non deve essere possibile istanziare due strutture dati di tipo belief per lo stesso agente se queste sono contraddittorie.
Il comportamento degli agenti é regolato dall'iterazione di due passi:

1. lettura dei messaggi presenti ed aggiornamento conseguente dello stato mentale, tenendo conto degli attuali opinioni e obiettivi

2. esecuzione di una o piú delle azioni fra quelle presenti per il raggiungimento di un obiettivo, con eventuale aggiornamento delle opinioni

Le azioni in cui gli agenti si impegnano possono includere anche comunicazioni (informa, richiedi ecc.) o azioni private (interne), tipo l'aggiornamento dei propri componenti mentali.

L'esistenza di un clock é di importanza centrale per le operazioni dell'interprete, nonostante il fatto che non tutti gli agenti devono per forza avere i propri clock sincronizzati: questa situazione si verifica piú facilmente se non tutti gli agenti risiedono nella stessa macchina. Esistono comunque protocolli di sincronizzazione per limitare questo fenomeno.

Un altro componente fondamentale oltre a quelli elencati per un framework AOP é un meccanismo di comunicazione fra agenti basati sullo scambio di messaggi con relativi comandi.

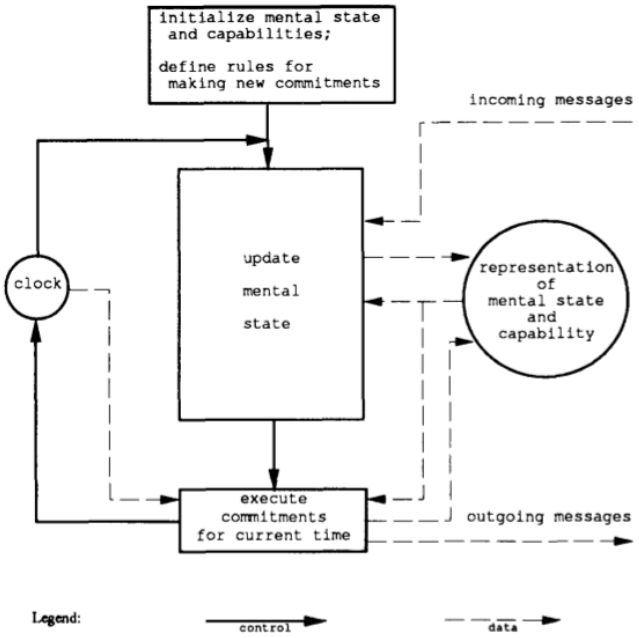

Figura 1.3: Interprete per Agenti BDI

1.3 Comunicazione fra Agenti

1.3.1 Agent Communication Languages (ACL)

Ogni agente deve avere la capacitá di percepire e agire efficientemente nel proprio ambiente, ovvero di spedire e ricevere messaggi: a tale scopo sono necessari un linguaggio e un protocollo di comunicazione comuni, che permetta di realizzare delle forme di interazione, cooperazione e negoziazione. Siccome diversi agenti possono processare per e riferirsi ad uno stesso oggetto in modo non uniforme (es. diverse interfacce per ciascuna entitá), é necessario che la struttura del linguaggio permetta l'integrazione di queste rappresentazioni disuguali.

Un modo efficace di implementare un ACL é quello di basarsi sulla teoria dello speech acts (atti discorsivi), che non é altro che un approccio linguistico utilizzato per modellare l'adattivitá del linguaggio umano. I filosofi Austin e Searle svilupparono questa teoria, vedendo il linguaggio naturale come suddiviso in azioni come richieste, suggerimenti, impegni, risposte ecc. e notando che certe espressioni sono paragonabili ad azioni fisiche che sembrano modificare lo stato dell'ambiente. I tre aspetti principali del modello sono:

- locuzione, un'espressione fisica da parte di chi parla, come una richiesta;
- illocuzione, concetto inteso dall'espressione;
- perlocuzione, azione risultante dalla locuzione.

Queste tre componenti saranno soggette a delle limitazioni nel contesto della AOP rispetto ad uno di comunicazione umana, infatti un buon ACL dovrebbe

fornire un meccanismo con il quale un agente che inoltra una richiesta ad un suo pari sia in grado di prevedere le possibili risposte che possono pervenire dall'agente interpellato in relazione al messaggio trasmesso in partenza, potendo quindi fare previsioni sulle reazioni del ricevente ed eventuali cambiamenti di stato conseguenti.

Il concetto di illocuzione é usato per definire il tipo di messaggio nella toria dello speech acts. Le intenzioni del sender sono chiaramente definite e il ricevente non ha dubbi riguardo al tipo del messaggio arrivatogli. Nel contesto della comunicazione tra agenti gli atti discorsivi sono anche detti performativi: tale termine serve per identificare la parte illocuzionaria di una classe speciale di espressioni, quali ad esempio promesse, rapporti, richieste ed ordini. La teoria vede l'interazione fra agenti come scambio di conoscenze in quanto i messaggi hanno un'intenzione (illocuzione) e un contenuto (locuzione), ad esempio si puó trovare un comando del tipo: TELL(A,B,P), con cui l'agente A informa B della condizione P.

1.3.2 Protocolli di Comunicazione

Un protocollo é definito come un insieme di regole a supporto della comunicazione: di solito controlla il formato, il contenuto e il significato dei messaggi. Vengono usati anche per assistere la negoziazione e l'interazione fra agenti. Quando piú agenti hanno obiettivi in conflitto, il compito del protocollo é di massimizzare i payoff degli agenti. Si possono suddividere in tre categorie:

- announcement protocol, usato dalle entitá per informare gli altri dei servizi che é in grado di offrire
- ask-about protol, consiste di due messaggi e permette ad un agente di chiedere qualcosa ad un altro (ad esempio per chiedere un parere riguardo ad un belief)
- task/action agreement protocol, definisce una sequenza di messaggi utile ad un agente per incaricarsi di un compito di un altro agente.

1.3.3 Coordinamento

Affinché gli agenti collaborino in armonia, é necessario piú dell'ACL, in quanto essi possono eseguire operazioni diverse, avere gli obiettivi piú disparati nel contesto dell'ambiente (anche in conflitto fra loro) e possono scambiarsi messaggi gli uni con gli altri.

In un sistema di agenti possono instaurarsi collaborazioni o cooperazioni.

Una collaborazione viene stabilita quando un agente ha l'autonomia di eseguire un compito e, per velocizzare l'operazione, é autorizzato ad accettare e/o chiedere l'aiuto di altri agenti.

La cooperazione invece si ha quando l'agente in questione non é in grado di portare a termine il lavoro per conto suo.

