# Agenti Intelligenti

Lorenzo Sciandra

Anno Accademico 2021/2022

# Contents

# List of Figures

LIST OF FIGURES

# List of Tables

## 0.1 Premessa

Questa è semplicemente la versione aggregata, non modificata, degli appunti che ho preso durante le lezioni del corso nell'anno accademico 2021-22. Dato che erano pensati originariamente solo per uso personale è possibile che contengano typos così come errori grammaticali.

# Part I

# Matteo Baldoni

# Chapter 1

# Introduzione agli Agenti Intelligenti

## 1.1 Trends della computazione

Cinque *trends* hanno caratterizzato la storia della computazione e ritroviamo negli agenti:

- **Ubiquità**: la continua riduzione dei costi di elaborazione ha permesso di introdurre capacità di calcolo in luoghi e dispositivi una volta impensabili. Man mano che la capacità di elaborazione si diffonde, elementi sofisticati (e intelligenti) diventano onnipresenti;

- **Interconnessione**: i sistemi informatici ad oggi sono collegati in rete in grandi sistemi distribuiti. Dato che i sistemi distribuiti e concorrenti sono diventati la norma, l'informatica viene sempre più ritratta principalmente come un *processo di interazione*;

- **Intelligenza** e **Delega**: la complessità dei compiti che siamo capaci di automatizzare e delegare ai computer è cresciuto costantemente. I computer fanno di più per noi senza un nostro intervento;

- **Human Orientation**: ci si è spostati verso concetti e metafore che sono più vicine al modo con cui noi comprendiamo e vediamo il mondo. I programmatori concettualizzano e realizzano software in termini di livello superiore, più orientato all'uomo, mediante l'uso di *astrazioni*.

Tutto questo implica la necessità di costruire sistemi informatici in grado di agire efficacemente, in modo indipendente, per nostro conto. Al fine di agire in modo che rappresentino al meglio i nostri interessi, si hanno sistemi che possono **cooperare** e **raggiungere accordi** o **competere** con altri sistemi che hanno interessi diversi.

## 1.2 Concetti fondamentali

Le definizioni presentate sono frutto della convenzione di usare l'essere umano come metafora di un sistema di computazione ideale.


Figure 1.1: Rapporto Agente-Ambiente

> Un **agente** è un sistema di computazione, un sistema informatico, capace di agire autonomamente in un qualche ambiente con il fine di raggiungere gli obiettivi di un utente o proprietario che l'ha progettato.

La principale caratteristica degli agenti è la loro autonomia, ovvero la capacità di agire in maniera indipendente, di esibire controllo sul proprio stato interno.

> Un **sistema multiagente** consiste in una serie di agenti, che interagiscono l'uno con l'altro. Nel caso più generale, gli agenti agiscono per conto di utenti con differenti obiettivi e motivazioni. Per interagire con successo, richiedono la capacità di coordinarsi e quindi cooperare e negoziare tra di loro, in modo analogo a quanto fanno le persone.

Viene indicata come **micro visione** il problema di progettare un agente capace di agire in modo indipendente, autonomo, che possa svolgere con successo i compiti che gli deleghiamo. Viene invece detta **macro visione** la progettazione di società di agenti capaci di interagire tra di loro.

## 1.3 Cosa s'intende per agente intelligente?

Per prima cosa dobbiamo sottolineare come il paradigma ad agenti metta sullo stesso piano l'agente e l'ambiente nel quale opera, come strumento per capire se le sue azioni siano o meno sperabili.

Devo infatti sempre tenere a mente l'ambiente in cui agisce l'agente per sapere cosa percepire e come modificarlo per ottenere gli obiettivi prefissati.

> Un **agente intelligente** è un sistema di computazione capace di eseguire azioni autonome in modo *flessibile* in un ambiente.

Con il termine *flessibile* nella definizione s'intende:

1. **Reattivo**: l'agente deve mantenere una costante interazione con l'ambiente e rispondere ai cambiamenti che occorrono su di esso, in tempo perché la risposta sia utile. Se

l'ambiente è *statico/fisso* non è necessario preoccuparsi di esso, mentre se è dinamico, come il mondo reale, è necessario considerare la possibilità che l'esecuzione di azioni non abbiano successo, esattamente come è necessario chiedersi se valga la pena di eseguire una certa azione. Tutto questo sapendo che saremo generalmente in un contesto di informazione incompleta. Un possibile **agente reattivo** può essere implementato tramite semplici regole condizionali del tipo: **if** *head* **then** *body*;

2. **Proattivo**: l'agente deve tentare di raggiungere gli obiettivi, non sono agenti solo guidati dagli eventi, ma prendono l'iniziativa, sono in grado di riconoscere le opportunità. Per far questo gli agenti devono mantenere uno **stato** che contiene due tipi di conoscenza:

   - informazioni su come l'ambiente evolva autonomamente;
   - informazioni su come le proprie azioni impattano sull'ambiente.

   L'**agente proattivo** necessita anche di informazioni sull'obiettivo in modo che possa agire per raggiungerlo. Nel farlo deve essere in grado di ragionare sui piani;

3. **Sociale**: l'agente deve avere l'abilità di interagire con altri agenti (anche umani) attraverso un qualche tipo di linguaggio di comunicazione e cooperare con essi. Il mondo reale è spesso complesso e non possiamo raggiungere i nostri obiettivi senza l'aiuto di altri.

La reattività può essere in conflitto con la proattività, progettare un agente in grado di bilanciare bene questi due aspetti è ancora un problema di ricerca aperto. Per distinguere un agente intelligente da un semplice programma Franklin e Graesser propongono la seguente definizione:

> Un **agente autonomo** è un sistema situato in un ambiente che può percepire ed agire su di esso nel tempo con il fine di perseguire una propria agenda e cosi' facendo influire nelle successive percezioni.

### 1.3.1 Triangolo della computazione di Meyer

Un ingrediente cruciale per produrre software di qualità è la **modularizzazione** dell'architettura software che permette di ottenere: *correttezza, riusabilità, robustezza ed estensibilità* del software. Meyer definisce la computazione come: *l'uso di certi **processi** per applicare determinate **azioni** a degli **oggetti** stabiliti*. A seconda di quanto siano rilevanti queste 3 componenti abbiamo a che fare con diverse modularizzazioni e quindi diversi triangoli di Meyer:

Il paradigma basato su agenti, nel nostro caso, si basa su due astrazioni di prima classe:

- **Agente** che è la forza processo con obiettivo da raggiungere;
- **Ambiente** che è la forza dato/oggetto. L'ambiente è l'insieme dei dati da elaborare e fornisce le condizioni circostanti per l'esistenza degli agenti. Questo media l'interazione tra gli agenti e l'accesso alle risorse. In natura non esite il concetto di interazione senza quello di ambiente.


- **process(or)**: a physical CPU, a process or a thread
- **action**: operations making up the computation (machine language operations up to subroutines)
- **object**: data structures to which the actions apply (computation-dependent or files, databases, etc.)

Figure 1.2: Meyer Triangle

In questo paradigma gli agenti sono autonomi, hanno un proprio ciclo deliberativo guidato dai goals e sono situati/posizionati in un ambiente che risulta un elemento programmabile tanto quanto lo sono loro. Gli agenti sono l'evoluzione dei processi, mentre l'ambiente è l'evoluzione del concetto di oggetto.

In questa rappresentazione i processi codificano il *goal*, i *belief* o credenze dell'agente sono il ponte tra ambiente e agente. Le intenzioni sono usate per determinare le azioni e quindi il piano che devo eseguire per raggiungere il goal. I piani rappresentano quindi la decomposizione funzionale del goal in accordo con le azioni applicabili di cui dispongo. Gli *artefatti* sono dati con il loro ciclo di vita.

## 1.4 Architetture per sistemi ad agenti

Presentiamo le architetture in accordo con il periodo storico durante le quali sono state proposte:

- 1956 – 1985: incapsulamento di algoritmi di risoluzione, deduzione, ... dentro agli agenti, basati quindi sul **ragionamento simbolico/logico**. Il classico approccio per costruirli consiste nel vederli come casi particolari di sistemi basati sulla conoscenza. L'architettura di un agente deliberativo contiene una esplicita rappresentazione o **modello simbolico** dell'ambiente e prende decisioni attraverso un **ragionamento simbolico**. In questo contesto i problemi da affrontare sono:

  1. **Il problema della trasduzione**: ovvero il problema della traduzione del mondo reale, dell'ambiente, in una descrizione simbolica accurata in tempo perché possa essere utile;
  2. **Il problema della rappresentazione/ragionamento**: ovvero il problema di come rappresentare simbolicamente le informazioni su entità e processi complessi del mondo reale, e come fare in modo che gli agenti ragionino con queste informazioni in tempo perché i risultati possano essere utili.

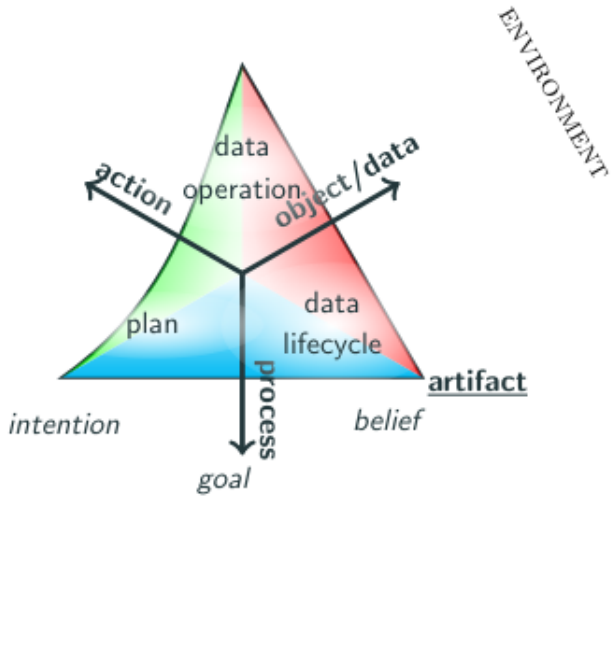



Figure 1.3: Rappresentazione Agente con triangolo Meyer

Nessuno di questi problemi sembra essere facilmente risolvibile, servono alternative. Il problema risiede nella generale complessità degli algoritmi di manipolazione simbolica la maggior parte dei quali risultano infatti **altamente intrattabili**. Il decision making basato sulla logica del primo ordine è **indecidibile**, mentre con la logica proposizionale, nel peggiore dei casi, risulta **co-NP-completo**. Le tipiche soluzioni possono essere quelle di indebolire la logica o utilizzare una rappresentazione simbolica non derivata dalla logica o spostare l'enfasi dal ragionamento al tempo di esecuzione al tempo di progettazione, usando piani già precompilati e perfetti per situazioni specifiche;

- 1985 – *oggi*: dato che l'approccio logico era si corretto, ma lento e a volte non completo, nascono soluzioni basate sulla **reattività** per garantire maggiore velocità;
- 1990 – *oggi*: architetture ibride, nate per combinare il meglio delle architetture basate sul ragionamento logico e quelle reattive.

# Chapter 2

# Agenti Razionali

## 2.1 Agenti come sistemi intenzionali

Quando si spiega l'attività umana si può dire che gli uomini compiano azioni come conseguenza di qualcosa che è creduto/desiderato. Questo riguarda l'uso di una **psicologia popolare** che illustra il comportamento umano attraverso l'attribuzione di **attitudini** come *credere, volere, sperare* e così via. Le attitudini in questo contesto sono chiamate **nozioni intenzionali**. Spesso sapere le intenzioni di un agente ci permette di prevedere alcuni suoi comportamenti, se sappiamo infatti che ci sono dei desideri/credenze da parte di un agente possiamo aspettarci che esso compia determinate azioni al fine ottenere ciò che è desiderato. Il filosofo Daniel Dennett ha coniato il termine **sistema intenzionale** per descrivere entità "*il cui comportamento può essere previsto dal metodo di attribuzione di credenze, di desideri e di acume razionale*".
McCarthy ritiene appropriato ed utile, in alcune occasioni, attribuire una posizione intenzionale a sistemi informatici. Dire che un sistema ha adottato la mia intenzione significa dire che ha svolto il compito che io desideravo, questo però non gli attribuisce consapevolezza di averlo fatto. L'agente infatti non ci comprende, non modifica le sue azioni per portare a termine le nostre intenzioni.

> **Shoham**: *più i sistemi di calcolo diventano complessi, più hanno bisogno di astrazioni e metafore potenti per spiegare il loro funzionamento – le spiegazioni di basso livello diventano impraticabili. Le posizioni intenzionali sono una tale astrazione.*

L'idea è quindi associare una posizione intenzionale a sistemi complessi da capire da un punto di vista pratico/meccanico, permettendo di introdurre astrazioni/metafore. L'uso dell'astrazione risulta fondamentale nell'informatica ed è infatti applicata in più contesti come la programmazione o un sistema operativo. Le nozioni intenzionali sono quindi astrazioni, che forniscono un modo semplice e familiare per descrivere, spiegare e prevedere il comportamento di sistemi complessi. Questo ci permetterà di programmare un agente ad alto livello, a livello intenzionale, consci del fatto che dovrà poi essere compilato in un livello più basso per essere eseguito.

## 2.2 Practical Reasoning

Il **practical reasoning** è il ragionamento sulle azioni, sul processo di capire cosa fare. Questo è il meccanismo che mi permette di passare dai desideri ai fatti. Il practical reasoning si distingue dal *theoretical reasoning* in quanto quest'ultimo ragiona sulle credenze, mentre il primo sulle azioni date le intezioni/credenze. Il practical reasoning consiste di due attività:

- **deliberazione** (*deliberation*): decidere quale stato di cose vogliamo raggiungere. Il suo output sono le itenzioni;
- **pianificazione** (*means-ends reasoning – planning*): decidere come raggiungere questi stati di cose. Il suo output sono i piani.

Dopo aver ottenuto il piano l'agente deve ovviamente eseguirlo. La deliberazione e la pianificazione sono due processi computazionali e come tali li ipotizziamo in contesti di risorse limitate (come il tempo). Ossia ci aspettiamo che l'agente decida cosa fare e come farlo in tempo ragionevole affinchè sia ancora utile. Vorremmo quindi algoritmi efficienti e che terminino sempre. E' qui che entra in gioco il concetto di proattività come favorita dalle intenzioni, cioè sono proprio le intenzioni quelle che portano l'agente all'azione. Bratman vede nelle intenzioni e non nei desideri ciò che fa scattare il practical reasoning, i desideri possono essere contrastanti mentre le intenzioni no. Questo sottolinea le intenzioni come persistenti in modo ragionevole all'esecuzione di altre azioni e vincolanti per il practical reasoning. Nel practical reasoning non posso deliberare più intenzioni in contrasto, ma posso averne più coerenti e consistenti tra loro. Possiamo quindi dire che:

- le intenzioni pongono problemi agli agenti che devono determinare modi per realizzarle;
- le intenzioni forniscono un “filtro” per l’adozione di altre intenzioni, che non devono entrare in conflitto;
- gli agenti monitorano il successo delle loro intenzioni e sono inclini a riprovare se i loro tentativi falliscono;
- gli agenti credono che le loro intenzioni siano possibili;
- gli agenti credono che porteranno avanti le loro intenzioni;
- in determinate circostanze, gli agenti credono che realizzeranno le loro intenzioni;
- gli agenti non devono avere intenzione su tutte le conseguenze delle loro intenzioni.

Quest’ultimo problema è noto come problema dell'**effetto collaterale** o problema del package deal.

## 2.3 Costruzione di un agente - Agent Control Loop

Andiamo ora a definire il programma di controllo di un agente che sfrutta il practical reasoning. In particolare lo scriveremo come un insieme di chiamate di funzioni, ognuna delle quali per essere implementata può richiedere enormi conoscenze, dato che risiedono in aree diverse della letteratura in AI. E' importante che il tempo impiegato dall'agente per deliberare le intenzioni e per ragionare sulle azioni per costruire il piano sia breve, dato che il mondo potrebbe cambiare nel mentre e ciò che viene calcolato potrebbe non essere più valido. Avremo i seguenti istanti temporali: $t_0$ quando inizia a deliberare, $t_1$ quando inizia a pianificare e $t_2$ quando inizia ad eseguire, che definiscono:

$$t_{deliberate} = t_1 - t_0 \qquad\qquad t_{planning} = t_2 - t_1$$

Supponiamo inoltre che la deliberazione sia **ottimale**, cioè, se seleziona un'intenzione da raggiungere, allora questa è la migliore per l'agente poichè ne massimizza l'utilità attesa. Abbiamo quindi che l'agente al tempo $t_1$ trova la migliore intenzione per il tempo $t_0$. L'agente corre quindi il rischio che l'intenzione selezionata non sia più ottimale nel momento in cui l'agente l'ha deliberata: **calculative rationality**. Questo ragionamento può essere ripetuto anche per la pianificazione. Il comportamento complessivo sarà quindi ottimale in una delle seguenti circostanze:

1. quando il tempo impiegato per deliberare e pianificare è incredibilmente piccolo;
2. quando il mondo è garantito rimanere **statico**;
3. quando un'intenzione ottimale se raggiunta al tempo $t_0$ è **garantita ottimale** fino a $t_2$.

Non sempre queste condizioni sono possibili da soddisfare, per farlo si cerca generalmente un buon compromesso attraverso l'uso di euristiche in base alla situazione. Uno degli approcci più gettonati è quello di precompilare dei piani, dato che algoritmi di pianificazione real-time possono essere troppo lenti. Presentiamo una prima versione dell'agente:

$B$ è un insieme di credenze (*beliefs*) soggette a revisione ad ogni iterazione andando ad usare il risultato $\rho$ della percezione corrente. Con ogni ciclo ci si aspetta infatti che almeno un'intenzione venga soddisfatta ed è quindi necessario osservare ogni volta come è cambiato il mondo per valutare cosa rivedere. L'implementazione della funzione `brf(`$B, \rho$`)` (*belief revision function*) non è semplice perchè se pensiamo alle credenze come insieme di predicati logici, questa deve essere **non monotona**, ossia deve essere in grado di confutare fatti precedentemente creduti. Con $I$ (*intentions*) indichiamo invece le intenzioni dell'agente che vengono calcolate a partire dai desideri $D$ (*desires*) e dalle credenze mediante un filtro. Da qui si può capire perchè tali agenti vengano chiamati $BDI$. Nel codice la deliberazione è stata quindi scomposta in due passaggi:

- **Option generation** in cui l'agente genera un insieme di possibili alternative o desideri mediante la funzione `options(·)` a partire dalle credenze e dalle intenzioni precedenti;
- **Filtering** in cui l'agente sceglie tra alternative che competono e si impegna a raggiungerle. Quando un'opzione è restituita da `filter(·)` viene scelta dall'agente come

**Agent Control Loop Version 3**:

$B := B_0$;
$I := I_0$;
**while** *true* **do**
  get next percept $\rho$;
  $B := brf(B, \rho)$;
  $D := options(B, I)$;
  $I := filter(B, D, I)$;
  $\pi := plan(B, I)$;
  $execute(\pi)$

Figure 2.1: Agent Control Loop versione base

intenzione, verso cui ne fa un **commitment**. Il commitment implica persistenza temporale, un'intenzione infatti, una volta adottata, non dovrebbe immediatamente essere abbandonata.