Perché si coordinano gli agenti:

- prevenire il caos, in cui ci si puó facilmente trovare se non c'é un'entitá di controllo centralizzata

- distribuire informazioni, risorse e competenze, in modo da risolvere problemi complessi piú efficientemente
- incrementare l'efficienza del sistema
- gestire interdipendenze tra le azioni degli agenti
- gestire la maggior parte dei problemi, in quanto questi richiedono diversi tipi di conoscenze per essere risolti, per cui nessun agente puó farcela da solo e quindi si combinano le diverse capacitá di ciascuno
- gli agenti solitamente hanno informazioni incomplete riguardo al problema

I tre piú comuni meccanismi di coordinamento sono: strutture organizzative, meta-livelli e il multi-agent planning.

Con le strutture organizzative si formalizzano le interazioni tra individui, in quanto offrono un ambiente vincolato e previsioni riguardo al comportamento degli agenti (set di regole) che guida le loro decisioni e azioni; inoltre vengono mantenute informazioni a lungo termine riguardo gli agenti e la societá complessiva.

I meta-livelli sono strutture astratte che si possono instaurare tra agenti che sono correlati da un problema comune: grazie a queste astrazioni é possibile formalizzare le interazioni, relazioni e comunicazioni fra questi gruppi di agenti.

Con multi-agent planning, si intende la pianificazione delle azioni degli agenti riguardo a compiti condivisi con comportamenti in linea con l'obiettivo generale del sistema: sará cosí possibile prevedere come questi influenzeranno l'ambiente.

1.3.4 Negoziazione

L'obiettivo principale della negoziazione é quello di risolvere i conflitti che potrebbero interferire nel comportamento cooperativo. Per ottenere questo risultato potrebbe essere necessario:

- modificare i piani locali di un agente se l'interazione non avviene con successo
- identificare situazioni dove potenziali interazioni sono possibili
- ridurre inconsistenza e incertezza riguardo ai differenti punti di vista o ai piani in comune, attraverso lo scambio organizzato di informazioni.

Il meccanismo di negoziazione migliora l'efficienza del sistema consentendo agli agenti di ridistribuire i compiti fra di loro con l'obiettivo di minimizzare gli sforzi individuali: se i compiti vengono distribuiti correttamente nel gruppo si otterrá anche una maggiore stabilitá per ogni agente (es. minor probabilitá di blocco, di obbligo di cambiare intentions a causa dell'irrealizzabilitá di un desire in corso di elaborazione ecc.).

1.4 Confronto fra AOP e OOP

Nella presentazione della AOP, Shoham dichiara che questa "puó essere vista come una specializzazione della OOP"; nonostante questo, ovviamente, vi sono delle differenze tra le due.

OOP versus AOP

	OOP	AOP
Basic unit	object	agent
Parameters defining state of basic unit	unconstrained	beliefs, commitments, capabilities, choices, ...
Process of computation	message passing and response methods	message passing and response methods
Types of message	unconstrained	inform, request, offer, promise, decline, ...
Constraints on methods	none	honesty, consistency, ...

Figura 1.4: Tabella riassuntiva differenze AOP vs OOP

La AOP presenta costrutti mentali per la progettazione e l'analisi del sistema computazionale e per la rappresentazione dello stato, compito che nella OOP é svolto dalle variabili di stato. Mentre la OOP propone la visione di un tale sistema come composto di moduli in grado di comunicare fra loro e con modalitá individuali di gestione dei messaggi in arrivo, la AOP specializza questo modello stabilendo lo stato dei moduli (ovvero lo stato mentale degli agenti) in modo che sia composto da opinioni, capacitá e decisioni, ognuna delle quali prevede una sintassi ben definita. Sono posti vari vincoli nello stato mentale degli agenti che sono approsimativamente in corrispondenza con le loro controparti umane. Una computazione vede gli agenti informarsi, inoltrare richieste, fare offerte, accettarne, rifiutarne e assistersi l'un l'altro.

Il comportamento di un agente estende quello di un oggetto, perché gli agenti hanno libertá di controllare e di cambiare i loro comportamenti, inoltre non richiedono stimoli esterni per completare i loro compiti: per questo gli agenti sono detti elementi attivi, e gli oggetti passivi. Il comportamento degli agenti presenta un certo grado di imprevedibilitá, a causa delle interazioni che si verificano fra di loro, della loro autonomia e per il fatto che gli effetti delle loro azioni non sono certi prima che queste vengano compiute. Hanno l'abilitá di comunicare con l'ambiente e le altre entitá, anche ingaggiando transazioni multiple concorrentemente.

Mentre nella OOP é presente il concetto di classe per la definizione del tipo e della funzione di un oggetto, nella AOP questa viene sostituita con il ruolo; analogamente i metodi lasciano il posto ai messaggi, i quali sono indipendenti dall'applicazione in quanto seguono la sintassi standard dell'ACL in uso, a differenza dei metodi per un oggetto che deveno essere specifici, ad-hoc per quella classe.

Dal punto di vista computazionale, le aree chiave di differenza sono l'autonomia e l'interazione: ulteriori concetti fonte di diversificazione che l'AOP adotta sono

- decentralizzazione, cioé nella OOP gli oggetti sono organizzati centralmente, perché i metodi vengono invocati sotto il conrollo di altri componenti del sistema, mentre per gli agenti la computazione puó avvenire sia in maniera centralizzata che decentralizzata (internamente all'agente)

- classificazione multipla e dinamica, ovvero gli agenti presentano un meccanismo piú flessibile rispetto a quello degli oggetti riguardo al proprio tipo, in quanto un oggetto, una volta istanziato da una classe, non potrá mai cambiarla

- impatto minore, se in un sistema di agenti ne viene perso uno durante un'elaborazione, il sistema puó far fronte a questa evenienza grazie al supporto della collettivitá degli agenti presenti nell'ambiente; nella OOP al mancare di un oggetto viene lanciata un'eccezione come minimo

- emergenza, ovvero in un gruppo di agenti che collaborano vi saranno caratteristiche emergenti non proprie del singolo agente, ma dovute al fatto che le singole entitá formano una societá, grazie all'interazione propria dell'AOP; gli oggetti, invece, non interagiscono per propria natura fra loro, o meglio non senza un meccanismo a livello superiore, come un thread di controllo ad esempio.

Capitolo 2

AgentSpeak in Jason

Nonostante gli agenti BDI siano stati oggetto di ricerca sia da un punto di vista teorico che della progettazione pratica, rimane comunque un divario tra i due: questo accade principalmente a causa della complessitá richiesta per provare il ruolo della parte teorica nell'implementazione logica (requisito fondamentale per descrivere un'entitá come agente). Per ridurre la distanza tra teoria e pratica in precedenza la norma era quella di concentrarsi su una architettura BDI astratta, che servisse sia come idealizzazione di un sistema implementato, che come strumento di analisi per le proprietá teoriche. Anche adottando tale astrazione, peró, rimane comunque difficoltoso dimostrare una corrispondenza uno-a-uno tra il modello teorico e l'implementazione astratta.