Per capire come un agente si impegna verso le sue intenzioni introduciamo il concetto di **Commitment Strategies** come meccanismo che un agente usa per determinare quando e come un'intenzione possa essere abbandonata. Ci sono diverse strategie:

- **Blind commitment**: un agent blind committed continuerà a mantenere un'intenzione fino a quando non crederà che l'intenzione sia stata effettivamente **raggiunta**. Ci si può riferire anche con **fanatical commitment** che non ha accezione negativa;
- **Open-minded commitment**: un agente di questo tipo manterrà un'intenzione fintanto che la **crede possibile**. Il problema di questa strategia è la credenza dell'agente che può essere diversa dalla realtà dei fatti, poichè dipende fortemente dalla sua percezione dell'ambiente;
- **Single-minded commitment**: un agente single-minded committed continuerà a mantenere un'intenzione finché non ritiene che l'intenzione sia stata **raggiunta** o che **non sia più possibile raggiungerla**. Anche qui il problema risiede nella capacità di dimostrare, di dare prova che non sia più possibile raggiungere l'intenzione.

Tra queste strategie non vi è una migliore, in maniera assoluta, delle altre, ma vanno applicate a seconda del dominio/contesto. Per ora il nostro agente è fanatico poichè quando determina un piano lo porta sempre a termine completamente. Il ciclo di controllo dell'agente è inoltre **overcommitted** ossia è impegnato in maniera eccessiva sia nei fini (intenzioni) che nei mezzi (piano), dato che non li rimette mai in discussione. Iniziamo ad introdurre la possibilità di ripianificare se necessario e di riconsiderare le intenzioni rispetto alle nuove percezioni che possiamo avere dopo l'esecuzione di un'azione del piano:

**Agent Control Loop Version 7:**

```
B := B0;
I := I0;
while true do
    get next percept ρ;
    B := brf(B, ρ);
    D := options(B, I);
    I := filter(B, D, I);
    π := plan(B, I);
    while not empty(π) or succeeded(I, B) or
      impossible(I, B) do
        α := hd(π);
        execute(α);
        π := tail(π);
        get next percept ρ;
        B := brf(B, ρ);
        if reconsider(I, B) then
            D := options(B, I);
            I := filter(B, D, I)
        if not sound(π, I, B) then
            π := plan(B, I)
```

Figure 2.2: Agent Control Loop versione finale

Il metodo `sound(`$\pi, I, B$`)` è semplice ed efficiente, fa solamente un check della correttezza del piano con le intenzioni e le credenze correnti. E' molto meno complesso della pianificazione e serve infatti per evitare di farla ogni volta che è stata compiuta un'azione del piano. Il piano potrebbe non essere più corretto non solo a causa del piano stesso, ma il nuovo contesto potrebbe non rendere più valido il fine determinato dalle intenzioni. Anche in questo caso infatti si permette una ridefinizione delle intenzioni se si supera il controllo `reconsider(`$I, B$`)`. Con il controllo che eseguiamo per il while interno possiamo notare come l'agente sia adesso di tipo single-minded, non più fanatico, poichè termina l'esecuzione se le intenzioni hanno avuto successo o se sono diventate impossibili da soddisfare. La `reconsider(` $\cdot$ `)` è una componente di controllo di meta-livello ed, esattemente come per la `sound(` $\cdot$ `)`, ha un costo molto inferiore al processo deliberativo complessivo. Kinny e Georgeff investigarono in maniera sperimentale l'efficacia delle strategie di riconsiderazione delle intenzioni e scoprirono che dipendono dal **dinamismo dell'ambiente** rappresentato dal tasso di cambiamento del mondo $\gamma$. Si possono definire due strategie opposte di riconsiderazione:

- **Bold agents** (*audaci*): agenti che non si fermano mai a riconsiderare le intenzioni;
- **Cautious agents** (*cauti*): agenti che si fermano a riconsiderare le intenzioni dopo ogni azione.

Se $\gamma$ è basso gli agenti audaci fanno meglio rispetto a quelli cauti, viceversa se $\gamma$ è alto.

### 2.3.1 Procedural Reasoning System

Il **PRS**, descritto in Linguaggi e Architetture per Agenti BDI 13, fu la prima architettura per agenti sviluppata che includeva il paradigma *belief-desire-intention*. Al fine di velocizzare in PRS gli agenti non pianificano, ma utilizzano una libreria di piani precompilati. I piani sono costruiti (manualmente) dal programmatore e sono ottimi in un certo dominio/contesto specifico. Essi sono caratterizzati da:

- **goal**: post-condizione del piano, ciò che ottengo dopo l'esecuzione;
- **contesto**: pre-condizione del piano, necessaria affinchè si possa eseguire il piano;
- **corpo**: sequenza delle azioni da eseguire. In PRS le azioni sono complesse, sono dei programmi contenti se necessario ulteriori goals, visti come sottochiamate ad altre funzioni.

Quando un agente inizia la sua esecuzione, il goal iniziale è inserito in uno stack: **lo stack delle intezioni**, che contiene tutti i goals in attesa di essere soddisfatti. L'agente cerca tra i suoi piani per vedere quali hanno post-condizione che coincide con goal in cima allo stack. Tra tutti questi piani seleziona quelli che hanno le precondizioni soddisfatte dall'insieme dei beliefs correnti. Questi formano l'insieme delle **possibili opzioni** dell'agente. A questo punto il processo di selezione tra i diversi piani è la deliberazione ottenuta mediante l'utilizzo di un piano di meta-livello. Effettuata la scelta del piano, la sua esecuzione può portare ulteriori goals nello stack delle intenzioni e così via. Le azioni atomiche potranno essere eseguite direttamente dall'agente. Se un piano fallisce, l'agente può selezionare un nuovo piano tra quelli candidati.

AgentSpeak(L) 5 è il linguaggio di programmazione per agenti BDI introdotto da Rao nel 1996, che propone di colmare il gap tra specifica teorica ed implementazione di un agente BDI. Jason 6 è la prima significativa implementazione di AgentSpeak(L) realizzata in Java da Hubner e Boldrini.

# Chapter 3

# Comunicazione Tra Agenti

## 3.1 Introduzione

Siamo in un contesto di sistema distribuito di agenti poichè le soluzioni centralizzate sono spesso impraticabili in quanto i sistemi e i dati possono appartenere ad organizzazioni indipendenti. Le informazioni coinvolte sono quindi distribuite e risiedono in sistemi informativi di grosse dimensioni, complessi ed indipendenti. Ci sono quattro tecniche principali per affrontare la "dimensione" e la "complessità" di un sistema informativo:

1. modularità;
2. distribuzione;
3. astrazione;
4. intelligenza.

I sistemi multiagenti possono essere considerati come parte dell'approccio di **intelligenza artificiale distribuito** (DAI) che prevede l'utilizzo di tutte le precedenti tecniche.

> I sistemi multiagente sono quindi il modo migliore per caratterizzare o progettare sistemi distribuiti di computazione.

Nella progettazione di un tale sistema vogliamo garantire sia un **disaccoppiamento** tra gli agenti, ossia fare in modo che nessun agente debba dipendere da altri sia una loro **forte coesione**, ossia fare in modo che ogni agente sia composto solo da ciò che gli permetta di ottenere i propri obiettivi. Per lo sviluppo di soluzioni globali e distribuite gli agenti necessitano quindi di essere eseguiti in maniera autonoma e sviluppati indipendentemente. Gli agenti si coordinano e cooperano per la soluzione di problemi, condividendo capacità e lavorando in parallelo, affrontando possibili errori attraverso la ridondanza e rappresentando molteplici punti di vista ed esperienze.

## 3.2 Comunicazione

Gli agenti operano in ambienti progettati per essere aperti e distribuiti per permettere l'interazione, attraverso la **comunicazione**, con agenti eterogenei attraverso un **linguaggio comune** in accordo tra le parti. A tal fine gli ambienti multiagente devono fornire un'**infrastruttura** specifica per la comunicazione e l'utilizzo di protocolli.

> Gli agenti comunicano con il fine di raggiungere i loro obiettivi o quelli della società/sistema in cui esistono.

Le infrastrutture includono:

- **protocolli di comunicazione**: per permettere agli agenti di comprendere i messaggi scambiati, anche attraverso l'utilizzo di ontologie. Questi protocolli specificano come comporre un messaggio in modo che sia comprensibile da altri agenti. Un protocollo di comunicazione potrebbe includere i seguenti tipi di messaggio:
  - proposta di esecuzione di un'azione;
  - accettare una proposta di esecuzione di un'azione;
  - rifiutare una proposta di esecuzione di un'azione;
  - ritirare una proposta di esecuzione di un'azione;
  - esprimere disaccordo rispetto una proposta di esecuzione di un'azione;
  - proporre una differente esecuzione di azione;
- **protocolli di interazione**: per permettere agli agenti di svolgere conversazioni, ossia scambi strutturati di messaggi, come comporre i messaggi per costruire un corretto dialogo.

La comunicazione permette agli agenti di **coordinare** le loro azioni e i loro comportamenti. Questa abilità è sia parte della percezione (ricevere messaggi) che parte delle azioni (inviare messaggi). In un sistema multi-agente questi hanno bisogno di interagire/coordinarsi per ottenere i propri obiettivi. La coordinazione può essere:

- **cooperazione**: se è una coordinazione tra agenti non antagonisti;
- **negoziazione**: se è coordinazione tra agenti antagonisti, competitivi o semplicemente self-interested.

### 3.2.1 Agent Communication Languages

A differenza del paradigma ad oggetti, qua c'è un disaccoppiamento forte, un agente $A_1$ può chiedere tramite ad un messaggio all'agente $A_2$ di eseguire un'azione, ma è quest'ultimo che prende la decisione e risponderà in base ai suoi obiettivi: *sì / no / forse*. Non c'è alcun tipo di controllo di $A_1$ su $A_2$. Quello che possono fare è eseguire **azioni comunicative** al fine di influenzare gli altri agenti in modo opportuno. Per modellare la comunicazione tra

agenti ci si rifà alla comunicazione tra esseri umani e alla teoria nota come **speech act theory** proposta dai filosofi Austin e Searle. Questa teoria considera qualsiasi atto comunicativo come un'azione che ha come effetto la possibile modifica di stati mentali (*credenze, desideri, obiettivi, ecc...*) di chi è coinvolto nella comunicazione. Un atto comunicativo ha tre aspetti:

- **Locuzione**: l'atto fisico compiuto da chi parla, la produzione di enunciati grammaticali;
- **Illocuzione**: il significato inteso dell'enunciato, quello che il parlante desidera esprimere;
- **Perlocuzione**: l'azione che risulta dalla locuzione.

La speech act theory usa il termine **performativa** per identificare la forza illocutoria dell'enunciato. Le condizioni per il successo dell'atto comunicativo proposte da Austin sono:

- deve esistere una procedura convenzionale accettata per la performativa, le circostanze e le persone coinvolte devono essere come specificato nella procedura;
- la procedura deve essere eseguita correttamente e completamente;
- l'atto deve essere sincero e la comprensione deve essere completa. La sincerità è fondamentale per evitare possibile ambiguità.

Cohen e Perrault forniscono una semantica agli atti comunicativi utilizzando le tecniche sviluppate nell'ambito dell'intelligenza artificiale e della pianificazione. In particolare, viene usata la pianificazione mediante precondizioni ed effetti delle azioni (*atti comunicativi*) in termini di stati mentali, per pianificare il dialogo. Per far questo si parla di ACL come strumento per fornire agli agenti i mezzi per scambiarsi informazioni e conoscenza. Un ACL è solitamente ad un livello più alto rispetto ai soliti strumenti di comunicazione per i sistemi distribuiti come chiamate di procedure remote. Un messaggio in ACL descrive uno stato desiderato attraverso un linguaggio dichiarativo, spesso in termini di stati mentali. Come linguaggio ACL vediamo: KQML e FIPA ACL.

### Knowledge Query and Manipulation Language

KQML è un linguaggio d'interazione ad alto livello usato dentro Jason come struttura dei messaggi ed indipendente da:

- il meccanismo di trasporto (*tcp/ip, RMI, ecc...*);
- il linguaggio in cui è espresso il contenuto (*KIF (linguaggio logico basato su FOL per esprimere proprietà di oggetti in KB), Prolog, ecc...*);
- l'ontologia utilizzata.

Parte fondamentale di un messaggio KQML è la specificazione del tipo di messaggio o **performativa**, come *achieve, ask-one, reply, tell, sorry, ecc...* Uno dei problemi di KQML che ne causò la scomparsa fu il fatto che le varie performative non furono mai standardizzate e questo portò alla definizione di diversi set di performative usate che a volte non coprivano tutte le possibili esigenze/azioni. La sintassi di un messaggio KQML si basa su una notazione a lista simile a quella utilizzata in Lisp. Consiste nel tipo di performativa seguita da un numero di coppie "parola-chiave / valore":

```
(ask-one                    ← performativa
    :sender joe             <--- il mittente della performativa
    :receiver stock-server  <--- il destinatario della performativa
    :reply-with ibm-stock   <--- se il mittente si aspetta una risposta, rappresenta l'identificatore della risposta
    :language LPROLOG       ← il linguaggio di rappresentazione del contenuto
    :content (PRICE IBM ?price) <--- il contenuto del messaggio
    :ontology NYSE-TICKS    <--- l'ontologia assunda dal parametro :content
)
```

Figure 3.1: Esempio di messaggio con KQML

La sequenza di atti comunicativi in KQML costituisce un **dialogo** fra i 2 interlocutori, inoltre è possibile inserire in *:content* un altro messaggio KQML. Una delle novità introdotte da KQML sono i **facilitatori della comunicazione**, agenti intermediari che durante il dialogo si occupano di instradare i messaggi agli opportuni agenti. Questo meccanismo centralizza la comunicazione sul broker, dato che tutti i messaggi devono passare da lui, c'è un forte disaccoppiamento tra gli agenti che si trovano agli estremi della comunicazione:

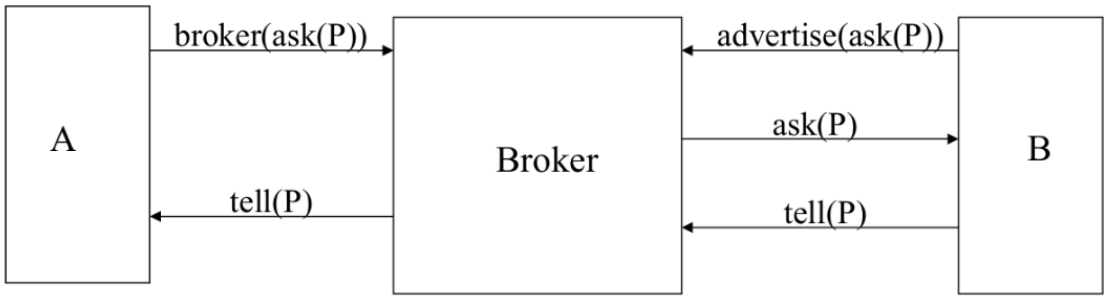



Figure 3.2: Comportamento broker in KQML

Inizialmente questo linguaggio non disponeva di semantica, dava solo un contenitore in cui mettere il messaggio senza dare significato nè al contenuto nè alle performative. Labrou e Finin introducono una semantica per KQML in termini di:

- **precondizioni**: dato un mittente $A$ e un destinatario $B$ descrivono lo stato necessario per $A$ al fine di inviare la performativa, e le precondizioni per $B$ per accettarla ed elaborarla con successo;

- **postcondizioni**: descrivono lo stato di $A$ dopo aver espresso la performativa e lo stato di $B$ dopo averla ricevuto e compresa;
- **condizioni di completamento**: definite per una performativa descrivono lo stato finale dopo, per esempio, che una conversazione sia stata completata.

  Precondizioni, postcondizioni e condizioni di completamento descrivono lo stato degli agenti in un linguaggio basato su stati mentali espressi mediante attitudini (*beliefs, knowledge, desire e intention*) e descrizioni di azioni (*per inviare e elaborare un messaggio*). Non è fornito un modello semantico per gli stati mentali espressi mediante attitudini.

### FIPA ACL

La *Foundation for Intelligent Physical Agents* (FIPA) è un consorzio tra università, centri di ricerca ed aziende private soprattutto nell'ambito delle telecomunicazioni. FIPA è stata fondata nel 1996 con il fine di produrre degli standard per il software per agenti interagenti ed eterogenei e sistemi basati su agenti, come ad esempio Jade 4. A tal fine propongono FIPA ACL con l'intento di superare i limiti di KQML iniziando col definire 22 atti comunicativi standardizzati. FIPA ACL e KQML sono molto simili dal punto di vista sintattico, ciò che gli differenzia è principalmente la semantica, seppur sempre basata sugli stati mentali.

**Esempio di messaggio FIPA ACL**

```
(inform
    :sender agent1
    :receiver agent2
    :language Prolog
    :content ‘‘weather(today, raining)’’
)
```

Figure 3.3: Esempio messaggio FIPA ACL

Per introdurre la semantica viene proposto il **Semantic Language** (SL) che è un linguaggio formale basato su una logica multimodale quantificata con operatori modali:

- $B_i\ \phi$: l'agente $i$ crede $\phi$;
- $U_i\ \phi$: l'agente $i$ è incerto su $\phi$, ma pensa che $\phi$ sia più probabile di $\neg\phi$;
- $C_i\ \phi$: l'agente $i$ desidera che valga correttamente $\phi$.

Vengono introdotti anche degli operatori per ragionare sulle azioni:

- **Feasible**$(a, \phi)$: l'azione $a$ può occorrere e in tal caso $\phi$ sarà vera;
- **Done**$(a, \phi)$: l'azione $a$ è appena avvenuta e $\phi$ sarà quindi vera;

- **Agent**$(i, a)$: l'agente $i$ è l'unico in grado di svolgere l'azione $a$.

Grazie agli operatori modali di base e al ragionamento sulle azioni si può definire il concetto di **goal persistente** $PG_i\ \phi$ o l'intenzione $I_i\ \phi$ che è un goal persistente che impone all'agente di agire. La semantica degli atti comunicativi è specificata come un insieme di formule SL che descrivono le:

- **precondizioni di fattibilità** (*feasibility preconditions*): condizioni che devono valere per il mittente per eseguire in modo appropriato gli atti comunicativi. Questo a differenza di KQML in cui le precondizioni dovevano valere per entrambe le parti coinvolte nella comunicazione;
- **effetti razionali** (*rational effects*): gli effetti che un agente può aspettarsi che occorrano come risultato dell'esecuzione di una azione, il motivo del perchè viene scelta un'azione per l'esecuzione. All'agente destinatario non è richiesto garantire che occorra l'effetto atteso.

Un aspetto rilevante della semantica di FIPA ACL è la modularità in quanto ogni performativa viene generalmente definita a partire da due di base: la **INFORM** e la **REQUEST**. Sulla base delle performative FIPA introduce il **test di conformità** che garantisce le precondizioni da parte del mittente, senza però alcuna necessità di garantire gli effetti sul destinatario, composto da:

- **Syntactic conformance testing**: l'obiettivo è determinare se un agente rispetta la sintassi del linguaggio ogni volta che comunica;
- **Semantic conformance testing**: l'obiettivo è determinare se un agente rispetta la semantica del linguaggio ogni volta che comunica. Aspetto complesso da verificare soprattutto se non abbiamo agente BDI, ma un programma convenzionale. Wooldridge propone di mappare concetti per la verifica di un programma classico (*invarianti, ecc...*) all'interno di concetti BDI basati sugli stati mentali.

Per garantire la conformità generalmente si procede con il definire un software con il quale l'agente dovrà interagire, se ci riesce sarà conforme. Questo permette di controllare perfettamente l'aspetto sintattico, ma non semantico.