AgentSpeak(L) é un linguaggio di programmazione astratto, vincolato, con eventi ed azioni che vuole offrire una formalizzazione alternativa per gli agenti BDI, in armonia con la parte teorica della AOP, mentre Jason é una sua implementazione pratica in Java, o meglio, una piattaforma di sviluppo per sistemi mutli-agente (MAS, Multi Agent Systems) basata su un interprete per una versione estesa di AgentSpeak.

Come detto in precedenza, gli agenti BDI sono sistemi situati in un ambiente mutevole, che ricevono continui input percettivi e svolgono azioni che influiscono sull'ambiente, il tutto basandosi sul loro stato mentale interno: in AgentSpeak il comportamento di un agente é dettato dai programmi.

L'architettura dell'agente BDI adottata in AgentSpeak é la chiave con cui questo linguaggio di programmazione cerca di unificare teoria e pratica, in quanto utilizza un semplice linguaggio di specificazione come modello esecutivo di un agente e, solo in seguito, permette di ascrivere le attitudini mentali, ma da un punto di vista esterno.

E' di notevole importanza, inoltre, precisare la distinzione fra programma ad agenti ed architettura ad agenti, in quanto l'architettura é la struttura software all'interno della quale il programma é eseguito: l'interpretazione di quest'ultimo determina effettivamente il ciclo di ragionamento (reasoning cycle) di un agente. [1]

[1] NOTA: Il nome per esteso del linguaggio, presentato da Rao in [5], sarebbe AgentSpeak(L), ma siccome viene raramente usato per esteso, anche qui verrá indicato solo con AgentSpeak.

2.0.1 Multi-Agent Systems (MAS)

Si é detto che Jason usa AgentSpeak per implementare sistemi multi agente (MAS), ma finora si é descritto solo il concetto di agente singolo che coopera con altri agenti, senza specificare molto sul mondo abitato da questi. Ebbene con MAS (Multi-Agent System) si intende un sistema in cui piú agenti abitano lo stesso ambiente, che risulta quindi condiviso: ognuno di questi ha una propria sfera d'influenza, cioé una porzione dell'ambiente che é in grado di controllare parzialmente. Nella maggior parte dei casi le varie sfere d'influenza si sovrappongono, risultando in una parte di ambiente controllata unitamente da piú agenti: in un sistema di questo tipo, essi possono instaurare fra loro diversi tipi di rapporti organizzativi (come di paritá o di subordinazione) ed avere una certa conoscenza l'uno dell'altro.

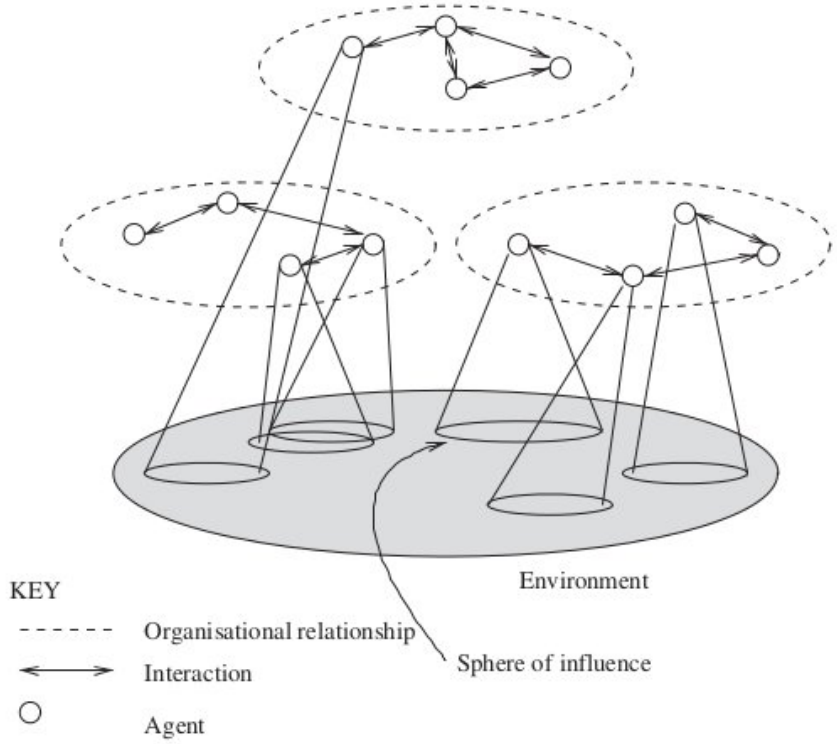

Figura 2.1: Tipica Struttura di un MAS

Ora, conoscendo le entitá agente e avendo un'idea di come queste possano interagire in un MAS, é possibile identificare dei requisiti che un linguaggio di programmazione deve soddisfare per essere in grado di dirigere un sistema del genere:

- supporto della delegazione di obiettivi ad alto livello, ovvero quando si vuole delegare un compito ad un agente, lo si vuole poter fare semplicente comunicandogli cosa deve fare e non dandogli una descrizione esecutiva (come deve raggiungere l'obiettivo, porzioni di codice)

- problem solving orientato ai traguardi (goal), cioé si vuole che gli agenti siano in grado di agire al fine di raggiungere gli scopi che sono loro stati delegati, e che lo facciano in maniera sistematica
- produzione di sistemi che sono reattivi rispetto all'ambiente
- integrazione del comportamento goal-oriented e reattivo, cioé, se un agente sta agendo verso il completamento di un compito, deve comunque essere in grado di rispondere agli stimoli esterni e viceversa
- supporto della comunicazione a livello di conoscenze e della cooperazione

2.0.2 Basi per la programmazione logica

In seguito si descriveranno le modalitá con cui si puó effettivamente scrivere del codice AgentSpeak in Jason per implementare dei MAS composti di agenti BDI.

Prima, peró sono necessari alcune precisazioni sui termini che verranno utilizzati.

- Ogni simbolo (sequenza di caratteri) che inizia con una lettera minuscola é detto atom, é equivalente ad un costante ed é usato per rappresentare individui o oggetti.
- Se il simbolo inizia con una lettera maiuscola viene interpretato come una variabile; inizialmente sono tutte non istanziate, e il loro istanziamento é detto unificazione: una formula é detta fondata (ground) se tutte le sue variabili sono state istanziate.
- La parola termine (term) é usata per riferirsi ad una costante, una variabile o una struttura.
- Le informazioni vengono rappresentate in forma simbolica dai predicati (predicate), che esprimono una particolare proprietá.
- Un predicato (o la sua negazione) che rappresenta una relazione fra oggetti é detto letterale (literal).
- Una annotazione (annotation) é un termine complesso (una struttura) che fornisce dettagli fortemente associati con un particolare belief

2.1 Architettura BDI in AgentSpeak

Le tre componenti dell'achitettura BDI vengono realizzate in AgentSpeak attraverso i Belief, i Goal e i Plan: c'é quindi un leggero scostamento da quelli che sono i pilastri originali della teoria (Belief, Desire e Intention), infatti il significato é lievemente diverso e molto piú pratico.