**Protocolli di interazione** Gli agenti non possono prendere parte a un dialogo semplicemente scambiandosi dei messaggi ACL. L'analisi di numerose conversazioni umane mostra che sono presenti degli schemi (pattern) ricorrenti di conversazioni. L'ambizione di FIPA era quella di introdurre dei protocolli di interazione come mattoni base per costruire i dialoghi. In generale, i protocolli specificano il contenuto della comunicazione, non solo la forma. I protocolli sono solitamente modellati come macchine a stati finiti in cui se dobbiamo rappresentare l'interazione tra più agenti, gli archi vengono etichettati con il nome dell'agente che deve compiere la tale azione (*esempio A: Request*):

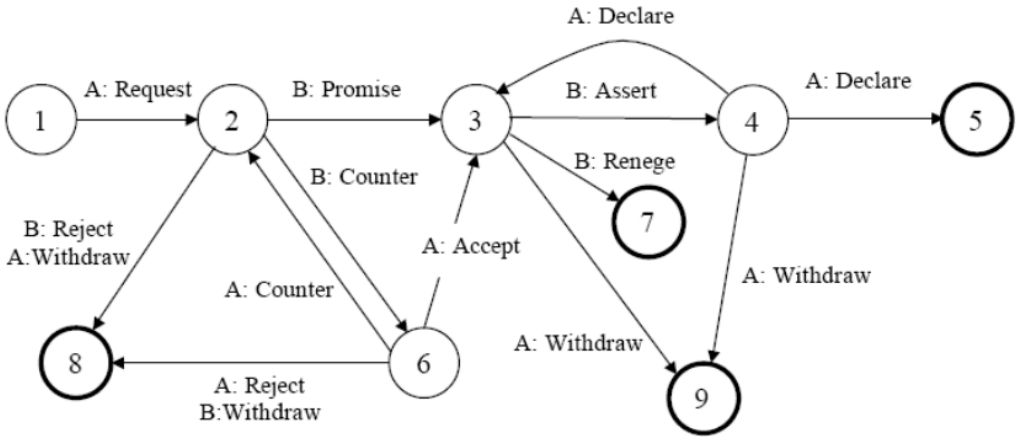



Figure 3.4: Esempio di Protocollo di Interazione Sociale

Sono stati proposti numerosi altri formalismi, come:

- **Reti di Petri**;
- **Definite Clause Grammars**: estensioni delle grammatiche context free dove i non terminali possono essere composti da termini e il corpo di una regola può contenere procedure;
- **AUML**: le specifiche di FIPA definiscono i protocolli attraverso AUML, una estensione di UML per gli agenti.

### 3.2.2 Conclusioni sugli ACL visti finora

Questi ACL che offrono una metafora molto vicina al linguaggio umano, falliscono spesso nell'utilizzo pratico, poichè difficili da verificare. La semantica basata su stati mentali ha infatti come aspetto negativo il fatto che è impossibile fidarsi completamente di altri agenti o fare forti assunzioni a proposito dello stato interno degli agenti e sul loro modo di ragionare soprattutto in contesti eterogenei e competitivi.

### 3.2.3 ACL su Semantica Sociale

L'approccio sociale, invece, considera le conseguenze sociali di eseguire un atto comunicativo, riconoscendo che la comunicazione è **inerentemente pubblica** e che quindi dipende dal contesto sociale degli agenti. L'idea è che la comunicazione definisca degli **impegni/contratti** tra gli agenti che devono essere rispettati. Il contratto crea delle aspettative verso un certo evento la cui realizzazione può essere verificata. Gli effetti sociali quindi, a differenza degli stati mentali in cui non è possibile fare introspezione sugli agenti, sono facilmente verificabili poichè direttamente osservabili.
Singh definisce quali sono le caratteristiche che un buon ACL dovrebbe avere:

- **Formalità**: chiarezza della specifica per guidare al meglio colui che realizza un sistema;

- **Dichiaratività**: descrivere *cosa* (what) piuttosto che *come* (how);
- **Verificabilità**: determinare se un agente sta agendo in accordo con una data semantica;
- **Significatività**: trattare la comunicazione secondo il suo significato e non come arbitrari token che devono essere ordinati in una qualche maniera.

Un protocollo è rispettato se e solo se l'interazione specificata/desiderata occorre durante l'esecuzione. Per la sua verifica è fondamentale avere una semantica basata su fatti "*osservabili*" e che sia indipendente dagli stati mentali dei partecipanti. E' inoltre importante saper garantire la verificabilità sia da un partecipante sia da una terza parte che deve monitorare l'interazione. L'approccio sociale è basato sui **practical commitments** (impegni) tra agenti: un agente (il **debtor**) si impegna verso un altro agente (il **creditor**) a rendere vero un qualche fatto o ad eseguire una qualche azione. Secondo questo approccio i vari atti illocutori possono essere visti in termini di commitment sociali che i partecipanti creano. Il creare i commitments è il punto di base per la costruzione di un'interazione: è proprio perchè esiste tale impegno che gli agenti vorranno eseguire certe azioni. Si possono definire anche i **metacommitments** che riguardano *l'impegno ad impegnarsi* per catturare un'ampia varietà di relazione legali e sociali in un unico framework. Un commitment può essere espresso come una 4-upla *C( debitor, creditor, antecedent, consequence)* con significato: il *debitore* è socialmente legato al *creditore* a rendere vera la condizione *conseguente* se la condizione *antecedente* è vera. I commitments sono manipolabili dagli agenti facendoli assumere diversi stati all'interno del loro ciclo di vita:

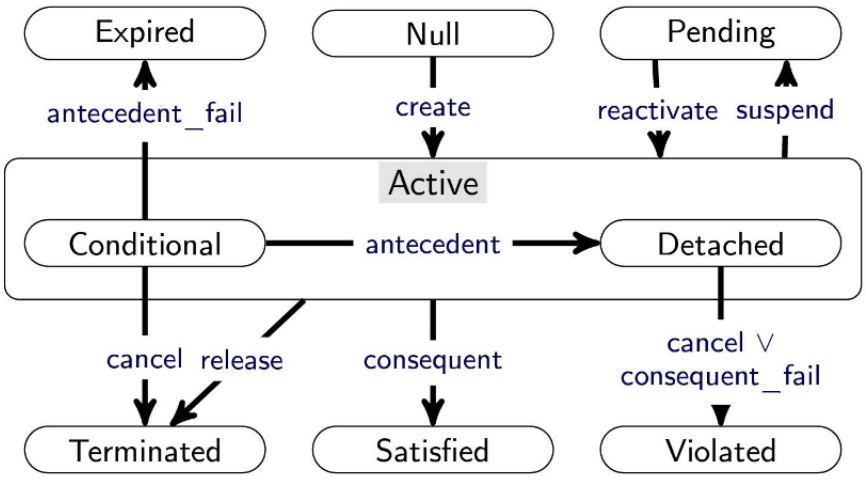



Figure 3.5: Ciclo di vita con commitments

### Protocolli a Commitment

Yolum e Singh introducono i protocolli basati su commitment nel 2001 come insiemi di azioni il cui significato espresso in termini di effetti sullo stato sociale è condiviso da tutti gli

agenti interagenti. In particolare gli effetti sono espressi in termini di operazioni sui commitments: *create, delete, release, delegate, assign, discharge*. I protocolli a commitments vengono introdotti per dare massima libertà d'esecuzione agli agenti per le loro azioni che devono essere guidate solamente dai commitments. Gli agenti giocano ruoli diversi nella società e i ruoli definiscono l'associazione con gli impegni sociali e le obbligazioni verso gli altri ruoli.

> Il solo vincolo che un protocollo a commitment deve soddisfare perché una interazione sia di successo è che tutti i commitments siano discharged.

Questi protocolli sono molto più flessibili dei protocolli d'interazione basati su automi, perchè mi permettono di cogliere molte varianti senza darne una definizione esplicita. La libertà d'azione agli agenti è massima poichè non importa il come (il percorso di azioni), ma solamente il fatto che alla fine dell'interazione non ci siano commitments pendenti. Implicitamente le azioni degli agenti fanno progredire il commitments nel suo diagramma degli stati. L'esecuzione corretta di una sequenza di azioni che portano ad avere nessun impegno pendente viene detta **run del protocollo**. Il dialogo tra 2 agenti avviene tramite l'intreccio dei commitments: un agente mette a disposizone delle azioni (conseguenza del commitment) a patto che ci siano delle precondizioni desiderate. Ogni agente non è in grado da solo di soddisfare i propri obiettivi e allora esegue delle azioni per gli altri al fine di ottenere dagli altri le azioni desiderate. Le precondizioni di un impegno se diventano vere lo trasformano in un **obbligo**. In questo modo risulta molto semplice la **verifica** della correttezza di un'interazione andando a vedere se essa sia o meno una run del protocollo. Poiché i requisiti di un protocollo vengono espressi solamente attraverso commitments, gli agenti possono essere conformi sulla base della loro comunicazione (non è necessaria conoscere la loro implementazione).

#### Metodi di Coordinazione

Esistono metodi per coordinare attività diversi dai protocolli come:

- **metodi a blackboard**: si usa una lavagna come se fosse una memoria condivisa in cui si possono fare operazioni di input/output con attesa. Il paradigma prevede la presenza di $n$ gessetti che devono essere posseduti per scrivere sulla lavagna. Calato nella programmazione ad agenti la blackboard altro non è che l'ambiente;
- **metodi a norme**: le *leggi sociali* sono uno dei meccanismi più diffusi per coordinare le attività. Una norma esprime uno schema di comportamento atteso che può essere o meno forzato.

# Chapter 4

# Jade

## 4.1 Storia

La piattaforma Jade è stata sviluppata da Telecom all'interno di FIPA, la cui mission era: *"promozione di tecnologie e specifiche di interoperabilità che facilitino l'interazione end-to-end di sistemi di agenti intelligenti nei moderni contesti commerciali ed industriali"*. Al fine di portare a termine tale obiettivo FIPA si occupava di aspetti disparati:

- Gestione del ciclo di vita degli agenti;
- Trasporto di messaggi;
- Struttura del messaggio;
- Protocolli di interazione tra agenti;
- Ontologie;
- Sicurezza.

## 4.2 Piattaforma ad agenti di FIPA

Ad alto livello possiamo dire che l'interesse di FIPA era rendere la comunicazione tra agenti il più semplice e trasparente possibile. L'agente di per sè, come progettazione, era fuori dagli interessi di FIPA che li considerava come: *autonomi, reattivi, proattivi, goal-driven, sociali, adattivi e cognitivi*.

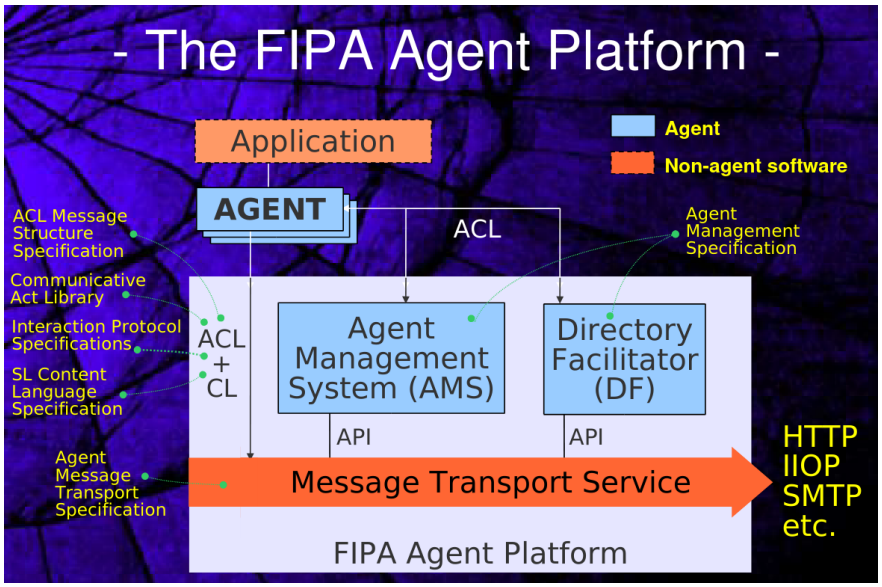



Figure 4.1: FIPA Agent Platform

La gestione degli agenti in Jade segue il seguente schema:

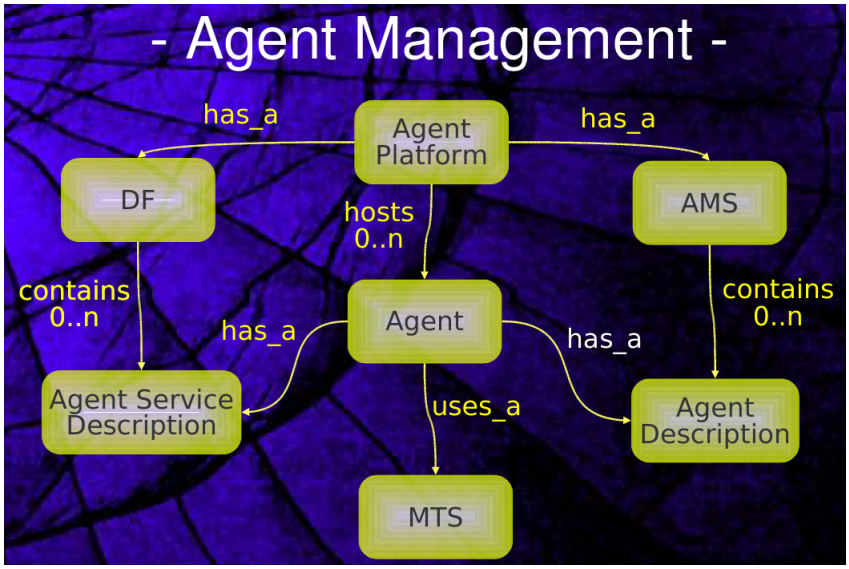



Figure 4.2: Agent Management

Una piattaforma ad agenti ha le seguenti componenti fondamentali: gli agenti, il **Directory Facilitator**, il **Message Transport Service** e l'**Agent Management System**. L'AMS e il DF sono degli agenti che si occupano rispettivamente della gestione degli agenti e del servizio di pagine gialle per pubblicizzare i compiti che mettono a disposizione gli agenti e, in quanto tali, è possibile interagire con loro attraverso ACL. Le operazioni che AMS e DF devono garantire sono: registrare, deregistrare, modificare e cercare le descrizioni delle informazioni che memorizzano. Nello specifico DF contiene le seguenti informazioni riguardo ad un agente: Name, Location, Services, Protocols, Ontologies, Lease-time e Scope; mentre l'AMS solamente: Name, Owner e State. L'MTS va invece a specifare la struttura e il tipo

di messaggi ACL che la piattaforma Jade gestisce verso agenti risiedenti sulla stessa o su diverse piattaforme.

## 4.3 Jade

Jade è una piattaforma per agenti che implementa i servizi e le infrastrutture base di un'applicazione multi-agente distribuita. A livello pratico Jade è una libreria Java che ha l'intento di nascondere FIPA e le specifiche dei vari standard usati al programmatore. L'architettura che propone Jade è la seguente:

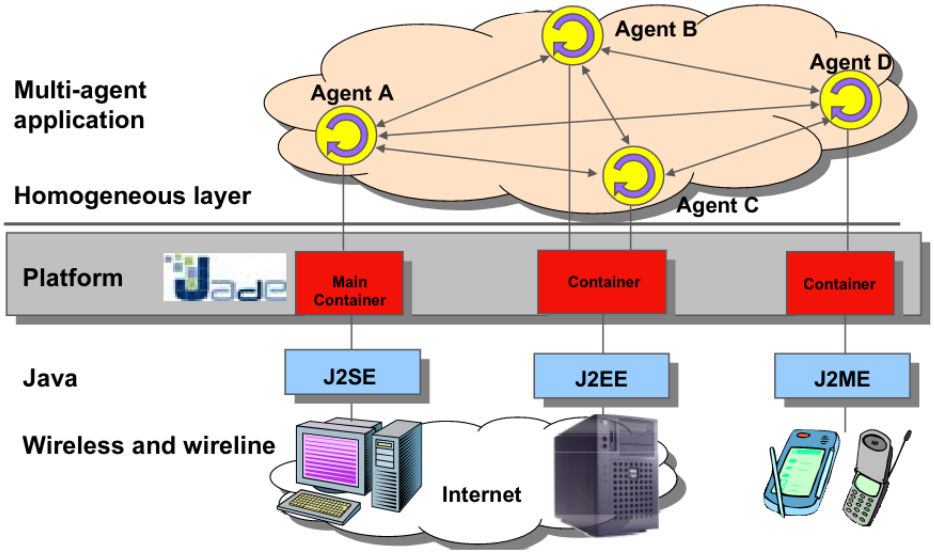



Figure 4.3: Architettura Jade

- una **applicazione multi-agente** basata su Jade che comprende una collezione di componenti attivi chiamati **agenti**;
- ogni agente ha un nome univoco ed è un peer che può comunicare in entrambe le direzioni con tutti gli altri agenti. Ogni agente risiederà su un certo **container** che devono comunicare tra di loro in maniera trasparente e che possono migrare facilmente con la piattaforma;
- un **container** che svolge il ruolo di main dove l'AMS e il DF vivono.

La GUI di Jade permette la gestione, il controllo, il monitoraggio e il debug di una piattaforma multi-agente attraverso:

- **Remote Management Agent** che introduce le seguenti funzionalità:
  - monitora e controlla la piattaforma e tutti i suoi containers remoti;
  - gestione remota del ciclo di vita di tutti gli agenti;
  - composizione ed invio di messaggi personalizzati agli agenti;

  - avvio di altri tool grafici che riguardano gli agenti;
  - monitora, in sola lettura, altre piattaforme compliant con FIPA;

- **Dummy Agent** che permette di:

  - comporre ed inviare un messaggio personalizzato;
  - caricare e/o salvare la coda dei messaggi da e verso un file;

- **Sniffer Agent** che puo:

  - mostrare il flow di interazione tra agenti selezionati;
  - mostrare il contenuto di ogni messaggio che è stato scambiato;
  - salvare e/o caricare il flow di interazione da un file già esistente;

- **Introspector Agent** le cui funzionalità sono:

  - monitorare lo stato interno degli agenti e quindi:
    * messaggi ricevuti/inviati e pendenti;
    * stato dell'agente;
    * behaviour e sub-behaviour schedulati;
  - esecuzione del debugging passo per passo, a break points e lentamente;

- **Log Manager Agent**: è la GUI che permette di modificare a run-time il logging della piattaforma aggiungendo nuovi handlers di logging, modificare il livello del logging e altro.

### 4.3.1 Classi e Metodi utili

#### Classe Agent

Per la generazione di un nostro Agente personalizzato dobbiamo estendere la classe primitiva `Agent` e sovrascrivere il suo metodo `setup()` dal quale l'esecuzione parte sempre. Ogni agente viene identificato da un **AgentID** univoco che lo contraddistingue con forma:

$\langle localname \rangle$ @ $\langle platform\text{-}name \rangle$ esempio: $\langle peter \rangle @ \langle 10.0.11.74 : 1099/JADE \rangle$

Tra i metodi propri della classe `Agent` e che vengono quindi ereditati troviamo:

- `doWait(N)`: mette l'agente in attesa di $N$ secondi, lo blocca temporaneamente;
- `doDelete()`: programma la cancellazione dell'agente che avverrà dopo la sua `setup()`. Nello specifico setta il flag dell'eliminazione a *True* che verrà eseguita dal gestore degli agenti;

- `takeDown()`: è il metodo finale, duale del `setup()`, che viene invocato sempre in uscita quando l'agente viene rimosso;
- `addBehaviour()`: aggiunge un task/comportamento all'agente che da lì in poi sarà schedulato;
- `removeBehaviour()`: toglie il comportamento dal pool dei comportamenti attivi. Non esegue la `onEnd()`.