2.1.1 Belief

La prima cosa da fare quando si programma un agente in Jason é quella di specificare i belief di partenza. In AgentSpeak il belief base, ovvero l'insieme dei belief di ciascun agente, é realizzato attraverso un set di predicati e letterali

che rappresentano le credenze dell'agente. A questo riguardo é doverosa una precisazione, riguardo alla modalitá di veritá: infatti i belief non sono concetti veri in senso assoluto, ma fatti che l'agente crede siano attualmente veritieri. Per esprimere una proprietá di un oggetto o un individuo con un predicate la sintassi é la seguente:

property(object).

ad esempio aggiungendo tall(john). al belief base, l'agente crederá che john sia alto. Per inserire tra i belief il fatto che sussiste una relazione fra due oggetti attraverso l'uso di un literal, si dovrá scrivere qualcosa del tipo:

relationship(object1,object2).

ad esempio l'aggiunta di likes(jack,music) avrá come effetto quello di far credere all'agente che a jack piace la musica.

Annotazioni

Per ogni belief é possibile specificare un'annotazione, includendola fra parentesi quadre dopo il literal, cosí da esprimere ulteriori informazioni riguardo a quella particolare credenza dell'agente: ad esempio

busy(mark)[expires(spring)].

sta ad indicare che l'agente crede che mark sia attualmente impegnato, ma, quando si verificherá la condizione spring, questo belief non sará piú vero. Nonostante il fatto che tutto ció che si puó indicare con una annotazione puó essere espresso anche con un literal piú articolato (nel caso di prima busy(mark,spring)), l'uso di queste strutture ha comunque due vantaggi:

1. eleganza della notazione
2. gestione facilitata del belief base

Un chiarimento: per l'interprete Jason una annotazione non significa niente di default, ma é possibile personalizzare con funzioni Java questo comportamento. Ció, per il nostro esempio, significa che se non si specifica nulla, il belief busy(mark) rimarrá nel set anche dopo il verificarsi dell'evento spring, ma Jason fornisce il supporto per poter specificare un metodo Java che, all'attuarsi della condizione, rilevi che il belief é diventato obsoleto e lo rimuova dal belief set dell'agente.

Esistono peró anche altre importanti categorie di annotation, che invece hanno significato per l'interprete: in particolare source, che viene utilizzata per memorizzare eplicitamente la provenienza dell'informazione immagazzinata come belief. Ci sono tre tipi di source per le informazioni degli agenti in un MAS:

- *percept*, per le informazioni ottenute dalla ripetuta osservazione dell'ambiente
- *nomeAgente*, se un belief é stato aggiunto come risultato di una comunicazione con un altro agente (di cui si sta specificando il nome)

- *self*, per rappresentare una categoria di annotazione detta note mentali (mental notes), ovvero degli 'appunti' che l'agente tiene in memoria su azioni compiute nel passato o promesse, che facilitano l'esecuzione dei plan

Negazione

Per quanto riguarda i belief di un agente, é necessario fare attenzione quando si nega un predicato o un letterale. Infatti, nella programmazione logica, la negazione é gestita con l'assunzione di mondo chiuso (closed world assumption). In poche parole ció significa che l'interprete ragiona nel seguente modo: tutto ció che non é risaputo essere vero, né derivabile dai fatti di cui si é al corrente usando le regole del programma, viene assunto come falso. In questo contesto, l'unico tipo di negazione ammesso é tale che la negazione di una formula é vera se l'interprete non riesce a derivarla dai fatti e regole del programma, e viene indica in AgentSpeak con l'operatore 'not'.
Questo tipo di negazione in molti casi risulta troppo semplice, infatti, sotto la closed world assumption, tutto ció che l'agente non crede essere vero, viene assunto come falso. Per alcune classi di sistemi aperti (come il Web ad esempio) é utile poter esprimere l'ignoranza di un agente riguardo ad un argomento, ovvero la condizione per cui fra i suoi belief non compare niente a riguardo, quindi non puó arrivare a nessuna conclusione.
Per tali motivi é stato introdotto un altro tipo di negazione, detto negazione forte, che serve ad esprimere che un agente crede esplicitamente che qualcosa é falso e viene denotato dell'operatore '~'.Ad esempio, se un agente sa che not p. e not ~p. sono entrambe affermazioni vere allora si puó concludere che esso non ha alcuna informazione riguardo a p, in quanto né p né ~p possono essere derivati (o sono presenti) nel belief set dell'agente.

Regole

In Jason é possibile specificare delle regole (rules) per ogni belief. Questi costrutti permettono all'agente di evincere nuove informazioni basandosi su cose che giá sa, con lo scopo di semplificare alcuni compiti, ad esempio rendendo le condizioni in un plan piú succinte.
Una regola si specifica con il simbolo ':-' dopo un letterale: questo rappresenta la conclusione che l'agente dovrá trarre se la regola (ovvero ció che c'é dopo il ':-') é vera, ovvero se la condizione che essa rappresenta é soddisfatta. Un esempio:

colour(col,obj) :- colour(col,obj)[source (S)] &*(S==self | S== percept)*

é la sintassi con cui si specifica una regola sul belief colour, e in questo caso si vuole imporre all'agente la credenza che un oggetto sia di un certo colore solo se tale informazione proviene da se stesso (mental note) o se é appena stato percepito dall'ambiente, quindi evitando di 'dare retta' ad altri agenti nel caso questi comunichino qualcosa riguardo al colore di un oggetto.
Il tipo di ragionamento che l'agente esegue per valutare la clausola imposta da una regola é detto ragionamento teorico, cioé su rappresentazioni astratte di fatti risaputi allo scopo di derivarne ulteriori conoscenze.

Belief iniziali

Sebbene il programmatore abbia la facoltá di manipolare ed ispezionare le belief base in diversi modi, tutto quello che si deve fornire inizialmente all'agente AgentSpeak é il set dei belief iniziali, ovvero il codice sorgente deve dire all'interprete quali belief dovrebbero essere fra quelli dell'agente quando inizia la sua esecuzione, ovvero prima che inizi ad osservare l'ambiente attraverso i suoi sensori per la prima volta. Se non diversamente specificato, ogni belief nel codice di un agente vengono assunti come note mentali.
Ad esempio il codice di un semplice agente potrebbe essere questo:

started.

ovvero esso possiede come unico belief quello di essere partito. Tale belief potrá essere usato in un goal o in un plan successivamente.