Ogni agente è definito su un solo thread per facilitare il suo eventuale spostamento tra diversi containers. Le operazioni sopra definite lavorano quindi sul singolo thread dell'agente. Il fatto che ogni agente sia su un solo thread non è limitativo poichè ovviamente le sue azioni sono collaborative tra di loro, non in competizione. Sarà quindi il singolo agente che decide l'ordine delle azioni che vuole compiere. Agenti diversi, in competizione tra loro, saranno su threads distinti.
L'esecuzione di un agente rispetta il seguente modello:

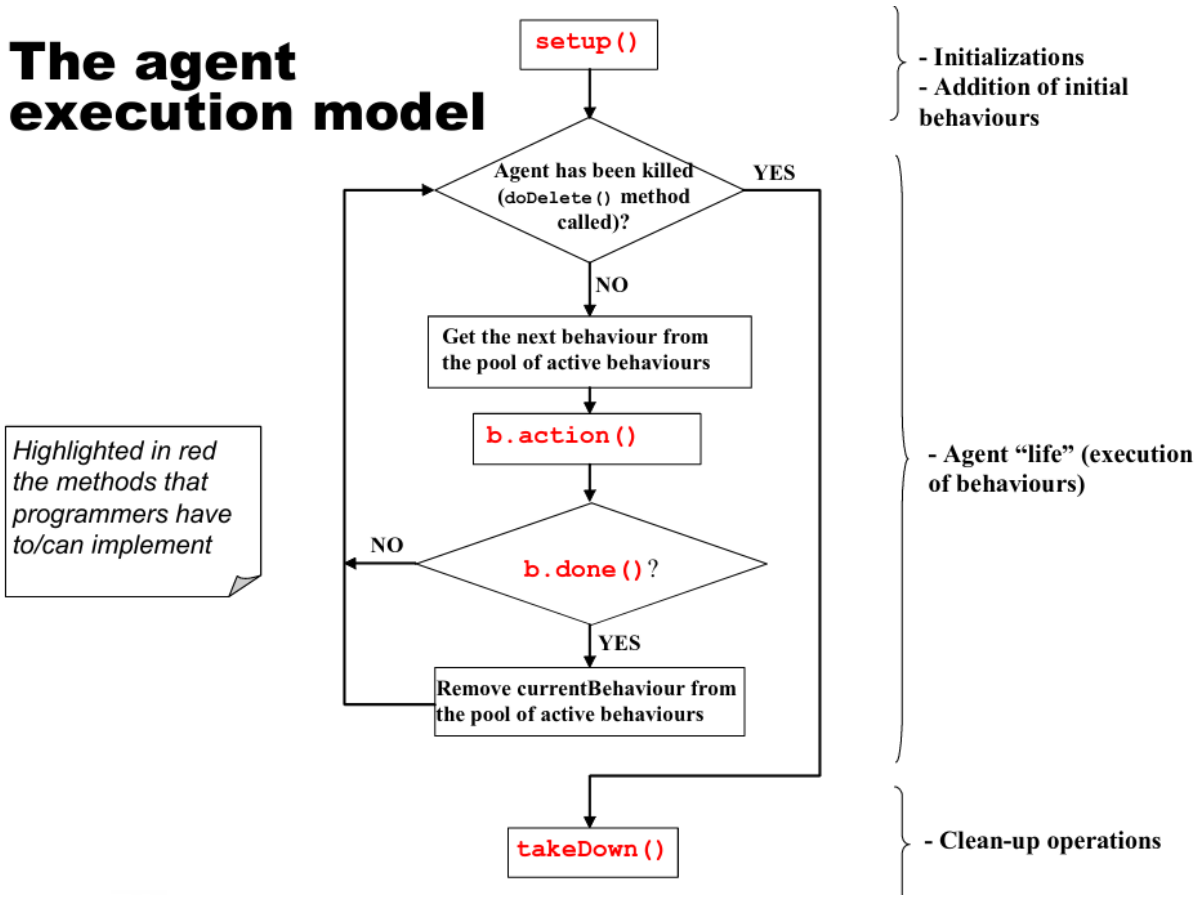



Figure 4.4: Agent Execution Model

**Classe Behaviour** Per definire un task per l'agente dobbiamo estendere la classe `Behaviour` andando a ridefinire i metodi:

- `action()`: definisce il comportamento dell'agente;
- `done()`: metodo booleano che controlla se l'obiettivo dell'agente è stato raggiunto;
- `block()`: non blocca l'agente, semplicemente toglie il Behaviour dalla lista dei comportamenti da eseguire e lo mette in una lista d'attesa, riprendendolo non appena la condizione specificata si soddisfa. Non è come una `wait()` in cui il codice ricomincia da dove si era interrotto, bensì viene rieseguito l'intera `action()` quando il Behaviour ritorna in gioco. E' praticamente solamente un flag che avvisa lo scheduler di non considerare tale comportamento per il momento. La action corrente viene comunque eseguita tutta nonostante avvenga una `block()`;
- `onStart()`: metodo che viene eseguito solo una volta prima della prima esecuzione di action;
- `onEnd()`: metodo che viene eseguito solo una voilta alla fine dell'ultima esecuzione di action, quando `done()` restituisce True.

Idealmente `action()` descrive un passo d'esecuzione che deve essere ripetuto $N$ volte fino a quando lo stabilisce `done()`. Per un agente posso definire tutti i Behaviours che voglio che saranno portati avanti in parallelo facendo un passo per volta per ogni task. Dato che ogni agente ha un solo thread la parallelizzazione tra i tasks è simulata dallo scheduler. Un Behaviour può essere creato dinamicamente all'interno di una `action()` o all'interno del metodo principale `setup()`. Nella libreria troviamo dei Behaviours predefiniti a livello di `done()` e `action()` per implementare delle azioni classiche che potrebbero tornare utili. Lo scheduler dei Behaviours **non** è **preemptive** e, in quanto **cooperativo**, il metodo `action()` viene sempre eseguito completamente.

**Comunication Model** La comunicazione tra agenti viene realizzata in accordo con gli standards FIPA. Per realizzare la comunicazione tra agenti dobbiamo usare i seguenti metodi della classe `Agent`:

- `send()`: metodo *void* che ha bisogno di un paramentro della classe *ACLMessage*;
- `receive()`: non ha bisogno di parametri e restituisce un oggetto di tipo *ACLMessage*.

Inviare un messaggio comporta inserirlo all'interno della casella (coda) dei messaggi ricevuti di un altro agente. Quest'ultimo li leggerà in ordine di ricezione, ma deciderà lui quando e se leggerli. Il sistema di passaggio dei messaggi è **asincrono** ciò significa che se un agente fa `receive()` ed ha la casella vuota non aspetta di ricevere un messaggio, ma ottiene direttamente *null*. Bisogna far attenzione al fatto che i messaggi sono unici per ogni agente quindi se un suo Behaviour legge un messaggio dalla casella questo viene eliminato dalla coda dell'agente e non sarà più possibile ottenerlo. Gli agenti per conoscere gli indirizzi degli agenti (AgentID) a cui inviare messaggi possono usare il servizio delle **pagine gialle**. Ogni agente andrà quindi a registrarsi presso tale servizio indicando il compito che svolge e il proprio indirizzo. Generalmente questo serivizio viene implementato da un agente chiamato *DF Agent* sempre attivo che ascolta e registra gli altri agenti e verso cui bisogna comunicare per ricevere gli indirizzi.

# Chapter 5

# AgentSpeak(L)

## 5.1 Introduzione

AgentSpeak(L) è un linguaggio di programmazione *dichiarativo* per agenti BDI introdotto da Rao nel 1996 che si propone di colmare il gap tra specifica teorica ed implementazione di un agente BDI. Gli agenti BDI venivano da sempre trattati da due punti di vista:

1. **specifica teorica**: logiche modali con operatori modali per modellare le attitudini;
2. **implementazione**: strutture dati per rappresentare gli operatori modali.

Il gap tra teoria e pratica era eccessivo, la causa principale era la complessità del theorem proving e del model checking delle logiche usate per formalizzare gli agenti BDI. Con AgentSpeak(L) Rao introduce una formalizzazione alternativa degli agenti BDI, con una nuova architettura e un nuovo linguaggio di programmazione per agenti. AgentSpeak(L) può essere visto come estensione della programmazione logica per supportare l'architettura BDI. Gli agenti BDI sono visti come:

- sistemi collocati in un ambiente dinamico, che muta nel tempo;
- sistemi in grado di percepire informazioni provenienti dall'ambiente;
- sistemi in grado di compiere delle azioni ed apportare modifiche all'ambiente sulla base delle proprie attitudini mentali che sono:
  - **Beliefs**: catturano la componente di *informazione*;
  - **Desires**: catturano la componente *motivazionale*;
  - **Intentions**: catturano la componente *decisionale*.

Sono state proposte molte implementazioni degli agenti BDI, impiegate con successo in molti domini applicativi considerati critici. Le implementazioni sono caratterizzate da assunzioni che semplificano le definizioni di *belief*, *desire* ed *intention* in modo da modellare tali attitudini con maggiore facilità e migliorare l'efficienza computazionale. Le basi teoriche che supportano tali sistemi sono deboli. AgentSpeak(L) è un buon compromesso tra le due tendenze: quella che rimane troppo fedele alla logica garantendo solide basi, ma alta complessità e quelli che si discostano molto dalla logica per ottenere efficienza con perdita eccessiva di formalità.

## 5.2 AgentSpeak(L)

Si parte da un sistema BDI implementato e si formalizza la sua **semantica operazionale**. AgentSpeak(L) può essere visto come una versione semplificata e testuale di **PRS** (Linguaggi e Architetture per Agenti BDI 13) o della sua evoluzione **dMARS**. Nello specifico AgentSpeak(L) è un linguaggio di programmazione basato su un linguaggio semplificato del prim'ordine, con eventi e azioni usato per dettare il comportamento di un agente. I classici beliefs, desires ed intentions non sono rappresentati esplicitamente con formule modali, bensì il progettista attribuisce tali nozioni all'agente usando il linguaggio AgentSpeak(L):

- **Current belief state**: modello che l'agente ha di sé stesso, dell'ambiente e degli altri agenti;
- **Desires**: stati che l'agente vuole raggiungere, nello specifico AgentSpeak(L) fa riferimento a *goals* che altro non sono che *desires* scelti;
- **Intentions**: adozione di programmi per soddisfare certi stimoli.

#### Sintassi

Oltre ai classici connettivi della logica, si hanno variabili, costanti, simboli di funzione, predicati, quantificatori, ... e tutti gli altri costrutti della logica del prim'ordine. La novità consiste nella possibilità di definire degli **achievement goal** che si vogliono raggiungere mettendo il ! prima di un predicato e dei **test goal** che sono invece preceduti dal ?.

**Belief** Un **belief atomico** ha la seguente formulazione: $b(t_1,\ t_2,\ ...,\ t_n)$ dove $b$ è un simbolo predicativo e $t_1,\ t_2,\ ...,\ t_n$ sono termini. Può essere scritto in maniera compressa come $b(\mathbf{t})$. Un belief atomico e la sua negazione vengono chiamati **belief literal**. I belief atomici possono essere composti con simboli logici (es: $b(\mathbf{t}_1)\ \ \&\ \ c(\mathbf{t}_2)$) per formare beliefs complessi o semplicemente **beliefs**. Quando un belief è ground, ossia senza variabili libere viene detto **base belief**.

**Goal** È uno stato del sistema che l'agente vuole raggiungere. Ci sono due tipi diversi di goal definibili:

- **Achievement Goal**: $!g(\mathbf{t})$ l'agente vuole raggiungere uno stato in cui il belief $g(\mathbf{t})$ è soddisfatto;
- **Test Goal**: $?g(\mathbf{t})$ l'agente vuole verificare se il belief $g(\mathbf{t})$ sia o meno vero.

Quando un agente acquisisce un nuovo goal oppure nota una modifica nell'ambiente, esso può far scattare aggiunte o cancellazioni di goals e/o beliefs. Questi eventi vengono detti **triggering events**:

- aggiunta di belief o goal (achievement e test), indicati dal simbolo $+$;
- eliminazione di belief o goal (achievement e test), indicati dal simbolo $-$.

**Azione** Lo scopo di un agente è osservare l'ambiente e, sulla base di tali osservazioni e dei propri goals, eseguire determinate azioni. Le azioni compiute dall'agente possono modificare lo stato dell'ambiente. Sia $a$ un simbolo di azione e $\mathbf{t}$ una sequenza di termini, allora $a(\mathbf{t})$ rappresenta un'azione.

**Piani** I piani che possiamo definire in AgentSpeak(L) hanno le seguenti caratteristiche:

- sono **context sensitive** ossia prevedono la definizione di una condizione necessaria per poterli applicare e permettono di raggiungere e/o aggiungere altri sottogoals;
- possono essere invocati dagli utenti;
- consentono una decomposizione gerarchica dei goals;
- sono sintatticamente molto simili alle clausole della programmazione logica.

Un piano specifica il modo in cui un agente potrebbe raggiungere un determinato obiettivo:

$$\begin{array}{ll} \langle piano \rangle & \triangleq \langle head \rangle \leftarrow \langle body \rangle \\ \langle head \rangle & \triangleq \langle triggering\ \ event \rangle : \langle context \rangle \end{array}$$

Le cui parti hanno il seguente significato (se ne manca una viene impostata a *true*):

- **triggering event**: chiamata di procedura/l'evento che spiega perchè il piano è stato attivato, ossia l'aggiunta/rimozione di un belief o di un goal che tale piano si propone di gestire;
- **context**: le precondizioni, l'insieme di beliefs che dovrebbero essere soddisfatti nel set delle credenze dell'agente quando il piano è attivato;
- **body**: è una sequenza di **simple plan expressions**:
  - **goals** che l'agente vuole raggiungere (achievement goals) o testare (test goals);
  - **azioni** che l'agente deve eseguire

Instanziato un piano risulta quindi:

$$e : b_1 \ \& \ b_2 \ \& \ \cdots \ \& \ b_n \leftarrow h_1, \ h_2, \ ..., \ h_m.$$

**Semantica Operazionale**

A run-time un agente può essere visto come costituito da:

- Un insieme di **beliefs** $B$;
- Un insieme di **piani** $P$;

- Un insieme di **intentions** $I$ e ognuna è uno stack di piani parzialmente istanziati: ossia dove non necessariamente tutte le variabili devono essere istanziate. Un'intention è denotata con uno stack nel modo seguente: $[\ p_1 \mid p_2 \mid ... \mid p_z\ ]$. Viene chiamata **true intention**: $T \equiv [+!true : true \leftarrow true]$;
- Un insieme di **eventi** $E$ in cui ogni evento è una coppia $\langle e, i \rangle$ con $e$ evento ed $i$ intenzione. Un evento può essere esterno (modifica dell'ambiente) o interno (modifica dello stato dell'agente). Se l'evento è esterno allora $i = T$, altrimenti se è interno avrà l'intenzione esplicitata;
- Un insieme di **azioni** $A$ che l'agente può compiere per modificare il mondo;
- Un insieme di **funzioni di selezione**: $S_e,\ S_o,\ S_I$.

Il progettista specifica un agente scrivendo un insieme di *base beliefs* (o fatti dell'agente) e un insieme di piani (o regole del paradigma logico). L'agente *parte* poichè guidato da goal iniziali.

**Reasoning Cycle** Il comportamento di un agente può essere spiegato con il seguente ciclo:

1. Viene generato un *triggering event* causato ad esempio dall'aver notato una modifica nell'ambiente circostante o da un utente esterno che chiede all'agente di raggiungere un goal, ecc...;
2. Viene aggiunto l'evento con associato la causa/intenzione all'insieme $E$ (non c'è causa se è esterno);
3. Applicazione della **Belief Revision Function** che va a fare eventuali cambiamenti all'insieme $B$ dei beliefs a seguito dell'evento che è capitato;
4. $S_e$ seleziona un evento $E_0$ da $E$ per essere processato, rimuovendolo dall'insieme $E$;
5. Si osservano i piani nell'insieme $P$ che hanno la triggering event che può essere unificata (**unificatore rilevante**) con $E_0 \rightarrow$ **piani rilevanti**. L'unificazione si propaga al contesto e al body dei piani rilevanti (si parlerà sempre di *mgu: most general unifier*);
6. Presi i piani rilevanti una **correct answer substitution** va ad unificare il loro contesto con l'insieme dei base beliefs $B$. I piani le cui condizioni risultano essere conseguenze logiche del set $B$ vengono detti **piani applicabili** mediante un **unificatore applicabile** che è il risultato della composizione della correct answer substitution con l'unificatore rilevante;
7. $S_o$ va a scegliere uno tra i piani applicabili che viene detto $P_o$. Applicando l'unificatore applicabile al corpo di $P_o$ si ottiene l'**intended means** in risposta al *triggering event*;
8. $P_o$ unificato viene aggiunto alla lista $I$, in cui ogni intenzione è uno stack di piani parzialmente istanziati o *intention frames*. Se $P_o$ è causato da un evento esterno l'intended means è usato per creare una nuova intenzione il cui stack viene inserito in $I$. Se è invece causato da un evento interno viene inserito in cima all'intention frame esistente che ha "fatto scattare" l'evento interno;

9. $S_I$ sceglie la prossima intention di cui esegue un solo passo, che se è:
   (a) **achievement goal**: equivale a generare un evento interno per aggiungere il goal alla corrente intention (stack);
   (b) **test goal**: equivale a trovare una sostituzione per il goal che lo renda una conseguenza logica dei base beliefs; se viene trovata viene rimosso dal corpo dell'intention, altrimenti fallisce il piano;
   (c) **azione**: equivale ad aggiungerla al set di azioni $A$ e rimuoverla dal corpo dell'intention;
10. ritorna ad 1) fino a quando il set degli eventi $E$ è vuoto e non è possibile eseguire altre intentions.

Un agente può perseguire più di 1 goal contemporaneamente e quindi avere attivi più piani nell'insieme delle intenzioni, che possono essere portati avanti in parallelo. I comportamenti di uno stesso agente dovrebbero essere collaborativi e per questo non ci dovrebbero essere problemi. A fronte della doppia unificazione prima per avere piano rilevante e poi applicabile è comunque possibile che un piano abbia nel corso una variabile ancora libera e non istanziata.

# Chapter 6

# Jason

Jason è la prima significativa implementazione open source di AgentSpeak(L) 5, realizzata in Java da Rafael H. Bordini e Jomi F. Hubner con l' intento di semplificare la programmazione di agenti con un linguaggio ad oggetti. Jason di base utilizza SACI come infrastruttura per la comunicazione fra agenti, che rende possibile distribuire un sistema multi-agente sulla rete. E' però possibile usare FIPA ACL per la comunicazione, descritto in Comunicazione tra Agenti 3. Costruire un agente AgentSpeak(L) con Jason è molto semplice in quanto basta inserire il nome e il numero degli agenti in un file di configurazione `.mas2j` e poi creare un file `.asl` contenente i piani che descrivono il comportamento di ogni agente.

## 6.1 Differenza con AgentSpeak(L)

Jason, rispetto ad AgentSpeak(L), introduce importanti caratteristiche:

- **Gestione del fallimento dei piani**: sono previsti eventi per la gestione del fallimento dei piani che vengono generati se un'azione fallisce o non vi sono piani applicabili per un evento con aggiunta di un goal $+!g$. In tali situazioni viene generato un evento interno per $-!g$ associato alla stessa intention. Se il programmatore ha previsto un piano per $-!g$ e questo risulta applicabile, verrà eseguito (può essere visto come se fosse un'eccezione). Altrimenti, viene eliminata l'intera intention e segnalato un warning;
- **Supporto per lo sviluppo di ambienti da programmare in Java**;
- **Possibilità di eseguire un ambiente multi-agente utilizzando SACI**;
- **Possibilità di personalizzare in Java funzioni di selezione, di belief-revision, di azione e di comunicazione fra agenti**;
- **Libreria di "azioni interne" di base**: possono essere usate sia nel contesto che nel body di un piano e sono introdotte dal punto. Sono azioni interne, distinte dalle azioni che l'agente compie sull'ambiente mediante gli attuatori. Tra queste azioni interne c'è la `.send(···)` e per fare la risposta devo definire un piano che si azioni alla ricezione dell'evento e prenda l'informazione necessaria dalla `[source()]`. Vi è anche la `.print(···)`, la `.fail(···)` per far fallire un piano, la `.df_register(···)` per registrare l'agente al servizio di pagine gialle e molte altre;

- **Possibilità di aggiungere "azioni interne" definite in Java dall'utente**;
- **Aggiunta della negazione forte classica** dato che quella presente in AgentSpeak(L) è la negazione per fallimento. C'è il solito discorso del fatto che la negazione è in pratica non implementabile ed è solo zucchero sintattico per dire che per ogni predicato c'è anche il suo negato che afferma il non verificarsi del predicato proposizionalizzato su tutte le costanti non presenti nel predicato originario. A livello sintattico permette la definizione di liste Prolog like, stringhe, numeri, strutture, ecc... Esattamente come per Prolog permette di definire come credenze di base non solo predicati, ma anche regole;
- **Annotazioni**: c'è la possibilità di specificare **annotazioni** ai predicati atomici nella base belief, per esempio per conservare l'origine di tale informazione. L'annotazione è una lista di termini tra parentesi quadre associata ad un predicato, di dimensione pari ai suoi termini orginali: $p(t_1,\ t_2,\ ...,\ t_n)[a_1,\ a_2,\ ...,\ a_n]$. Vi sono due annotazioni particolarmente usate:
  - **[source(percept)]**: l'informazione è stata percepita dall'ambiente;
  - **[source(self)]**: l'informazione è stata aggiunta dall'agente stesso durante l'esecuzione di un piano.

  Tutti i predicati nella belied base hanno implicitamente un'annotazione speciale che riporta la sorgente dell'informazione. E' possibile "annotare" anche i piani, in questo caso si parla di **labels**. La label può essere un qualsiasi predicato (consigliata arietà zero), quindi anche un predicato con annotazione e può risultare utile per personalizzare la funzione di selezione di un piano applicabile.

#### Considerazioni per run-time

Ovviamente a tempo di esecuzione una variabile può unificare con un solo termine, quindi anche se ci sono più piani applicabili, grazie alla presenza di una variabile non ancora istanziata, alla fine solo uno verrà eseguito, quello che viene definito prima nel programma. Solo nella testa e nello specifico nel contesto di una regola posso fare backtracking, nel senso che se ci sono più piani applicabili ed uno fallisce posso eseguirne un altro. Questo non accade nel corpo di una regola se uso un test goal, perchè fare un test goal è un commit, un impegno che l'agente prende e può fallire. A livello pratico fare un test goal, anche se fallisce, è già come fare un passo, mentre se il piano fallisce e ne cerco un altro in base al contesto di passi non ne ho ancora fatti. I piani possono avere un'annotazione per il nome la cui sintassi è `@nome`. Nel momento in cui definisco un agente e lo eseguo avrà come eventi iniziali da gestire sia ovviamente i goals che i base beliefs che sono visti come eventi di credenze che possono far scattare dei piani se definiti.

## 6.2 Sistemi Multi-Agente

L'utente può definire un sistema multi-agente mediante:

- Un ambiente in cui gli agenti AgentSpeak(L) vengono collocati, programmato in Java;

- Un set di istanze di agenti AgentSpeak(L).

La configurazione dell'intero sistema multi-agente è data da un semplice file di testo con estensione `.mas2j`. Nel file vengono indicati il nome attribuito alla società di agenti, gli agenti AgentSpeak(L) che ne fanno parte e l'ambiente in cui si collocano tali agenti (cioè, la classe Java, eventualmente ridefinita dall'utente, per programmare l'ambiente).

## 6.3 JaCaMo

**Ja**(son)**Ca**(rtago)**Mo**(ise) è l'unione di Jason con:

- **Cartago**: piattaforma nata con l'obiettivo di realizzare ambienti per agenti mediante librerie in Java. Questo perchè nei classici linguaggi non c'era la possibilità di definire ambienti in modo che fossero oggetti osservabili e anche in grado di eseguire azioni nel momento in cui un agente avesse operato su di esso. L'idea alla base è vedere l'ambiente come strumento fondamentale per la comunicazione tra agenti come l'aria lo è per le persone umane. Deriva dall'idea di coordinazione proposta dai sistemi a *blackboard*, vista come memoria condivisa tra processi. Non c'è un approccio giusto o sbagliato, dipende dal livello di astrazione a cui ci poniamo;
- **Moise**: propone di definire la coordinazione tra agenti tramite un'organizzazione che ha come obiettivo quello di mantere attiva e viva la comunicazione, stabilendo norme fondamentali per la coordinazione. L'organizzazione non è un agente, ma un ambiente con particolari regole. Come esempio abbiamo l'università che è un ambiente normato che permette la coordinazione di migliaia di agenti tra studenti, professori, tecnici, ecc...

Le norme condizionano il modo in cui i processi operano sugli oggetti a livello di triangolo di Meyer. Le norme riguardano il "fare la cosa giusta"; i processi riguardano invece "fare ciò che ci porta al goal".

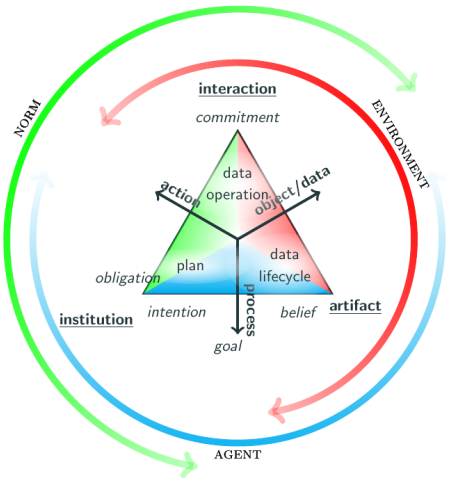



Figure 6.1: JaCaMo Meyer Triangle

Appoggiandosi alle norme gli agenti possono ragionare sulle conseguenze sociali delle loro azioni. Le norme possono essere sotanzialmente di due tipi diversi:

- **norme regolative**: specificano i modelli di comportamento accettabili (cioè azioni e interazioni) tramite obblighi, permessi, divieti;
- **norme costitutive**: definiscono le azioni istituzionali che hanno senso all'interno dell'istituzione di appartenenza. Specificano le condizioni perché le azioni possano essere utilizzate (sono applicabili).

JaCaMo è una piattaforma orientata alla programmazione multiagente che ambisce a programmare sistemi fornendo una perfetta integrazione di tre dimensioni:

- **Agenti**: grazie all'uso di Jason estesi in modo che possano recepire gli obblighi dettati dall' organizzazione. Gli agenti sono programmi che reagiscono diversamente verso gli obblighi che vengono emessi a seconda del *ruolo* che decidono di giocare nella società di agenti. Gli agenti osservano le proprietà dello stato organizzazionale che è un insieme dinamico di obblighi che cambia nel tempo;
- **Ambiente**: grazie all'integrazione con Cartago visto come insieme di spazi ognuno con artefatti che specificano delle operazioni e delle proprietà che sono osservabili;
- **Organizzazione**: per mezzo di Moise che specifica: norme, ruoli, goal, gruppi e schemi. Gli schemi sono delle decomposizioni funzionali di come raggiungere i goal organizzazionali, cioè quali sono i compiti assegnati ad ogni agente e quando devono essere compiuti. E' tutto fatto tramite norme: ci sono obblighi emessi a seconda dello stato dell'ambiente organizzazionale. Gli agenti osservano questi obblighi e aderiscono.

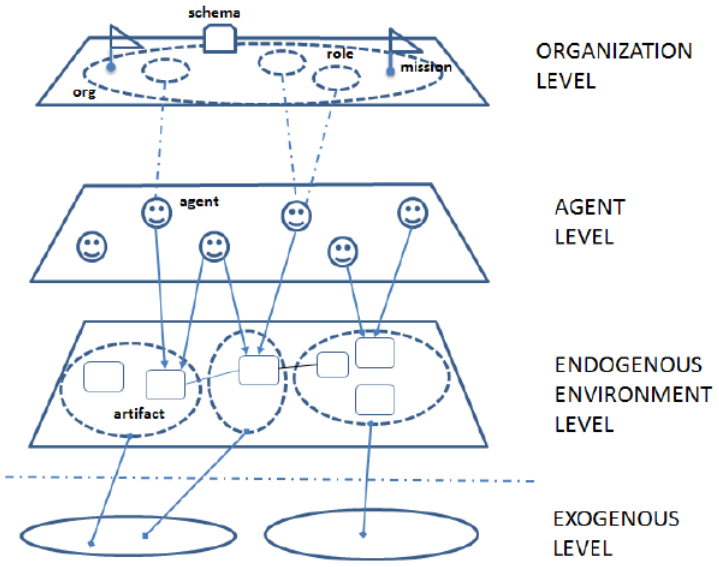



Figure 6.2: Livelli in JaCaMo

# Part II

# Alberto Martelli

# Chapter 7

# Logica Classica

### 7.0.1 Cos'è la logica?

La logica è stata sviluppata nel corso di molti secoli da matematici e filosofi per modellare il *ragionamento corretto* degli esseri umani e ha avuto un ruolo importante nello sviluppo dell'intelligenza artificiale per realizzare agenti e sistemi multiagente. La logica fornisce infatti strumenti per fare:

- **rappresentazione della conoscenza**: dato che è un linguaggio formale con una sintassi e semantica precisa permette una rappresentazione *dichiarativa* della conoscenza;
- **ragionamento automatico** mediante l'uso di regole di inferenza.

Queste proprietà della logica hanno diverse qualità come, ad esempio, quella di spiegare in modo preciso i passi di ragionamento di un agente.

### 7.0.2 Ruoli della logica per agenti

Date le proprietà citate risulta ovvio il fatto che la logica possa essere usata da un agente per rappresentare la conoscenza e ragionare. Dato che la conoscenza è espressa in un linguaggio formale, gli agenti possono usare metodi formali (inferenza) per derivare altra conoscenza. Meno ovvio risulta invece l'uso della logica per **specificare il comportamento** di un agente intelligente. In questo caso la logica può essere usata per **verificare** che l'agente si comporti come specificato, anche se l'agente non fa uso della logica nel suo funzionamento.

### 7.0.3 Logica Classica

Normalmente, quando si parla di logica in ambito IA si intende la cosiddetta logica classica, ossia la **logica proposizionale** o la **logica del prim'ordine**. Tuttavia, la necessità di modellare concetti diversi e le esigenze di efficienza hanno portato l'IA all'uso di logiche diverse da quelle classiche e anche alla definizione di nuove logiche come, ad esempio, le logiche non monotone. Nel seguito vedremo che nell'ambito degli agenti e sistemi multiagenti si preferisce utilizzare una logica non classica nota come logica modale. Come già affermato più volte gli agenti vengono spesso descritti come **sistemi intenzionali** attribuendo loro

stati mentali come credenze, desideri, ecc... In questo contesto la logica classica risulta inadeguata. Se volessimo ad esempio esprimere una credenza con la FOL avremmo:

$$\text{'John crede che Superman voli'} \quad \longrightarrow \quad Belief(John,\ vola(Superman))$$

Il predicato *Belief* risulterebbe problematico dal punto di vista:

- **sintattico**: in quanto non rispetta la forma standard $Predicate(Term_1, Term_2, ..., Term_n)$, ma propone un predicato all'interno di un altro predicato. Un termine infatti può essere solamente: una variabile, una costante o una funzione;
- **semantico**: gli operatori intenzionali come *Belief* sono **referentially opaque**: creano dei contesti chiusi in cui non è possible sostituire una formula con una equivalente come accade invece nella logica classica. Ossia se so che *Clark* e *Superman* sono la stessa persona non posso comunque derivare $Belief(John,\ vola(Clark))$ a meno che non si creda esplicitamente che *Superman* = *Clark*.

Vista l'inadeguatezza della logica classica per trattare sistemi intenzionali, sono state proposte tecniche alternative:

- l'approccio più diffuso è quello di usare la Logica Modale 8, che fornisce **operatori modali** come *Belief* che possono avere formule come argomenti;
- usare un **meta-linguaggio**, ossia un linguaggio del prim'ordine contenente termini che denotano formule. Questo approccio è poco diffuso e non verrà trattato.

### Logica classica Proposizionale

**Sintassi** Le **formule** sono costituite da **proposizioni atomiche** appartenenti ad un insieme $P$, e da **connettivi logici** secondo la seguente formulazione:

- $p \in P$ viene detta *formula atomica* poichè priva di connettivi;
- $\phi \ \vee \ \psi$ o $\neg\phi$ in cui $\phi$ e $\psi$ sono formule e '$\vee$', '$\neg$' connettivi logici.

Per semplicità introduciamo solamente i connettivi universali, in quanto da questi siamo in grado di ottenere qualsiasi altro connettivo più complesso. Ogni formula logica proposizione può infatti essere scritta in *CNF*.

**Semantica** La semantica definisce la verità delle formule rispetto ad ogni **modello**. Nella logica proposizionale, un modello assegna un valore di verità (*true* o *false*) ad ogni simbolo proposizionale. Se quindi $|P| = n$ allora ci sono $2^n$ possibili modelli. Per brevità un modello può essere rappresentato come un insieme $M \subseteq P$ che contiene tutte le proposizioni atomiche che sono vere nel modello. Quelle che non appartengono ad $M$ sono implicitamente false. Ottengo la verità delle formule complesse a partire dalle formule atomiche componendole attraverso l'uso di tabelle di verità. La semantica definisce quindi una **relazione di soddisfacibilità** $M \vDash \phi$ di una formula $\phi$ in un modello $M$:

- $M \vDash p \quad \iff \quad p \in M$;
- $M \vDash \phi \vee \psi \quad \iff \quad M \vDash \phi \vee M \vDash \psi$;
- $M \vDash \neg\phi \quad \iff \quad M \nvDash \phi$.

Una formula è **soddisfacibile** se e solo se c'è qualche modello che la soddisfa. Una formula è **valida** se e solo se è soddisfatta da ogni modello (**tautologia**). Una formula è una **contraddizione** se nessun modello la soddisfa. Data una base di conoscenza (insieme di formule) $KB$, una formula $\alpha$ segue logicamente da $KB$: $KB \vDash \alpha$ se e solo se, in ogni modello in cui $KB$ è vera, anche $\alpha$ lo è.

> **Teorema di deduzione**: date due formule $\alpha$ e $\beta$, $\alpha \vDash \beta$ se e solo se la formula $(\alpha \implies \beta)$ è valida.

Sono stati sviluppati numerosi **algoritmi di inferenza** per la logica proposizionale basati su schemi di assiomi e regole di inferenza come ad esempio il *Modus Ponens*:

$$\frac{\alpha \implies \beta \qquad \alpha}{\beta}$$

L'applicazione di una sequenza di regole di inferenza porta ad una derivazione: $KB \vdash \alpha$. Naturalmente ci si aspetta che ciò che si deriva da un insieme $KB$ sia vero in tutti i modelli di $KB$, ossia che: $KB \vDash \alpha \quad \iff \quad KB \vdash \alpha$.

### Logica del prim'ordine

La logica (classica) del prim'ordine estende la logica proposizionale con predicati, variabili, quantificatori universali e esistenziali.

# Chapter 8

# Logica Modale

La logica modale viene fatta risalire ad Aristotele sviluppata da filosofi interessati alla distinzione fra **verità necessarie** e **verità contingenti**. La verità contingente è un'affermazione che ora è vera, ma che potrebbe non più esserlo col passare del tempo. La verità necessaria è una verità assoluta, che non cambia, come il valore della $\sqrt{2}$. La sua formulazione *"moderna"* risale invece all'inizio degli anni '60 del secolo scorso ed è dovuta principalmente a Hintikka e Kripke. In particolare la **semantica della logica modale**, che è formulata in termini di **mondi possibili**, è dovuta ai due suddetti logici. Ogni mondo rappresenta una situazione/scenario considerato possibile dall'agente. Ciò che è vero in tutti i mondi si può dire che sia creduto dall'agente. Da un punto di vista filosofico, la proprietà fondamentale della logica modale è che tutto ciò che è vero in tutti i mondi è **necessariamente vero**. Ciò che è vero in qualche mondo è **possibilmente vero**. Come già accennato, la logica modale fornisce "**operatori modali**" come *Belief* che possono avere formule come argomenti. Possiamo quindi considerarla, almeno dal punto di vista sintattico, come una logica classica estesa con operatori modali.

## 8.1 Logica Modale Proposizionale

### 8.1.1 Sintassi

La logica modale proposizionale estende la sintassi della Logica Classica 7 proposizionale con due nuovi operatori che posono formare una formula complessa:

- $\Box\, \phi$ serve per catturare il concetto di **verità come necessità**;
- $\Diamond\, \phi$ serve per catturare il concetto di **verità come possibilità di accadere**.

Questi due operatori sono l'uno il duale dell'altro, quindi se volessimo ridurre all'osso la sintassi potremmo tenerne solamente uno dei due:

$$\Box\, \phi \equiv \neg\Diamond\neg\phi \qquad\qquad \Diamond\phi \equiv \neg\Box\neg\phi$$

Detto a parole la prima risulta: *se è necessario che $\phi$ sia vera, qualunque cosa sia $\phi$, non è possibile che $\phi$ sia falsa*. Mentre la seconda: *se è possibile che $\phi$ sia vera, qualunque cosa sia $\phi$, non è necessario che $\phi$ sia falsa*.

### 8.1.2 Semantica

Un **modello della logica modale** è dato come un insieme di mondi possibili. Ogni **mondo** è costituito da un **insieme di proposizioni** che sono considerate vere in quel mondo. Si noti che ogni singolo mondo può essere visto come un modello della logica proposizionale. La struttura del modello è descritta da una **relazione binaria di accessibilità** che collega coppie di mondi. Più precisamente un modello $M$ è una tripla $\langle W,\ R,\ L\rangle$ dove:

- $W$ è un insieme di mondi;
- $R \subseteq W \times W$ è una relazione di accessibilità non simmetrica tra mondi, usata per definire la semantica degli operatori modali:
  - la formula $\Box\ \phi$ è vera in un mondo $w \in W$ se $\phi$ è vera in tutti i mondi accessibili da $w$. La formula risulta vera anche se il mondo $w$ non ha nessun successore, independentemente da $\phi$;
  - la formula $\Diamond\ \phi$ è vera in un mondo $w \in W$ se $\phi$ è vera in qualche mondo accessibile da $w$;
- $L:\ W \to 2^{|P|}$ funzione che restituisce l'insieme delle proposizioni vere in ogni mondo $w \in W$.

Estendiamo il concetto **soddisfacibilità** di una formula in riferimento ai due nuovi operatori modali. La soddisfacibilità di una qualsiasi formula $\phi$ è definita rispetto ad un modello $M$ e ad un mondo $w$ di quel modello con la notazione $M \vDash_w \phi$:

- $M \vDash_w \Box\ \phi \quad \iff \quad (\forall w':\ R(w,w') \implies M \vDash_{w'} \phi)$;
- $M \vDash_w \Diamond\ \phi \quad \iff \quad (\exists w':\ R(w,w') \implies M \vDash_{w'} \phi)$.

Una formula $\phi$ è **valida** se è vera in tutte le coppie modello/mondo. La nozione di validità è analoga a quella di tautologia nella logica proposizionale classica. I modelli della logica modale sono spesso chiamati **Kripke structures**. Un modello, per facilità di comprensione, può essere rappresentato con un grafo i cui nodi contengono insiemi di proposizioni vere:

La coppia $\langle W,\ R\rangle$ è invece chiamata **Kripke frame**, non considero ciò che è vero nei mondi, ottengo un semplice grafo standard, tolgo il contenuto dai mondi.