2.1.2 Goal

Nella programmazione ad agenti, la nozione di goal (obiettivo, fine) é fondamentale. Esistono due tipi di goal in AgentSpeak: gli achievement goal e i test goal. I primi si indicano con !g(t) ed esprimono il fatto che l'agente vuole ottenere uno stato dell'ambiente in cui g(t) é un belief veritiero. Un test goal ?g(t), invece, indica che l'agente vuole verificare se g(t) é un belief vero o almeno se puó discendere da uno di quelli correntemente nello stato dell'entitá: questo puó richiedere solo il recupero di informazioni dal belief set, oppure l'esecuzione di alcune azioni da parte dell'agente per verificare lo stato dell'ambiente.
Quando si rappresenta un achievement goal !g in un programma ad agenti significa che l'agente si impegna ad agire in modo da modificare l'ambiente finché esso, attraverso la percezione, non crederá che g sia un belief veritiero (nel caso di un achievement goal). Ad esempio, aggiungendo ai goal dell'agente !own(house), questo agirá in modo da avere nel suo belief set own(house).
Normalmente un test goal, invece é usato semplicemente per recuperare delle informazioni dal belief base dell'agente: queste potrebbero essere giá state memorizzate o potrebbero richiedere l'esecuzione di alcune operazioni (come il sensing dell'ambiente). Per esempio, il goal ?saldoCC(xEuro) potrebbe corrispondere ad una ricerca nel belief set di un letterale del tipo saldoCC(300.00): se questo non fosse presente potrebbe innescare l'esecuzione di un plan per chiedere il saldo del conto corrente direttamente alla banca.
Come per i belief, é possibile specificare dei goal di partenza nel codice dell'agente: questi sono quelli che tenterá di raggiungere fin dall'inizio. Le variazioni nei belief e nei goal sono gli eventi che fanno partire l'esecuzione dei plan.

2.1.3 Plan

I plan sono delle sequenze di azioni che un agente si impegna ad eseguire in risposta ad un evento. Un plan in AgentSpeak consta di tre parti: il triggering event, il context (assieme costituistono l'head del plan) e il body. Queste sono separate da ':' e '<-'con la sintassi:

triggeringEvent : context <- *body*

Per ogni plan é inoltre possibile specificare un'etichetta (plan label), che puó essere vista come un nome. In realtá tutti i piani hanno etichette generate automaticamente da Jason se non ne viene specificata alcuna. Queste sono necessarie in quanto a volte risulta utile potersi riferire ad un plan (o ad una sua istanza) durante l'esecuzione: questo é possibile grazie alle label in quanto esse associano dei meta-dati al piano. La sintassi é la seguente:

@label te :cntx <- *body*

Un'etichetta puó anche non essere un semplice termine, infatti puó anche essere un predicato con annotazioni: é proprio con queste che si possono specificare i meta-dati (per interpretarli sará di dovere scrivere del codice Java a parte).

Triggering Event

Ci sono due importanti aspetti del comportamento di un agente: reattitá e pro-attivitá. Gli agenti hanno degli obiettivi a lungo termine che tenteranno continuamente di raggiungere (pro-attivitá), peró, mentre agiscono in questo senso, essi devono comunque essere reattivi ai cambiamenti dell'ambiente, in quanto tali variazioni potrebbero determinare se gli scopi dell'agenti sono ancora raggiungibili o meno. Certi cambiamenti, invece, potrebbero aprire nuove opportunitá per l'agente, che forse puó riconsiderare il perseguimento di nuovi goal o di altri sospesi a causa di variazioni che li hanno preclusi in precedenza. Per questi motivi i due tipi di cambiamenti importanti in un programma ad agenti sono quelli nei belief e nei goal (aggiunta e rimozione): questi creano all'interno del sistema degli eventi sulla base dei quali gli agenti agiranno.
Ebbene il triggering event esiste per dire all'agente, per ogni plan, quali sono gli specifici eventi per cui ciascun piano va utilizzato. Se ha luogo un evento che combacia con il triggering event di un plan, si dice che esso é relevant per un particolare evento.
Dato che esistono due tipi di goal, un solo tipo di belief e due tipi di eventi (aggiunta o rimozione) si ottengono sei diversi tipi di combinazioni per formare un triggering event, la cui sintassi é rappresentata in tabella:

Notation	Name
$+l$	Belief addition
$-l$	Belief deletion
$+!l$	Achievement-goal addition
$-!l$	Achievement-goal deletion
$+?l$	Test-goal addition
$-?l$	Test-goal deletion

Figura 2.2: Tabella dei possibili triggering event

Eventi per aggiunta o rimozione di belief possono avvenire, ad esempio, quando un agente aggiorna il suo belief set secondo la sua percezione dell'ambiente,

ottenuta normalmente ad ogni ciclo di esecuzione, mentre l'aggiunta di goal accade solitamente come conseguenza dell'esecuzione di altri plan, ma anche a come risultato della comunicazione fra agenti. Diversamente la rimozione di un goal é usata per la gestione del plan failure, ovvero la situazione che si viene a creare quando un plan fallisce o non puó piú continuare.
I plan con test goal nel triggering event, invece, sono usati per il recupero di informazioni che non sono derivabili dai correnti belief dell'agente, e che quindi necessitano dell'esecuzione di alcune operazioni per il recupero dei dati richiesti.

Context

In un sistema come quello in cui solitamente si trovano degli agenti é complicato gestire i cambiamenti di un ambiente dinamico come puó essere quello di un MAS: in primo luogo perché tali variazioni possono implicare l'esecuzione di ulteriori azioni da parte degli agenti (oltre a quelle giá in corso), ma soprattutto perché, all'evolversi dell'ambiente le possibilitá dei vari plan di avere successo (raggiungere i goal) possono aumentare o diminuire. Questo é il perché i sistemi reattivi a pianificazione postpongono il piú possibile l'impegno di un'entitá ad eseguire un corso di azioni per il raggiungimento di un fine, ovvero la scelta di un plan per uno dei vari goal di un agente é fatta appena prima che esso inizi ad agire a riguardo.
Tipicamente un agente ha diversi plan per raggiungere uno stesso goal: il context é usato proprio per determinare quale, fra quelli disponibili, ha piú probabilitá di successo, data la situazione corrente e le informazioni disponibili piú recenti riguardo all'ambiente. Quindi un plan é scelto per l'esecuzione se il context é una conseguenza logica dei belief dell'agente: quando il context di un plan soddisfa questo requisito, esso é detto applicable in quel momento, e sará candidato all'esecuzione. La sintassi per la scrittura di un context consiste in una semplice congiunzione logica di letterali (attraverso gli operatori booleani), in tabella sono riportate le varie combinazioni per quanto riguarda il singolo letterale:

Syntax	Meaning
l	The agent believes *l* is true
`~`*l*	The agent believes *l* is false
`not` *l*	The agent does not believe *l* is true
`not ~`*l*	The agent does not believe *l* is false

Figura 2.3: Tabella sintassi letterali in un context

E' da notare che non credere che l sia falso (ultima riga) non corrisponde a credere che l sia vero, perché l'agente potrebbe semplicemente essere ignorante riguardo l.
Un fatto importante da tenere in mente é che ogni letterale presente nel context va controllato confrontandolo con i belief. Ad esempio, si consideri il seguente head di un plan:

+!prepare(something) : numberOfPeople(N) `&`*stock(something,S)* `&`*S* `>=`*N*

il quale serve per raggiungere il goal di preparare un pasto. Il triggering event é l'aggiunta dell'achievement goal !prepare(something), si supponga sia !prepare(pasta) in questo caso (cioé la variabile something viene unificata con il valore pasta). Prima di eseguire il plan (che potrebbe anche non essere l'unico per questo triggering event) si controlla il letterale numberOfPeople(N): si supponga che nel suo belief set l'agente trovi la riga di codice numberOfPeople(3), a questo punto passa a verificare il literal stock(something,S). Supponiamo anche in questo caso che sia presente un belief tipo stock(pasta,5) (se cosí non fosse si potrebbe aggiungere un test goal per eseguire determinate operazioni per unificare stock(pasta,S)) ad indicare che ci sono le scorte per preparare una pasta per 5 persone. Infine l'agente controlla l'ultima parte del context S >=N, che risulta vera (5 >= 3), sempre secondo ció di cui esso é convinto: a questo punto l'entitá puó iniziare l'esecuzione del body del plan (qui non indicato).

Body

Sequenza di formule (eseguita se il plan risulta applicable) separate da ';' e terminata dal punto, che determinano un flusso di azioni che dovrebbe gestire con successo l'evento che ha innescato il piano. Ad ogni modo, ciascuna formula che compare nel body potrebbe anche non essere una semplice azione che puó essere direttamente eseguita dagli attuatori dell'agente, ma puó anche essere un goal: in questa situazione ci si riferisce a questi con il termine di subgoal, i quali possono far partire l'esecuzione di un nuovo plan. Piú in dettaglio, ci sono sei tipi di formule che possono apparire nel body di un plan:

1. Azioni (actions),
2. Achievement Goal,
3. Test Goal,
4. Note Mentali (mental notes),
5. Azioni interne (internal actions),
6. Espressioni (expressions).

2.1.4 Formule nel Body

Actions

Una delle piú importanti caratteristiche di un agente é la capacitá di agire all'interno dell'ambiente: non c'é da sorprendersi che la maggior parte delle formule che si trovano nel body dei plan siano delle azioni, le quali rappresentano ció che l'agente é in grado di fare.
Dopo aver individuato le azioni che l'agente é in grado di compiere, occorre trovarne delle rappresentazioni simboliche, che serviranno per rifersi a queste all'interno del programma e permettere all'architettura software di interfacciarsi con gli attuatori (meccanismi software o hardware che eseguono le azioni, che non fanno parte dell'interprete AgentSpeak).
Il costrutto per riferirsi simbolicamente alle azioni é dato semplicemente da un predicato fondato (ground predicate), quindi ci si deve assicurare che tutte

le variabili che vi compaiono siano istanziate prima che l'action possa essere eseguita.
L'interprete distingue le azioni dagli altri tipi di predicate in base alla posizione nel plan: infatti se viene riconosciuto nel context un predicato, allora questo sará interpretato come test e quindi verificato rispetto ai belief, mentre nel body si prenderanno tutti i predicati come azioni.
Data l'estraneitá degli attuatori rispetto all'interprete, quando un plan manda in esecuzione un'azione, questo viene sospeso finché essa non é terminata e ha restituito un feedback booleano che comunichi se é stata eseguita o meno. Eseguire una action non significa necessariamente che i cambiamenti previsti avranno luogo: questo dovrá essere confermato da una successiva percezione dell'ambiente.

Achievement Goal

Quella di avere degli obiettivi da raggiungere é un'altra caratteristica chiave per un agente. Il comportamento complesso che un'entitá di questo tipo esibisce richiede che possano essere eseguite piú di semplici azioni: include raggiungere ulteriori obiettivi prima di poter continuare con l'esecuzione (cioé prima che altre azioni possano essere eseguite).
Per tale motivo é possibile specificare nel body di un plan anche degli achievement goal, che dovranno essere raggiunti prima di poter continuare con l'esecuzione del plan. Come detto in precedenza, avere un nuovo obiettivo (aggiunta di un achievement goal) puó portare all'esecuzione di un nuovo plan (che ha come triggering event proprio il suddetto +!g), quindi, come per le azioni, il plan chiamante dovrá essere sospeso finché quello appena chiamato non avrá terminato la sua esecuzione. Questo vale per la dichiarazione di achievement goal con la sintassi giá vista.
Usando una diversa sintassi, cioé con '!!' al posto del singolo punto esclamativo, é possibile specificare degli achievement goal che non sospendano il plan corrente: infatti in alcuni casi si potrebbe volere raggiungere un obiettivo e non avere bisogno di fermare l'esecuzione attuale (si pensi ad azioni eseguibili in parallelo o anche a goal per i quali non interessa quando verranno raggiunti). Per esempio si puó scrivere:

!!at(home);
call(john);

intendendo che si compierá l'azione di chiamare John appena dopo aver aggiunto un nuovo goal di essere a casa: ció significa che l'agente effettuerá la chiamata possibilmente prima di aver cominciato ad agire in qualsiasi modo verso l'obiettivo 'andare a casa', ma che comunque verrá perseguito.

Test Goal

I test goal sono normalmente utilizzati per recuperare informazioni dal belief base, o per controllare se qualcosa che crede l'agente é effettivamente vero durante l'esecuzione del body di un plan.
Qui si potrebbe sollevare un'obiezione sul motivo per il quale non si possa usare il context del piano per raccogliere tutte le informazioni necessarie prima che l'agente inizi l'esecuzione; le ragioni in realtá sono due: una concettuale e

una pratica. In primo luogo il context serve, e dovrebbe essere utilizzato, solo per determinare se un piano é applicabile o meno, mentre in termini pratici si potrebbe aver bisogno di informazioni che variano nel tempo e quindi, per aver conoscenza dei valori corretti, si debba aspettare il piú possibile prima di recuperarle, in modo da avere dei dati piú aggiornati possibile. Quest'ultimo fatto é dovuto alla natura dinamica dell'ambiente peculiare dei MAS.
Occorre precisare un aspetto tecnico: se durante l'esecuzione di un plan un test goal al suo interno fallisce, allora l'intero plan lo segue e viene innescato il meccanismo per la gestione del plan failure. A livello teorico questa non sembra una questione di cosí grande rilievo, ma in Jason quello che succcede dopo del processo descritto merita di essere esposto. L'interpete Jason, infatti, prima di far fallire un plan, tenta di creare un evento, che potrebbe coincidere con un triggering event di test-goal addition per un plan definito dal programmatore (é questo il meccanismo personalizzabile offerto da Jason per il plan failure): se un tale piano (relevant) non esiste, allora il test goal é da considerarsi effettivamente fallito.
E' interessante anche osservare che un test goal potrebbe essere sosituito da un achievement goal equivalente (cioé che va a prelevare le stesse informazioni), ma questo é sconsigliato, in quanto un achievement goal obbliga in ogni caso l'interprete alla generazione di un evento per l'handling dell'achievent-goal addition (che é un meccanismo computazionalmente costoso), laddove un test goal potrebbe risolversi con un semplice istanziamento delle variabili (unificazione) nel plan, che cosí puó riprendere la sua esecuzione immediatamente. Per questa ulteriore ragione ha perfettamente senso l'esistenza e l'utilizzo dei test goal.