### 8.1.3 Proprietà classiche

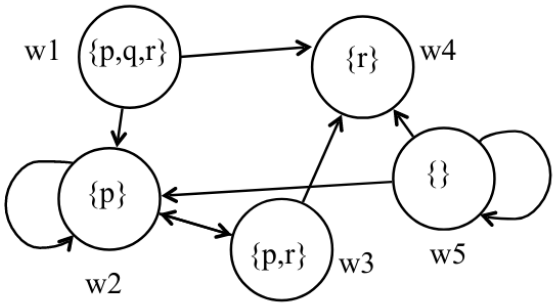
Dalla semantica della logica modale seguono tre proprietà fondamentali:

- **Assioma** $K$: $\Box(\phi \implies \psi) \implies (\Box\ \phi \implies \Box\ \psi)$. Questo è un assioma fondamentale, sempre vero, che prende il nome in onore di Kripke e ci permette di portare dentro l'operatore $\Box$;
- **Necessitation**: $\phi \vdash \Box\ \phi$. Questa è una regola di inferenza che afferma che se $\phi$ è valida e quindi una tautologia allora è necessariamente vera;
- **Distributività**: ne abbiamo due tipi diversi a seconda dell'operatore:
  - $\Box\ (\phi \wedge \psi) \equiv (\Box\ \psi \wedge \Box\ \psi)$, ma non vale $\Box\ (\phi \vee \psi) \neq (\Box\ \psi \vee \Box\ \psi)$;
  - $\Diamond\ (\phi \vee \psi) \equiv (\Diamond\ \psi \vee \Diamond\ \psi)$, ma non vale $\Diamond\ (\phi \wedge \psi) \neq (\Diamond\ \psi \wedge \Diamond\ \psi)$.

W = {w1, w2, w3, w4, w5}

R = {<w1,w2>, <w1,w4>, <w2,w2>, <w2,w3>, <w3,w2>, <w3,w4>, <w5,w2>, <w5,w4>, <w5,w5>}

L = {<w1,{p,q,r}>, <w2,{p}>, <w3,{p,r}>, <w4,{r}>, <w5, ϕ>}




Si noti che la relazione R può collegare un mondo a se stesso con un loop, ad esempio <w2,w2>

Figure 8.1: Modello della Logica Modale

### 8.1.4 Logica modale per agenti

Ci possiamo chiedere cosa c'entri questa logica con le problematiche degli agenti, visto che la logica modale è la logica del necessario e del possibile. La risposta è molto semplice: si può dare all'operatore modale $\Box$ (e al suo duale $\Diamond$) un significato diverso da quello standard (necessario o possibile) che modelli una proprietà degli agenti, senza modificare la semantica basata sui mondi possibili. Ad esempio, volendo ragionare sulle credenze, potremmo attribuire all'operatore $\Box$ il significato di *Belief* di un agente. Per chiarezza si può addirittura sostituire $\Box$ con **B** mantenendo ovviamente le proprietà semantiche. Seguendo questo principio nascono diverse logiche modali per agenti:

- **Logica Epistemica** 9 (conoscenza e credenze): prevede l'uso di $K_a\phi$ per esprimere che l'agente $a$ sa che $\phi$ è vera e $B_a\phi$ per dire che crede che sia vera $\phi$;
- **Logiche BDI**: 11 prevede l'uso di $B_a\phi$ per dire che l'agente $a$ crede che sia vera $\phi$, $D_a\phi$ per dire che desidera che $\phi$ sia vera e $I_a\phi$ per dire che ha intenzione di rendere vera $\phi$;
- **Logiche deontiche** (obblighi e permessi): prevede l'uso di $O\phi$ per dire che $\phi$ è obbligatorio e $P\phi$ per dire invece che è permesso;
- **Logica Temporale** 10 (lineare): prevede l'uso di $X\phi$ per dire che $\phi$ sarà vero al prossimo istante temporale, $G\phi$ per dire che $\phi$ sarà sempre vero, $F\phi$ per dire che prima o poi diventerà vero e $\psi U\phi$ per dire che $\psi$ è vero almeno fino a quando non diventa vero $\phi$;
- **Logica dinamica** (delle azioni): prevede l'uso di $[\pi]\phi$ per dire che dopo l'esecuzione del piano $\pi$, $\phi$ sarà vero.

In particolare per modellare con la logica modale le diverse proprietà degli agenti (*conoscenze, credenze, tempo*, ...) può essere necessario modificare la logica in modo opportuno. Questo può essere fatto sia aggiungendo nuovi assiomi a quelli validi di base sia andando a dare

delle condizioni per i Kripke frame e quindi intaccare la struttura a grafo del modello. La **teoria della corrispondenza** afferma che questi due modi portano in maniera equivalente alle stesse versioni della logica modale. Con gli assiomi andiamo a scartare i modelli in cui gli assiomi non valgono, mentre impostando delle condizioni per i frame andiamo direttamente a restringere la classe dei modelli ammissibili e costruibili. Detto questo le proprietà che ci interessano sono:

| Simbolo | Nome | Proprietà del frame | Assioma |
|---|---|---|---|
| T | Riflessiva | $\forall w \in W \ (w,w) \in R$ | $\Box \phi \Longrightarrow \phi$ |
| D | Seriale | $\forall w \in W \ \exists w' \in W \ (w,w') \in R$ | $\Box \phi \Longrightarrow \Diamond \phi$ |
| 4 | Transitiva | $\forall w,w',w'' \in W$ $((w,w') \in R \ \wedge \ (w',w'') \in R) \Longrightarrow (w,w'') \in R$ | $\Box \phi \Longrightarrow \Box\Box \phi$ |
| 5 | Euclidea | $\forall w,w',w'' \in W$ $((w,w') \in R \ \wedge \ (w,w'') \in R) \Longrightarrow (w',w'') \in R$ | $\Diamond \phi \Longrightarrow \Box\Diamond \phi$ |

Table 8.1: Proprietà logica modale

Ovviamente queste proprietà possono essere combinate per formare 11 sistemi modali diversi, matematicamente sarebbero 16, ma alcuni sono tra loro equivalenti. Il sistema modale più usato è il $KDT45$ che usa tutte le proprietà. Per curiosità il problema della validità di questi sistemi modali è decidibile e la complessità è **PSPACE-complete**.

# Chapter 9

# Logica Epistemica

Una possibile evoluzione della Logica Modale 8 è la **logica epistemica** che è la logica della conoscenza e delle credenze. Innanzitutto introduciamo due operatori modali $K_a$ e $B_a$ che rappresentano le conoscenze (knowledge) e le credenze (belief) di un agente $a$. I due operatori hanno la stessa semantica di $\Box$ nel senso che per essere veri in un mondo $w$ devono essere veri in tutti i mondi da esso raggiungibili, ma si distingueranno in base alle proprietà che soddisfano. Di solito non si introducono altri operatori modali per esprimere la possibilità, dato che *necessario* e *possibile* sono duali e si può usare la notazione $\neg K_a \neg \phi$ e $\neg B_a \neg \phi$.

## 9.1 Onniscienza Logica

Un grosso problema in cui incorriamo quando usiamo la logica modale per modellare la conoscenza è quello dell'onniscienza logica che deriva dall'assioma $K$ e dalla proprietà di necessitation che caratterizzano ogni logica modale. Con la logica epistemica abbiamo infatti:

$$K_a(\phi \Longrightarrow \psi) \Longrightarrow (K_a\phi \Longrightarrow K_a\psi) \qquad \text{assioma K}$$
$$\phi \vdash K_a(\phi) \qquad \text{necessitation property}$$

Secondo la necessitation l'agente deve conoscere tutte le tautologie logiche e, secondo l'assioma $K$, deve conoscere tutto ciò che deriva dalle sue conoscenze. Questo vuol dire che la conoscenza dell'agente è chiusa rispetto alla conseguenza logica. Si parla di infinita conoscenza. Noi trascureremo questo problema e vedremo come usare la logica modale per modellare knowledge e belief.

## 9.2 Assiomi per Knowledge e Belief

Introduciamo nella logica epistemica gli assiomi importantanti $T, D, 4$ e $5$ che abbiamo già visto in Logica Modale 8, ma diamone un'interpretazione in termini di conoscenza e credenza:

| Simbolo | Nome | Assioma |
|---|---|---|
| T | Coerenza | $K\,\phi \Longrightarrow \phi$ |
| D | Non Contraddizione | $K\,\phi \Longrightarrow \neg K \neg \phi$ e $B\phi \Longrightarrow \neg B \neg \phi$ |
| 4 | Introspezione Positiva | $K\,\phi \Longrightarrow KK\,\phi$ |
| 5 | Introspezione Negativa | $\neg K\,\neg \phi \Longrightarrow K \neg K\,\neg \phi$ |

Table 9.1: Assiomi logica epistemica

Il significato degli assiomi è il seguente:

- $T$: ciò che l'agente conosce è vero, accettabile solo per la conoscenza, non per le credenze. Non vogliamo che un agente conosca qualcosa che è falso, ma accettiamo che l'agente creda vero qualcosa che è falso;
- $D$: la conoscenza e la credenza dell'agente sono non contradditorie, se l'agente conosce $\phi$ allora non conosce $\neg\phi$ allo stesso modo vale per la credenza;
- 4: l'agente sa di sapere quel che conosce;
- 5: l'agente sa di non sapere quel che non conosce.

# Chapter 10

# Logica Temporale

## 10.1 Ragionare su tempo e azioni

Le principali caratteristiche di un agente sono di essere situato in un ambiente e di agire su di esso nel corso del tempo sulla base delle percezioni e dei propri obiettivi. Per fare questo l'agente deve essere in grado di ragionare sul tempo e sulle azioni. Per quanto riguarda la logica temporale questa è un tipo di Logica Modale 8 e, date le sue molte varianti, consideriamo una logica temporale in cui il tempo è:

- **discreto**: *definito su istanti di tempo discreti*;
- **inizia**: *ha un istante iniziale che origina il tutto*;
- **infinito**: *dato l'istante iniziale si susseguono infiniti istanti di tempo discreti*.

Da questa definizione possiamo introdurre due tipi di logiche temporali proposizionali: **Linear-time logic**, **Branching-time logic**. Mentre per il ragionamento sulle azioni useremo la logica classica e il **calcolo delle situazioni**.

## 10.2 Ragionare sul tempo

### 10.2.1 Linear Temporal Logic

Come si evince dal nome stesso il tempo è visto come sequenza infinita di istanti di tempo.

#### Sintassi

Andiamo ad estendere la sintassi della logica proposizionale con 2 operatori modali sul tempo:

- **X**$\alpha$: *nel prossimo stato/istante temporale sarà vero* $\alpha$;
- $\alpha$ **U** $\beta$ : $\alpha$ *è vera almeno fino a quando non sarà vera* $\beta$. Preso l'istante temporale $t$ in cui $\alpha$ diventa vera la formula sarà vera sse $\alpha$ continua ad essere verà almeno fino all'istante $t'$ in cui diventa vera $\beta$.

#### Semantica

La struttura del tempo è un insieme totalmente ordinato di istanti di tempo (isomorfo a $\mathbb{N}$). Un **modello** $M$ è una **struttura lineare** $\langle S, x, L \rangle$ dove:

- $S$ è l'insieme di stati che corrispondono ai mondi della logica modale standard;
- $x : \mathbb{N} \to S$ definisce la sequenza infinita di stati ordinati;
- $L : S \to 2^{|P|}$ definisce l'insieme delle preposizioni vere in uno stato.

#### Altri operatori

Dalla combinazione di **X** e **U** possiamo ricavare, se necessario, altri operatori modali come ad esempio:

- $\mathbf{F}\alpha \equiv \mathit{true}\ \mathbf{U}\ \alpha$, con significato: *$\alpha$ sarà prima o poi vera*;
- $\mathbf{G}\alpha \equiv \neg\ \mathbf{F}\ \neg\alpha$, con significato *$\alpha$ è sempre vera*, o più precisamente *non sarà mai vera $\neg\alpha$*.

### 10.2.2 Computation Tree Logic

Come si evince dal nome stesso il tempo è visto come albero dove ogni istante (nodo interno) può avere più istanti successori o figli (**alberi infiniti**). Abbreviata con $CTL^*$ ci permette di formulare **path formulas** e **state formulas**. Le prime riguardano i cammini di lunghezza infinita della struttura temporale ad albero e sono simili alle formule della $LTL$. Le state formulas riguardano invece il ragionamento su uno stato e su tutti i cammini da lui uscenti. La $CTL^*$ contiene la $LTL$ che viene usata per modellare i cammini dell'albero.

#### Sintassi

**State formulas** Oltre alla classica sintassi della logica proposizionale troviamo 2 operatori modali:

- $\mathbf{A}\pi$ : *per tutti i cammini uscenti dallo stato analizzato deve valere $\pi$ path formula*;
- $\mathbf{E}\pi$ : *deve esistere almeno un cammino uscente dallo stato analizzato in cui vale $\pi$ path formula*.

Formule proposizionali atomiche o complesse sono delle state formulas, che riguardano solo l'istante temporale in cui ci troviamo, mentre i 2 nuovi operatori modali possono essere applicati solamente a path formulas. **A** e **E** sono duali.

**Path formulas** Oltre alla classica sintassi della logica proposizionale troviamo i due operatori modali **X** e **U** con la stessa semantica della logica $LTL$. Una qualunque state formula può essere una path formula, così come ovviamente l'applicazione dei due nuovi introdotti su una o più path formulas.

**Semantica**

Un **modello** $M$ è una tripla $\langle S, T, L \rangle$ dove $S$ e $L$ sono come in $LTL$, mentre, informalmente, $T$ è un albero infinito i cui nodi sono stati. Dato un modello $M$, la semantica di una state formula $\alpha$ si riferisce ad uno stato $s$ $(M, s \vDash \alpha)$, mentre la semantica di una path formula $\pi$ si riferisce a un cammino $x$ $(M, x \vDash \pi)$.

Le logiche temporali sono spesso usate per verificare la correttezza di sistemi concorrenti (ad esempio sistemi multiagente) con la tecnica del Model Checking 12. Visto che il model checking affronta problemi di complessità elevata, diventa rilevante l'efficienza della implementazione. Per questo, in pratica, non si utilizza la logica $CTL^*$ ma si fa uso di una logica chiamata $CTL$ che restringe la sintassi risultando più efficiente. La sintassi della $CTL$ non permette la combinazione e l'annidamento diretto degli operatori linear-time nella composizione delle path formulas. I due operatori modali **X** e **U** o loro derivazioni possono quindi essere applicati direttamente solo a state formulas. In pratica, $LTL$ e $CTL$ sono utilizzate a seconda del tipo di problema che si intende risolvere perché hanno caratteristiche diverse e non sono confrontabili.

## 10.3 Ragionare sulle azioni

La prima trattazione di questo argomento risale al **calcolo delle situazioni** introdotto da McCarthy. Il ragionamento sulle azioni si basa sulla logica classica e gli aspetti principali del calcolo delle situazioni che vengono presi in considerazione sono:

- **Situazione**: stato del mondo in un qualche istante di tempo. $S_0$ situazione iniziale, $do(a, s)$ denota una situazione risultante dall'esecuzione dell'azione $a$ nello stato o situazione $s$;
- **Fluente**: proposizione il cui valore di verità può variare da una situazione ad un'altra;
- **Azione**: causa un cambiamento nello stato $s$ del mondo.

Ogni azione è descritta da due assiomi: un **assioma di possibilità** o precondizioni che dice quando è possibile eseguire l'azione e un **assioma di effetto** o conseguenza che specifica quello che accade quando un'azione possibile è eseguita.

### 10.3.1 Frame Problem

McCarthy e Hayes sono stati i primi ad osservare che un'azione influenza solo un numero limitato di fluenti. *Come esprimere in modo parsimonioso che tutto il resto del mondo non cambia?*

Questo è chiamato **frame problem**, perché corrisponde a ciò che succede in un film quando si passa da un fotogramma ad un altro. Un approccio è quello di scrivere assiomi di frame che affermino esplicitamente ciò che non cambia. Ad esempio lo spostamento di un agente non modifica la posizione di alcun oggetto, a meno che l'agente non lo stia trasportando. Questo approccio è molto costoso dal punto di vista computazionale e quindi sono state proposte soluzioni più economiche.

**Successor State Axioms**

Reiter ha proposto una soluzione del frame problem introducendo i **Successor State Axioms**, che fanno riferimento ad un fluente per volta. GOLOG è uno dei Linguaggi Logici per Agenti 14 basato sul calcolo delle situazioni, in cui le azioni primitive sono specificate dandone le precondizioni ed effetti rappresentati come successor state axioms.

# Chapter 11

# Logica per Agenti BDI

## 11.1 Contesto

Secondo Wooldridge e Ciancarini i metodi formali, come la logica, possono essere usati per:

- specificare un agente o un sistema di agenti;
- programmare direttamente sistemi di agenti (Linguaggi Logici per Agenti);
- verificare per un agente o per un sistema di agenti che certe proprietà ottenute siano corrette e desiderate (Model Checking).

Ora ci concentriamo sul primo punto e vogliamo fare una modellazione logica per gli agenti che combini tutti i loro diversi aspetti caratterizzanti: *stati mentali*, *azioni*, *ecc...* Ci aspettiamo inoltre che la logica dell'agente ci permetta di rappresentare i loro aspetti dinamici/temporali. Una completa teoria degli agenti, espressa in linguaggio logico, dovrebbe definire come tutti gli attributi/caratteristiche degli agenti siano tra loro correlati. I due più significativi approcci alla modellazione logica di agenti BDI sono:

1. *Cohen and Levesque's intention logic*;
2. *Rao and Georgeff's BDI logic*.

Entrambe le teorie sono in grado di rappresentare stati mentali di agenti come *beliefs, goals, intentions e actions* e di ragionare su aspetti dinamici per mezzo di logiche temporali e multimodali.

## 11.2 Cohen & Levesque intention logic

Con il loro articolo *"Intention is choice with commitment"* gli autori propongono una logica che riguarda principalmente il **rational balance** relativo agli stati mentali e alle azioni di agenti autonomi. L'obiettivo principale è di esplorare le relazioni che le intenzioni hanno nel mantenere questo bilanciamento razionale in un agente. Seguendo e sostenendo le teorie di Bratman, Cohen e Levesque ritengono che il comportamento razionale dovrebbe essere analizzato in termini di beliefs, desires and intentions (BDI). Per far questo iniziano elencando le proprietà che le intenzioni di un agente razionale dovrebbe possedere:

- le intenzioni pongono dei problemi per gli agenti, che devono determinare modi di soddisfarle;
- le intenzioni filtrano altre intenzioni in conflitto con quelle attuali;
- gli agenti "tracciano" il successo delle loro intenzioni, e sono disposti a tentare di nuovo se il loro tentativo fallisce;
- gi agenti credono che le loro intenzioni siano possibili;
- gli agenti non credono che non riusciranno a soddisfare le loro intenzioni;
- in certe circostanze, gli agenti credono che riusciranno a soddisfare le loro intenzioni;
- gli agenti non si aspettano tutti i side-effects delle loro intenzioni, non hanno come intenzione tutte le conseguenze delle attuali intenzioni. Bratman sostiene che ciò che uno intende è, approssimativamente, un sottoinsieme di ciò che uno sceglie.

### 11.2.1 Sintassi

La logica è multi-modale basata su 4 operatori modali di base di cui 2 su stati mentali e 2 su azioni:

- $(BEL\ \ i\ \ \phi)$: l'agente $i$ crede che la formula $\phi$ sia vera;
- $(GOAL\ \ i\ \ \phi)$: l'agente $i$ ha come goal (desire) che $\phi$ sia vera;
- $(HAPPENS\ \ \alpha)$: l'azione $\alpha$ sarà prossima a capitare;
- $(DONE\ \ \alpha)$: l'azione $\alpha$ è appena accaduta.