Mental Notes

Un'importante considerazione da fare riguardo la programmazione in Jason é la necessitá di conoscere ció che l'agente é in grado di percepire dall'ambiente. Per quanto detto finora si potrebbe pensare che sia compito del programmatore aggiornare il belief set a seconda delle conseguenze delle azioni che hanno avuto luogo come parte del programma: questo non é necessario in Jason, infatti alcuni belief sono acquisiti direttamente (implicitamente) dal sensing dell'ambiente e vengono aggiornati automaticamente dal sistema. La piattaforma inoltre offre un'integrazione per la personalizzazione dell'update.
Nella pratica, comunque, é spesso utile creare dei belief durante l'esecuzione di un plan: in questo caso sono detti mental notes e si distinguono dagli altri grazie all'annotazione [source(self)]. Queste note servono all'agente per ricordarsi ad esempio di un'azione compiuta nel passato (da se stesso o un altro agente) o per avere informazioni riguardo a un compito interrotto per essere ripreso in un secondo momento oppure ancora riguardo una promessa fatta o un impegno preso.
Una questione che nasce con l'uso delle mental notes é quella della 'pulizia' del belief set, infatti queste note vanno rimosse manualmente quando diventano superate o inutili. Jason fornisce due operatori per risolvere il problema: '-' per eliminare una istanza di mental note e '-+' per eliminarne una precedente istanza e aggiungerne una nuova. Tentare di eliminare un belief quando non é presente risulta solamente in un nulla di fatto, non con il failure del plan all'interno del quale si tenta l'operazione. Con gli stessi operatori é possibile manipolare anche i belief che hanno annotazioni diverse da [source(self)], ma bisogna fare molta

attenzione, in quanto questo significa interferire con la percezione dell'ambiente dell'agente o il suo sistema di comunicazione.

Internal Actions

Il primo tipo di formula visto é stato quello delle azioni: queste hanno la caratteristica di modificare l'ambiente esterno, e si puó dire che sono eseguite fuori dalla 'mente' dell'agente. E' utile considerare anche un tipo di operazione che l'agente potrebbe aver bisogno di eseguire all'interno di se (della propria mente) e in un unico ciclo di ragionamento: queste sono le azioni interne.
Per quanto riguarda la sintassi esse si distinguono dalle normali azioni grazie al prefisso '.' e, oltre a quelle fornite di default da Jason, é possibile estendere il linguaggio attraverso l'accesso a legacy code (codice Java scritto dal programmatore) grazie al meccanismo offerto dalla piattaforma.
Le internal actions standard di Jason sono caratterizzate dall' empty library name, ovvero sono contenute in una libreria di cui non é necessario il nome, per cui possono essere chiamate semplicemente con .nomeAzione. Tali azioni implementano operazioni utile alla programmazione BDI, come ad esempio .send che é fondamentale per la comunicazione interagente.

Expressions

Ogni formula nel context e nel body di un plan deve ritornare un valore booleano (anche le azioni interne). Se si mette un'espressione (booleana) all'interno del body di un piano questa viene valutata, e se la condizione risulta falsa allora l'intero piano fallisce.
Occorre fare alcune precisazioni riguardo agli operatori in Jason, infatti:

- == significa uguale, che in questo caso si intede nel senso di perfettamente identico
- /== sta per diverso
- = é l'operatore che serve per controllare se due term possono essere unificati (che non é lo stesso di uguali)
- =.. infine serve per scomporre un literal in una lista (es. p(b,c)[a1,a2] =.. [p, [b,c], [a1,a2]])

2.2 Esempio di programma ad agenti

2.2.1 Descrizione

Si considera il seguente esempio, fornito con la distribuzione di Jason:

Un robot domestico ha l'obiettivo di servire da bere al suo proprietario (continuamente, si immagini che per uno stato di convalescenza debba rimanere costantemente idratato), da cui riceve richieste, al pervenire delle quali va al frigorifero, prende una bottiglia e la porta al proprietario. Oltre a questo il robot dovrá preoccuparsi delle scorte di bevande ed eventualmente ordinare al supermercato la consegna a domicilio.

Il sistema é composto da tre agenti: il robot, il proprietario e il supermercato. Le possibili percezioni che tali agenti possono avere sono:

- at(robot,place) cioé la posizione del robot; per semplicitá si assumono solo due posizioni possibili, cioé fridge e owner, quindi le corrispondenti percezioni (e quindi belief) saranno at(robot, fridge), at(robot, owner) oppure nessuna occorrenza di 'at' nel caso in cui il robot non si trovi in nessuno dei due posti (tragitto fra uno e l'altro)

- stock(drink, N), ovvero per ogni bibita presente la relativa scorta, percepito quando il robot é davanti al frigo.

- has(owner, drink), esprime se il proprietario attualmente ha o meno una bottiglia (non vuota)

Le action che i tre agenti sono in grado di eseguire sono

- per il robot: open(fridge), get(drink), close(fridge), handIn(drink), moveTowards(place), la quale assume che il movimento del robot sia semplificato e gestito dall'hardware (ad esempio con una specie di 'rotaia' in modo da non doversi preoccupare del percorso)

- l'owner ha solo sip(drink) ovvero sorseggiare dalla bottiglia portatagli, mentre anche

- supermarket ha l'unica action deliver(drink,N) con cui si suppone vengano consegnate N bottiglie della bibita specificata

2.2.2 Realizzazione Agenti

Agente Owner

!get(water). //goal iniziale

//plan per l'achieving del goal iniziale
@g +!get(water) : true <-.send(robot, achieve, has(owner,water)).

//plan che verrá eseguito una volta ricevuta la bottiglia
@h1 +has(owner,water) : true <-!drink(water).

//eseguito una volta finita la bottiglia (rimozione belief has)
@h2 -has(owner,water) : true <-!get(water).

//esegue l'azione di bere dalla bottiglia (se ne ha una)
@d1 +!drink(water) : has(owner,water) <-sip(water); !drink(water).

//altro piano per aggiunta goal drink, gestisce il caso in cui sia senza bottiglia
@d2 +!drink(water) : not has(owner,water) <-true.