L'azione $\alpha$ può essere sia singola (*evento primitivo*) che complessa/composta da sequenza di azioni base.

### 11.2.2 Semantica

Come ogni Logica Modale anche questa basa la semantica sui mondi possibili che si appoggiano però ad una Logica Temporale lineare. I mondi infatti non sono classici insiemi di proposizioni vere nel mondo, bensì sequenze discrete di eventi che si estendono all'infinito nel passato e nel futuro. I mondi sono connessi attraverso le relazioni $BEL$, $GOAL$, ecc... mentre il ragionamento in un mondo è basato sulla logica temporale LTL.

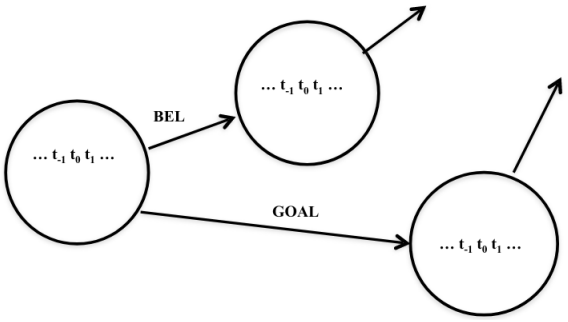



Figure 11.1: Rappresentazione semantica Intention Logic

Per stabilire la verità di formule dovrò, dato un modello, prendere in considerazione un suo mondo ed un suo istante temporale. La semantica di $BEL$ e $GOAL$ è data nel solito modo, assegnando ad ogni agente una *belief accessibility relation* ($KD45$), e una *goal accessibility relation* ($KD$). Per coerenza di agente intelligente si assume che la *goal relation* di ogni agente sia un sottoinsieme della sua *belief relation*. Dato che entrambi soddisfano l'assioma $K$ di Kripke entrambi gli operatori modali hanno la semantica dell'operatore modale $\Box$.

### 11.2.3 Nuovi operatori modali

Dai 4 operatori modali di base e da quelli della logica LTL se ne possono definire altri:

- operatori modali temporali:
  - $(LATER\ \ \phi) \equiv \neg\phi\ \wedge\ \mathbf{F}\phi$ : significa quindi che per ora vale $\neg\phi$, ma prima o poi diventerà vera;
  - $(BEFORE\ \ \phi\ \ \psi) \equiv \forall\alpha.\ (HAPPENS\ \ \alpha;\psi?)\ \Rightarrow \exists\beta.\ (\beta \leq \alpha) \wedge (HAPPENS\ \ \beta;\phi?)$ : dove ? è l'operatore di test e $(\beta\ \leq\ \alpha)$ significa che $\beta$ è una sottosequenza di azioni di $\alpha$. $(BEFORE\ \ \phi\ \ \psi)$ significa quindi che per ogni sequenza di azioni $\alpha$ che portano al verificarsi di $\psi$ esiste una sottosequenza $\beta$ che realizza $\phi$;
- operatori modali mentali usando anche gli operatori appena introdotti:
  - $(A\text{-}GOAL\ \ i\ \ \phi) \equiv (GOAL\ \ i\ \ (LATER\ \ \phi)) \wedge (BEL\ \ i\ \ \neg\phi)$ : l'agente $i$ ha $\phi$ come **achievement goal**, ossia crede che $\phi$ sia ad ora falso, ma desidera che in futuro diventi vero;
  - $(P\text{-}GOAL\ \ i\ \ \phi) \equiv (A\text{-}GOAL\ \ i\ \ \phi) \wedge$
    $[BEFORE\ \ ((BEL\ \ i\ \ \phi) \vee (BEL\ \ i\ \ \mathbf{G}\neg\phi))\ \ \neg(GOAL\ \ i\ \ (LATER\ \ \phi))]$ : l'agente $i$ ha come **persistent goal** $\phi$ se $\phi$ è un achievement goal e prima di abbandonare il goal o crede di averlo raggiunto o crede che non sia raggiungibile (mai vero nel futuro);
- operatori modali sulle azioni usando tutto ciò che è stato introdotto:

– $(INTEND\ i\ \alpha) \equiv (P\text{-}GOAL\ i\ [DONE\ i\ (BEL\ i\ (HAPPENS\ \alpha))?;\alpha])$ : l'agente $i$ intende compiere l'azione $\alpha$ se ha un goal persistente di raggiungere uno stato in cui lui aveva appena creduto di stare per eseguire $\alpha$, e subito dopo $\alpha$ viene eseguita. Il criterio che l'agente sia committed a credere di stare per eseguire l'azione $\alpha$, evita che l'agente sia tenuto ad eseguire l'azione accidentalmente o senza pensarci.

## 11.3 Rao & Georgeff BDI logic

La differenza principale rispetto al precedente approccio è che qui compare direttamente l'operatore modale per le intenzioni e non deve essere ricavato da quelli base. Inoltre i mondi possibili si appoggiano ad una logica temporale branching time e non lineare.

### 11.3.1 Sintassi

Il linguaggio estende le formule di stato di $CTL^*$ con:

- $BEL(\phi)$, $GOAL(\phi)$ e $INTEND(\phi)$ con solito significato mentale rispetto ad una formula $\phi$;
- $succeds(e)$, $fails(e)$, $does(e)$ con intuitivo significato per un evento $e$. Per ogni possibile condizione di un evento esiste anche la forma passata ($succeeded(e)$, $failed(e)$, $done(e)$) con riferimento ad un particolare istante temporale $t_i$ in cui è avvenuto.

### 11.3.2 Semantica

I mondi sono *branching time temporal structures* chiamate **time trees**. I *rami* di un time tree rappresentano le **scelte disponibili** all'agente in ogni momento. E' possibile distinguere tra *esecuzioni riuscite* di eventi e i loro *fallimenti*. Le formule temporali sono simili a quelle di $CTL^*$ ossia ci sono state formulas e path formulas. *Beliefs*, *goals* and *intentions* sono modellate nel solito modo. In ogni situazione c'è un insieme di mondi *(belief/goal/intention)-accessible*.

### 11.3.3 Commitments

Con questo linguaggio è possibile descrivere diverse strategie di commitment grazie agli operatori modali. A seconda di usare una formula $\phi$ o un evento $e$, si possono modellare commitments per azioni (*i.e. means*) o condizioni che saranno vere in futuro (*i.e. ends*):

- **Blind/Fanatical**: $INTEND(\mathbf{AF}\phi) \Rightarrow \mathbf{A}(INTEND\ (\mathbf{AF}\phi)\ \mathbf{U}\ BEL(\phi))$ per descrivere che un agente mantiene le proprie intenzioni fino a quando non crede di averle soddisfatte;
- **Open Minded**: $INTEND(\mathbf{AF}\phi) \Rightarrow \mathbf{A}(INTEND(\mathbf{AF}\phi)\ \mathbf{U}\ (BEL(\phi) \vee \neg GOAL(\mathbf{EF}\phi)))$ per descrivere un agente che mantiene le sue intenzioni finché o crede di averle soddisfatte o non sono più suoi goal;

- **Single Minded**: $INTEND(\mathbf{AF}\phi) \Rightarrow \mathbf{A}(INTEND(\mathbf{AF}\phi)\ \mathbf{U}\ (BEL(\phi) \vee \neg BEL(\mathbf{EF}\phi)))$ per descrivere che un agente mantiene le sue intenzioni finché o crede di averle soddisfatte o crede che non possano essere realizzate.

# Chapter 12

# Model Checking

## 12.1 Definizione

ll Model checking è un metodo per la verifica di proprietà di agenti e di sistemi multiagente o, più in generale, di sistemi concorrenti prevalentemente non terminanti. Come possiamo intuire dal nome stesso il model checking è un **approccio semantico**: dato un modello $M$ di una logica $L$ e una formula $\phi$ di $L$ dobbiamo capire se $\phi$ sia o meno **valida nel modello** $M$. In pratica, le proprietà da verificare sono basate generalmente su logiche temporali $L$ e sulla stretta relazione fra i modelli delle logiche temporali e gli automi a stati finiti che descrivono le computazioni di sistemi concorrenti. Il comportamento di un sistema concorrente $\pi$ può infatti essere formulato come un **state transition system** consistente di un insieme di stati, un insieme di transizioni fra gli stati e una funzione che che associa ad ogni stato un insieme di proposizioni vere in quello stato. Risulta quindi simile ad una struttura di Kripke $M_\pi$ ossia un modello della logica modale temporale differendosi da essa nel fatto che gli archi che collegano i nodi/stati sono etichettati con azioni. Un esempio di questo è l'agente microonde di 12.1.

## 12.2 Model Checking per LTL

Faremo riferimento solamente alla logica temporale LTL. Supponiamo di avere un sistema $\pi$ formulato come state transition system e una formula LTL $\phi$ che descrive una proprietà che vogliamo verificare. Per mostrare che $\phi$ vale per $\pi$, dobbiamo dimostrare che $\phi$ è vera per ogni esecuzione di $\pi$. Siccome sappiamo che un modello in LTL è una sequenza infinita di istanti temporali, dimostrare che $\phi$ è vera per ogni esecuzione di $\pi$, significa dimostrare che ogni cammino infinito di $\pi$ è un modello di $\phi$. Di solito, la dimostrazione che una formula $\phi$ è valida, ossia è vera per tutti i cammini infiniti del modello, viene fatta per **refutazione**: dimostrare che $\neg\phi$ è insoddisfacibile nel modello, ossia non esiste nessuna computazione che soddisfi $\neg\phi$. In altre parole, se si trova un cammino infinito che soddisfa $\neg\phi$, questo costituisce un **controesempio** che contraddice la validità di $\phi$.

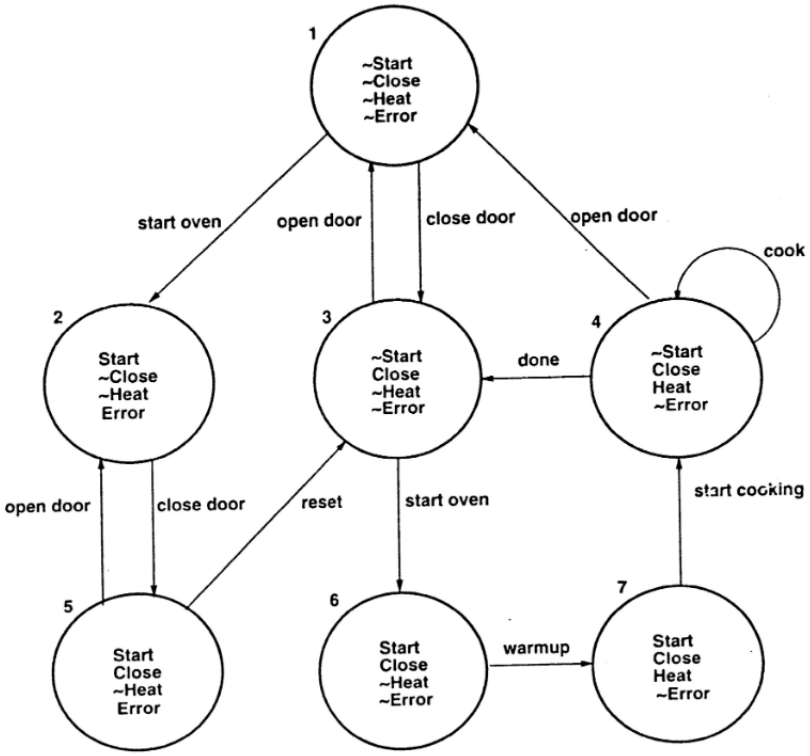



Figure 12.1: Agente Forno a Microonde

### 12.2.1 LTL e Automi su stringhe infinite

Se la proprietà da dimostrare è rappresentata con una formula LTL, il model checking può essere realizzato usando automi di Büchi, ossia automi che accettano stringhe infinite.

Un automa di Büchi ha esattamente la stessa struttura di un tradizionale automa a stati finiti, ma accetta stringhe infinite. L'automa $\langle \Sigma, S, \delta, S_0, F \rangle$ consiste di un alfabeto $\Sigma$, un insieme di stati $S$, una funzione di transizione $\delta \subseteq S \times \Sigma \times S$, un insieme di stati iniziali $S_0$ e un insieme di **stati di accettazione** $F$. Informalmente, una stringa infinita (run) $w$ è accettata dall'automa se è compatibile con $\delta$ (è un cammino infinito nel grafo dell'automa) e almeno uno stato in $F$ appare infinite volte in $w$.

Uno dei vantaggi principali di usare gli automi di Büchi per il model checking è che il sistema modellato e le proprietà da verificare sono rappresentabili allo stesso modo, ossia con automi di Büchi. Infatti il transition system di un sistema $\pi$, ossia il modello, può essere visto come un automa di Büchi $\mathcal{B}(\pi)$ i cui stati sono tutti di accettazione. Inoltre, esiste un algoritmo per tradurre una formula LTL $\phi$ in un automa di Büchi $\mathcal{B}(\phi)$ che accetta esattamente le sequenze infinite che sono modelli di $\phi$.

### 12.2.2 Procedura per il Model Checking

Per fare Model Checking si eseguono le seguenti operazioni:

1. siano dati il transition system $\pi$ e la formula LTL $\phi$ da verificare;

2. costruire i due automi di Büchi $\mathcal{B}(\pi), \mathcal{B}(\neg\phi)$;
3. costruire l'automa di Büchi che accetta l'intersezione dei linguaggi accettati da $\mathcal{B}(\pi)$ e $\mathcal{B}(\neg\phi)$ (il prodotto dei due automi di Büchi);
4. ogni run dell'intersezione è sia un run infinito di $\pi$ sia un modello di $\neg\phi$;
5. se l'intersezione è vuota, allora $\phi$ vale per $\pi$, altrimenti un run dell'intersezione fornisce un controesempio.

L'ultimo passo può essere eseguito in tempo circa lineare rispetto alla dimensione dell'automa, mentre la costruzione dell'automa intersezione può essere esponenziale rispetto alla dimensione della formula. Per trovare un controesempio è sufficiente trovare un cammino dell'automa intersezione che termina con un loop.

## 12.3 Conclusioni

Oltre alla verifica di proprietà, il model checking può essere utilizzato per altri tipi di ragionamento, come ad esempio la pianificazione (con cammini infiniti): dato l'automa che rappresenta un sistema computazionale $\pi$ e una formula $\phi$ che rappresenta il goal, tutti i run infiniti dell'automa prodotto di $\mathcal{B}(\pi)$ e $\mathcal{B}(\phi)$ soddisfano $\phi$ e quindi rappresentano un piano sequenziale. Se si vuole ottenere un piano di lunghezza finita con questo approccio, è sufficiente aggiungere nello stato finale un loop fittizio.

### 12.3.1 Model Checking in SPIN

SPIN è un tool per la verifica di sistemi concorrenti basato su tecniche di model checking:

- la specifica del modello è data nel linguaggio Promela, che consente di definire sistemi distribuiti mediante un insieme di processi formulati in un codice pseudo C, che consente l'uso di primitive di sincronizzazione e scambio di messaggi;
- la proprietà da verificare è una formula in linear time temporal logic (LTL).

### 12.3.2 Model Checking con CTL

Oltre al model checking basato su LTL, sono stati proposti e sviluppati numerosi model checker basati su approcci diversi. In particolare, possiamo citare l'approccio per verificare formule CTL, basandosi direttamente sulla decomposizione della struttura di Kripke in componenti fortemente connessi. Questo approccio è più efficiente di quello che usa LTL, ma, come avevamo osservato parlando di logiche temporali, le formule trattate da LTL e CTL non sono confrontabili.

# Chapter 13

# Linguaggi e Architetture per Agenti BDI

## 13.1 Procedural Reasoning System

Il PRS, sviluppato da Georgeff, è stata una delle prime architetture per agenti che abbia fatto uso del paradigma BDI. Questa architettura, che può essere adattata a diversi tipi di linguaggi per essere implementata, ha avuto molto successo ed è stata usata in molte applicazioni multi-agenti come il controllo del traffico aereo, sistemi diagnostici, ecc...

### 13.1.1 Architettura PRS

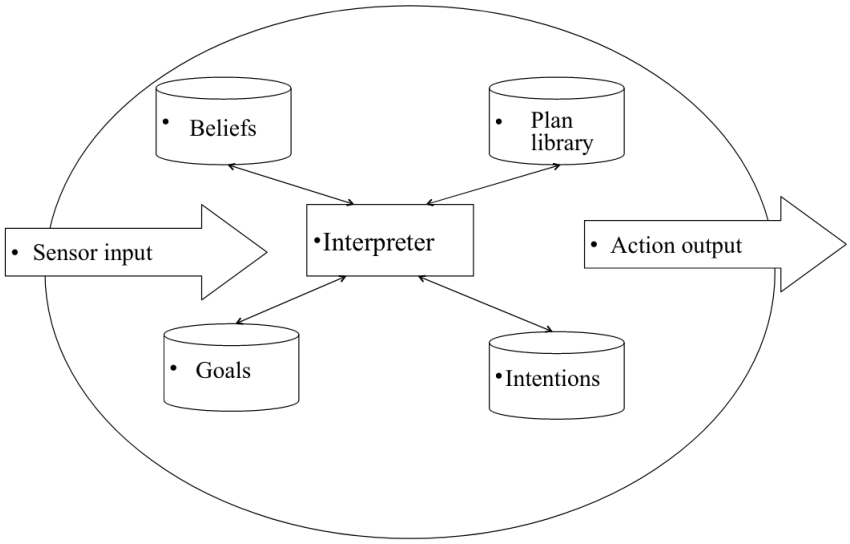



Figure 13.1: Architettura PRS

Gli input del sistema sono eventi, ricevuti attraverso una **coda degli eventi**, che possono essere:

- **esterni**: percepiti dall'ambiente;
- **interni**: frutto dell'aggiunta o cancellazione di beliefs o goals.

Gli output sono azioni esterne o interne, che sono eseguite dall'interprete. I beliefs sono rappresentati come fatti Prolog, ossia come atomi della logica del prim'ordine.

### 13.1.2 Piani PRS

In PRS gli agenti non pianificano, ma fanno riferimento ad una libreria di piani predefiniti dal programmatore. I componenti di un piano sono:

- *nome*;
- *condizione di invocazione*: l'evento che fa triggerare il piano;
- *precondizioni*: le condizioni che devono valere prima di avviare l'esecuzione del piano;
- *body*: è una sequenza di **simple plan expressions**, ossia:
  - azione atomica o
  - sottogoal.

Se ci sono più piani possibili/eseguibili per ottenere dei goals o sottogoals e quello che viene scelto si "pianta" viene scartato a run-time dall'interprete che ne seleziona un altro.

### 13.1.3 Interprete PRS

Per raggiungere un dato goal, l'agente formula l'intenzione di raggiungere questo obiettivo, cioè sceglie un piano applicabile *"attivato"* dal goal. Questo piano diventa un'intenzione ed è aggiunto alla **intention structure**. Quindi le intenzioni sono piani in fase di esecuzione. Ad ogni passo del loop principale, l'interprete sceglie un piano (parzialmente eseguito) nella intention structure e ne esegue un solo passo. Se ci sono molte opzioni disponibili, l'interprete può scegliere quella con massima utilità, o entrare in un ragionamento di metalivello usando "*metalevel plans*". I passi principali del **control loop** del PRS sono:

1. aggiorna beliefs e goals secondo gli eventi nella *event queue*;
2. osservare quali piani sono triggerati dai cambiamenti dei goals e beliefs;
3. uno o più piani applicabili sono scelti e messi nella *intention structure*;
4. il **deliberator** sceglie una intenzione (task) dalla intention structure;
5. viene eseguito un passo di quel task. Questo può risultare in:
   - esecuzione di una azione primitiva, o
   - scelta di un nuovo subgoal che viene inserito nella event queue.