Il goal iniziale dell'owner é quello di ottenere dell'acqua (!get(water)), per cui quando inizia l'esecuzione l'evento per l'aggiunta di belief +!get(water) é generato ed il plan @g sará innescato (ovvero l'evento appena sollevato é in match

con il triggering event del piano): questo si limiterá ad eseguire l'azione interna .send per comunicare al robot di raggiungere l'achievement goal has(owner, water). In questo modo tale goal verrá aggiunto fra quelli del robot, il quale avrá uno o piú plan per il suo raggiungimento: quando questo avverrá, l'owner se ne accorge attraverso una percezione dell'ambiente, la quale provecherá l'aggiunta del belief has(owner,water)[source(percept)] al belief set. Questo risulterá nella generazione dell'evento corrispondente, il quale innescherá il piano @h1, il quale non verifica nessuna condizione (context=true) e si limita a creare il subgoal !drink(water).L'evento conseguente a questo ha due relevant plan: @d1 e @d2, ma in questo caso solo @d1 é applicable, in quanto nel context é specificato che per la sua esecuzione deve essere verificata (o derivabile) la condizione has(owner,water): @d2 verrá eseguito quando il proprietario terminerá la bottiglia, ovvero quando il belief non sará piú presente. La rimozione di has(owner,water) innescherá il plan @h2, che fará ricominciare ciclicamente tutte le operazioni sopra descritte.

Agente Supermarket

```
//belief iniziale
lastOrderId(1).

//plan per raggiungere il goal "order" per l'agente Ag
+!order(product,qnt)[source(Ag)] : true <- ?lastOrderId(N);
OrderId = N+1;
-+lastOrderId(OrderId);
deliver(product,qnt);
.send(Ag,tell, delivered(Product,qnt,OrderId)).
```

Il belief iniziale rappresenta il numero dell'ultimo ordine, mentre il plan gestisce la richiesta da parte dell'agente Ag (che in questo caso sará il robot) di una certa quantitá qnt del prodotto product. Tale piano come prima cosa recupera l'id dell'ultimo ordine dal belief set attraverso un test goal; una volta recuperato, lo si incrementa e si memorizza il valore aggiornato, eliminando contemporaneamente l'informazione vecchia (attraverso l'operatore -+). Fatto questo l'agente consegna il prodotto ordinato tramite l'azione deliver ed infine comunica all'agente che ha richiesto questo servizio che la consegna é stata effettuata usando l'azione interna .send.

Agente Robot

```
//belief iniziale: é disponibile dell'acqua in frigo
available(water,fridge).

//due piani per la gestione del goal addition !has(owner,water) in due contesti
diversi

@p1 +!has(owner,water) : available(water,fridge)
<-!at(robot,fridge);
open(fridge);
get(water);
close(fridge);
```

!at(robot,owner);
handIn(water);
?has(owner,water).

@p2 +!has(owner,water): not available(water,fridge) <- *.send(supermarket, achieve, order(water,5));*
!at(robot,fridge).

//due plan per far il movimento del robot
@m1 +!at(robot,P) : at(robot,P) <- *true.*

@m2 +!at(robot,P) : not at(robot,P) <- *moveTowards(P);*
!at(robot,P).

//Gestione della consegna da parte del supermercato
@d1 +delivered(water,qnt,OrderId)[source(supermarket)]: true
<- *+available(water,fridge);*
!has(owner,water).

//aggiornamento del belief riguardante le scorte

@s1 +stock(water,0) : available(water,fridge) <- *-available(water,fridge).*

@s2 +stock(water,N): N ¿ 0 &not available(water,fridge)
<- *-+available(water,fridge).*

Il robot mantiene un belief riguardo alla presenza di acqua nel frigo: inizialmente c'é disponibilitá e quindi il belief sará presente come entry inziale del belief set con available(water,fridge). Per il robot é importante mantenere questa informazione, in quanto essa non puó essere aggiornata in ogni circostanza: lo si puó fare solo quando il robot si trova davanti al frigo e percepisce l'ambiente, il che corrisponde all'aggiunta (automatica) del belief +stock(water,N). Per gestire tale evento sono presenti i plan @s1 e @s2, che aggiungono o rimuovono il belief riguardo alla disponibilitá d'acqua in base al valore di N: la presenza o meno di questo nel belief set determinerá il comportamento del robot per quanto riguarda la gestione delle richieste d'acqua arrivategli dal proprietario. Per questo motivo esistono i plan @p1 e @p2 con lo stesso triggering event (achievement goal addition +!has(owner,water)) ma con diversi context per scegliere come agire in base allo stato delle scorte d'acqua. Se non c'é piú acqua in frigo, il robot procede ad ordinarla al supermercato (5 bottiglie alla volta) comunicando all'agente Supermarket di raggiungere il goal order(water,5): grazie al plan @d1 il robot 'si accorgerá' dell'avvenuta consegna, registrerá tra i suoi belief la rinnovata disponibilitá idrica e ripartirá con il suo compito di servire l'owner. Inoltre sono presenti @m1 e @m2 per il movimento del robot: é da notare che se il robot si trova giá nella postazione richiesta il plan non compie nessuna azione (body composto solo da true.).

Conclusioni

Grazie alcune interessanti caratteristiche della programmazione orientata agli agenti, negli ultimi anni si é visto un incremento per quanto riguarda l'attivitá di ricerca su questa tecnologia, e questo fatto é testimoniato dal sempre crescente numero di linguaggi, tool e piattaforme per lo sviluppo di MAS.
Le tecniche multi-agente, infatti, rendono piú semplice la distribuzione delle applicazioni, in quanto i vari agenti di un sistema potrebbero trovarsi in diverse macchine: il tipo comportamento di queste entitá permette di minimizzare l'intervento umano, visto che, se ben progettato, un agente puó decidere per conto suo e adattarsi al sistema.
La AOP ha il potenziale per fornire la migliore soluzione in diversi scenari, soprattutto quando c'é la possibilitá di impostare dei MAS in parallelo in ambienti distribuiti.
La tecnologia, inoltre, é semplice da personalizzare attraverso nuove componenti e servizi: un ulteriore motivo per cui molto probabilmente vedrá interessanti sviluppi nei prossimi anni.

Bibliografia

1. 'Agent Oriented Programming' di Yoav Shoham
2. 'Principles of Agent-Oriented Programming' di André Filipe de Moraes Batista, Maria das Graças Bruno Marietto, Wagner Tanaka Botelho, Guiou Kobayashi, Brunno dos Passos Alves, Sidney de Castro e Terry Lima Ruas
3. 'Progetto di sistemi multi-agente, modello BDI' di Andrea Omicini, Ivan Campolieti
4. 'BDI Agent Programming in AgentSpeak Using Jason' di Rafael H. Bordini e Jomi F. Hubner
5. 'AgentSpeak(L): BDI Agents speak out in a logical computable language' di Anand S. Rao
6. 'Programming Multi-Agent Systems in AgentSpeak using Jason' di Bordini, Hubner e Woolridge