Dato che l'interprete ad ogni iterazione sceglie una intenzione (task) e ne esegue un passo, l'esecuzione dei tasks può sembrare "interleaved" come fosse in un sistema multi-threaded. Per tenere traccia di questo, ogni intenzione (task) è implementata, come al solito, come uno stack di frames, che descrive uno stato intermedio dell'esecuzione del task.

## 13.2 AgestSpeak(L)

Il linguaggio AgentSpeak(L) è stato proposto da Rao come linguaggio di programmazione per agenti BDI che consente di essere scritto e interpretato in modo simile a programmi logici basati su clausole di Horn. Questo linguaggio è un tentativo di formalizzare linguaggi BDI mediante una **semantica operazionale**. Il linguaggio AgentSpeak(L) ha dato origine al linguaggio JASON che consente di realizzare sistemi multiagente.

## 13.3 Agent-Oriented Programming

Con l'articolo "*Agent-oriented Programming*" Shoham propone un nuovo paradigma di programmazione che promuove una visione sociale della computazione in cui più agenti interagiscono uno con l'altro. L'articolo presenta:

- un linguaggio formale AOP per descrivere stati mentali;
- un linguaggio di programmazione Lisp like AGENT-0 per definire agenti.

Possiamo considerare AOP come basato su un paradigma BDI.

### 13.3.1 AOP - Categorie mentali

Ci sono due categorie mentali principali:

- **Belief**: $B_a^t\ \phi$ l'agente $a$ crede $\phi$ al tempo $t$;
- **Obligation**: $OBL_{a,b}^t\ \phi$ l'agente $a$ è obbligato (o impegnato) dall'agente $b$ al tempo $t$ riguardo $\phi$. $\phi$ può essere un fatto che rappresenta un'azione. Le azioni infatti non sono distinte dai fatti: la presenza di una azione è rappresentata dal fatto corrispondente.

Possiamo poi definire categorie mentali derivate:

- **Decision**: $DEC_a^t\ \phi = OBL_{a,a}^t\ \phi$ dove la decisione o scelta è tratta come obbligazione verso se stessi.

Un'ulteriore categoria che non è però un costrutto mentale è:

- **Capability**: $CAN_a^t\ \phi$ che rappresenta il fatto che l'agente $a$ è in grado di compiere $\phi$ all'istante $t$.

Le formule, come visto, fanno riferimento esplicitamente al tempo mediante apici.

### 13.3.2 AGENT-0

Un programma in AGENT-0 si riferisce ad un singolo agente, il cui nome è implicito in tutte le istruzioni. I principali costrutti sono:

- **Facts**: $(t \ \ atom)$ al tempo $t$ vale $atom$;
- **Private Actions**: $(DO \ \ t \ \ p\text{-}action)$ esegui al tempo $t$ l'azione $p\text{-}action$;
- **Communicative Action**: che possono essere di due tipi diversi:
  1. $(INFORM \ \ t \ \ a \ \ fact)$ informa al tempo $t$ l'agente $a$ riguardo al fatto $fact$;
  2. $(REQUEST \ \ t \ \ a \ \ fact)$ richiedi al tempo $t$ all'agente $a$ il fatto $fact$.

Ogni agente in AGENT-0 ha 4 componenti fondamentali:

1. un insieme di *capabilities* che è in grado di fare;
2. un insieme di *beliefs* iniziali;
3. un insieme iniziale di *commitments*, azioni che vuole compiere;
4. un insieme di **commitment rules** che è il "programma" che stabilisce come si comporta l'agente. La commitment rule ha struttura:

   $$(COMMIT \ \ message\text{-}cond \ \ mental\text{-}cond \ \ (agent \ \ action)^*)$$

   Il "$*$" significa che ci possono essere più coppie agente-azione specificate. La **message-condition** è una combinazione logica di **message-patters** che hanno forma: $(From \ \ Type \ \ Content)$ dove $From$ è il nome del mittente, $Type$ è uno dei due possibili atti comunicativi e $Content$ è il contenuto (azione o fatto) del messaggio che dipende dal tipo. La **mental-condition** è invece una combinazione logica di **mental-patterns** presentati precedentemente mediante le categorie mentali di AOP.

#### AGENT-0 Interprete

*Beliefs*, *commitments*, e *capabilities* di un agente sono rappresentati ciascuno da un database. Beliefs e commitments possono cambiare ad ogni passo dell'esecuzione di un programma, mentre le capabilities e le commitment rules sono fisse. L'interprete esegue il seguente ciclo:

1. legge i messaggi correnti e aggiorna i beliefs e commitments (valutando le commitment rules);
2. esegue i commitment per il tempo attuale, eventualmente risultando in un cambiamento di beliefs.

I beliefs sono aggiornati o come risultato di essere informati o come risultato di eseguire un'azione privata. I commitments possono essere aggiunti mediante un messaggio se vengono rispettate le mental-conditions e le message-conditions.

# Chapter 14

# Linguaggi Logici Per Agenti

## 14.1 Introduzione

Secondo Wooldridge e Ciancarini i metodi formali, come la logica, possono essere usati nel contesto degli agenti per:

- *specificare sistemi* (Logiche BDI 11);
- *verificare sistemi* (Model Checking 12);
- *programmare direttamente sistemi* (Prolog).

Abbiamo già visto i primi due punti osserviamo ora il terzo. Questo richiede di usare un linguaggio logico che possa essere eseguito: **computational logic**. Un esempio classico è il Prolog, i cui programmi sono un insieme di clausole di Horn della logica classica e l'inferenza avviene mediante risoluzione SLD.

## 14.2 GOLOG

GOLOG (alGol in LOGic) è un linguaggio di programmazione basato sul **calcolo delle situazioni** progettato per domini logici. Prende in prestito dall'ALGOL molti costrutti standard della programmazione come la sequenza, il condizionale, le procedure ricorsive e il loop. Le situazioni possono essere viste come stati in cui è possibile eseguire certe azioni e che, a livello pratico, diventano argomenti dei predicati. ll linguaggio è usato per programmare robot di alto livello (cognitive robotics) e agenti software intelligenti.

### 14.2.1 Azioni Primitive

Le azioni primitive in GOLOG sono specificate dandone le *precondizioni* ed *effetti* come successor state axioms. Il **successor state axiom** è la proposta degli autori del GOLOG per risolvere il frame problem, andandone a definire uno per ogni fluente.

### 14.2.2 Azioni Complesse

GOLOG permette di definire azioni complesse usando l'abbreviazione $Do(\delta, s, s')$ dove $\delta$ è un'espressione di azione complessa. Intuitivamente $Do(\delta, s, s')$ è vera quando la situazione $s'$ è una situazione raggiungibile al termine di un'esecuzione di $\delta$ iniziata nella situazione $s$. La formalizzazione delle azioni complesse si basa sulla logica dinamica:

- **Primitive actions**: $Do(a, s, s') \equiv Poss(a, s) \ \wedge \ s' = do(a, s)$;
- **Test actions**: $Do(\phi?, s, s') \equiv \phi \ \wedge \ s' = s$;
- **Sequence**: $Do([\delta_1; \delta_2], s, s') \equiv \exists s'' \ t.c. \ Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$;
- **Nondeterministic choice**: $Do((\delta_1 | \delta_2), s, s') \equiv Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$;
- **Nondeterministic choice of action**: $Do((\pi x)\delta(x), s, s') \equiv \exists x \ t.c. \ Do(\delta(x), s, s')$.

E' possibile definire anche l'iterazione nondeterministica che risulta una definizione in logica del second'ordine dato che ho bisogno di quantificare sui predicati e procedure ricorsive la cui semantica è data come least fixed-point.

### 14.2.3 Eseguire programma GOLOG

Eseguire un programma GOLOG significa stabilire la seguente derivazione:

$$Axioms \ \models \ \exists s. \ Do(program, S_0, s)$$

ossia ci si chiede se dati gli assiomi che si possiedono esista una situazioni $s$ t.c. è raggiungibile eseguendo il programma a partire dalla situazioni di partenza $S_0$. Un programma eseguito con successo restituisce un legame per $s$:

$$s = do(a_n, do(a_{n-1}, ...))$$

sostanzialmente restituisce un piano, una sequenza di azioni che da $S_0$, applicate in successione, mi conducono a $s$. Per questo motivo GOLOG può essere usato per fare planning. Esattamente come per Prolog l'interprete del GOLOG è un theorem prover general-purpose (per la logica del second'ordine). Come per il Prolog, i programmi GOLOG sono eseguiti per i loro **side effects**, ossia per ottenere legami per le variabili quantificate esistenzialmente.

### 14.2.4 Estensioni del GOLOG

Dopo la sua descrizione nell'articolo GOLOG è stato esteso in vari modi:

- **CONGOLOG**: incorpora la concorrenza, trattando processi concorrenti con priorità diverse, interrupts, azioni esogene;
- **IndiGolog**: i programmi possono essere eseguiti incrementalmente per permettere azioni interleaved, planning, sensing, and exogenous events.

## 14.3 Concurrent METATEM

Concurrent METATEM proposto da Fisher è un linguaggio basato sulla esecuzione diretta di formule temporali LTL con istante $t_0$ iniziale e possibilità di ragionare all'infinito sia nel passato che nel futuro. L'idea del creatore era quella di creare l'equivalente di Prolog con la Logica Temporale 10. Un sistema Concurrent METATEM contiene un insieme di agenti che possono comunicare fra di loro via *asynchronous broadcast message passing*. Ogni agente è programmato dando una specifica del suo comportamento mediante logica temporale, che può essere eseguita direttamente. Dato che il tempo è modellato come una sequenza infinita di stati discreti, con un punto iniziale, il linguaggio fornisce operatori che riguardano sia il passato che il futuro.

### 14.3.1 Operatori del futuro

Abbiamo i classici operatori della logica LTL semplicemente con diversa sintassi:

- $O\ \phi$ : $\phi$ deve essere soddisfatta nel prossimo stato;
- $\phi$ **U** $\psi$ : $\phi$ deve essere vera fino a quando non si verifca $\psi$;
- $\diamondsuit\ \phi$ : $\phi$ deve essere vera in qualche stato futuro;
- $\square\ \phi$ : $\phi$ deve sempre essere vera negli stati futuri.

### 14.3.2 Operatori del passato

Possiamo definire degli operatori sul passato rigirando quelli definiti per il futuro:

- **O** $\phi$ : $\phi$ deve essere stata vera nello stato precedente;
- $\phi\ \mathcal{S}\ \psi$ : $\phi$ deve essere stata vera nel passato fino al verificarsi di $\psi$;
- $\blacklozenge\ \phi$ : $\phi$ deve essere stata vera in qualche stato del passato;
- $\blacksquare\ \phi$ : $\phi$ è sempre stata vera negli istanti del passato.

### 14.3.3 Programma METATEM

Un programma METATEM consiste di un insieme di regole con la forma:

$$\square\ (past\ \ and\ \ present\ \ formula \Rightarrow present\ \ or\ \ future\ \ formula)$$

dove deve sempre essere vero che date le formule presenti e passate valga o una formula nel presente o una nel futuro, con la parte destra vincolata ad essere una disgiunzione. Il linguaggio fornisce due meccanismi ortogonali per rappresentare la scelta:

- **indeterminatezza statica**: con l'operatore classico $\vee$;
- **indeterminatezza temporale**: con l'operatore sometime $\diamondsuit$ (in cui si cerca di soddisfare la formula legata a $\diamondsuit$ il prima possibile).

### 14.3.4 Interprete METATEM

L'interprete esegue in continuazione i seguenti passi:

1. verifica quali regole hanno gli antecedenti soddisfatti;
2. congiunge tutti i conseguenti di queste regole;
3. riscrive questa congiunzione in forma disgiuntiva e sceglie uno di questi disgiunti da eseguire;
4. se si trova una contraddizione è possibile fare backtrack ad una scelta precedente.

### 14.3.5 Comunicazione tra agenti

Come dice il nome del linguaggio stesso è stato pensato come linguaggio concorrente che permette agli agenti di comunicare tra loro. Ci sono tre categorie di predicati per i messaggi:

- **Predicati d'ambiente**: rappresentano messaggi che arrivano ad un agente da fuori (ambiente), senza sapere chi è il mittente;
- **Predicati di componente**: rappresentano messaggi che escono dall'agente verso l'ambiente;
- **Predicati interni**: messaggi che vengono spediti verso se stessi.

Ogni agente ha un'**interfaccia** che definisce come l'agente possa interagire con l'ambiente. L'interfaccia è composta da:

- un identificatore (nome) dell'agente in questione;
- una lista di predicati d'ambiente che rappresentano i tipi di messaggi che l'agente riconosce ai quali può rispondere;
- una lista di predicati di componente che rappresentano i tipi di messaggi che l'agente può produrre ed inviare verso altri agenti.

# Chapter 15

# Agenti Reattivi

## 15.1 Motivazione

Ci sono molti problemi irrisolti associati all'IA simbolica. Uno dei maggiori problemi non è tanto compiere azioni, quanto interfacciarsi con l'esterno, ovvero quale input ricevere e in che modo riceverlo. Questi problemi hanno portato alcuni ricercatori a discutere la realizzabilità dell'intero paradigma ed a sviluppare architetture reattive. Sebbene uniti dal credere che quelle assunzioni che sostengono l'IA mainstream siano sbagliate in qualche senso, i progettisti di agenti reattivi usano molte tecniche diverse tra di loro. Presentiamo quindi qua l'attività di uno dei più comunicativi ricercatori nel settore: Rodney Brooks.

## 15.2 Architettura Reattiva

Brooks ha proposto tre tesi fondamentali:

1. Un comportamento intelligente può essere generato **senza rappresentazione esplicita** del tipo proposto dall'IA simbolica, ossia senza l'uso di linguaggi logici per rappresentare la conoscenza;

2. Un comportamento intelligente può essere generato **senza ragionamento esplicito** del tipo proposto dall'IA simbolica, ossia senza l'uso di linguaggi logici per ragionare con meccanisci di inferenza;

3. L'intelligenza è una **proprietà emergente** di certi sistemi complessi.

Brooks identifica due idee chiave:

1. **Collocazione e personificazione**: l'intelligenza reale è situata nel mondo, non in sistemi senza sostanza come i "theorem provers" o i "sistemi esperti";

2. **Intelligenza e apparenza**: il *comportamento intelligente* è generato come un risultato di una interazione con l'ambiente. L'intelligenza sta negli occhi di chi guarda, non è una proprietà innata e isolata.

### 15.2.1 Subsumption Architecture

Per illustrare le sue idee Brooks introduce la subsumption architecture che è una **gerarchia di task** che eseguono **behaviours**:

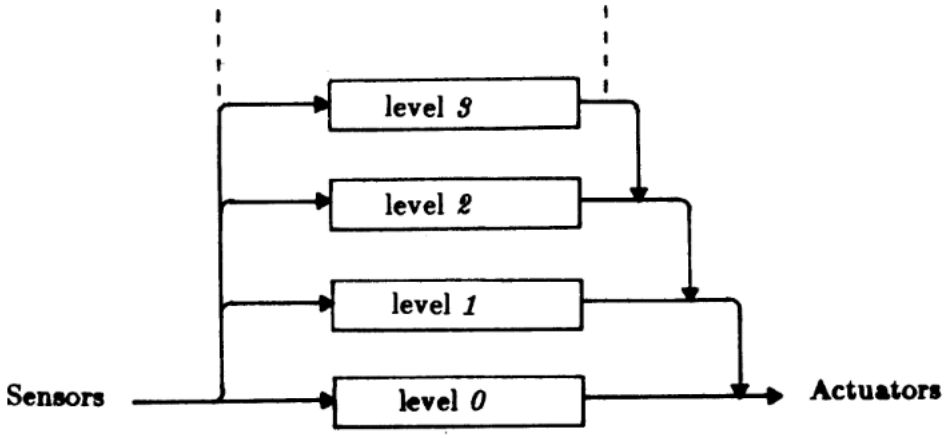



Figure 15.1: Gerarchia dei tasks

Ogni behaviour è una struttura a regole molto semplice che compete con altri per esercitare il controllo dell'agente. Gli strati più bassi rappresentano tipi di behaviour più primitivi che hanno la precedenza su strati più in alto nella gerarchia. I sistemi risultanti sono estremamente semplici in termini della quantità di computazione che fanno, ma alcuni robot eseguono task che sarebbero sconvolgenti se fossero eseguiti da sistemi simbolici di AI.

#### Selezione delle Azioni

Ogni livello è strutturato come una macchina a stati finiti per comprendere quale azione eseguire. La scelta di una azione è realizzata attraverso un insieme di behaviours. Un behaviour è una coppia $(c, a)$, dove $c$ è un insieme di percezioni dette condizioni e $a$ è una azione. Un behaviour $(c, a)$ scatta quando la funzione `see()` restituisce una percezione $p$, tale che $p \in c$. Associata a un insieme di behaviour rules c'è una relazione di inhibition $<$. Leggiamo $b_1 < b_2$ come "$b_1$ *inibisce* $b_2$", cioè $b_1$ è più basso nella gerarchia di $b_2$, e quindi ha la priorità su $b_2$.

### 15.2.2 Vantaggi degli agenti reattivi

I vantaggi di questo approccio sono:

- Semplicità;
- Economia;
- Trattabilità computazionale;
- Robustezza verso i fallimenti;

- Eleganza

### 15.2.3 Limiti degli agenti reattivi

- Dato che sono agenti senza modello dell'ambiente devono avere sufficienti informazioni disponibili localmente per decidere come comportarsi;
- L'agente ha una visione a breve termine dato che non può trattare informazione che non sia sull'ambiente locale;
- Difficile realizzare agenti reattivi che apprendono;
- Visto che i behaviour emergono da interazioni fra componenti più ambiente, è difficile vedere come ingegnerizzare agenti specifici (non esistono metodologie generali);
- È abbastanza complesso progettare agenti con molti comportamenti.

# Chapter 16

# Agenti Ibridi

## 16.1 Motivazione

Molti ricercatori hanno sostenuto che né un approccio completamente deliberativo né uno completamente reattivo sono adatti a costruire agenti. Per risolvere sia i problemi degli Agenti Reattivi 15 che quelli degli agenti deliberativi tradizionali si è proposto un approccio ibrido che cercasse di mettere insieme i punti forti di entrambi gli approcci.

## 16.2 Agenti Ibridi

Un approccio è quello di costruire un agente con due (o più) sottosistemi, che comprenda almeno i seguenti sottosistemi:

- **deliberativo**: contenente un modello simbolico del mondo, che sviluppa piani e prende decisioni nel modo proposto dall'IA simbolica;
- **reattivo**: capace di reagire ad eventi senza ragionamenti complessi.

Spesso ai componenti reattivi viene data precedenza rispetto a quelli deliberativi.
Questo tipo di struttura porta naturalmente all'idea di un'architettura a strati in cui i sottosistemi di controllo di un agente sono organizzati in una gerarchia, con livelli più alti che trattano l'informazione a crescenti livelli di astrazione. Un problema importante in queste architetture è la scelta del tipo di **schema di controllo** in cui inserire i sottosistemi dell'agente per trattare le interazioni fra i vari strati.
I due tipi di stratificazione possibili sono: stratificazione orizzontale vs verticale, come si può osservare in 16.1.

### 16.2.1 Stratificazione Orizzontale

Tutti gli strati sono direttamente connessi all'input sensoriale e all'output attuatore: ogni strato opera come un agente producendo suggerimenti su quale azione eseguire. Questa stratificazione prevede la definizione di un **controllore** che controlla ciò che entra in input e ciò che esce da ogni livello. E' lui che media l'interazione con l'ambiente. Ha come problema il fatto che il **sistema di controllo è un collo di bottiglia dove tutto deve passare**.

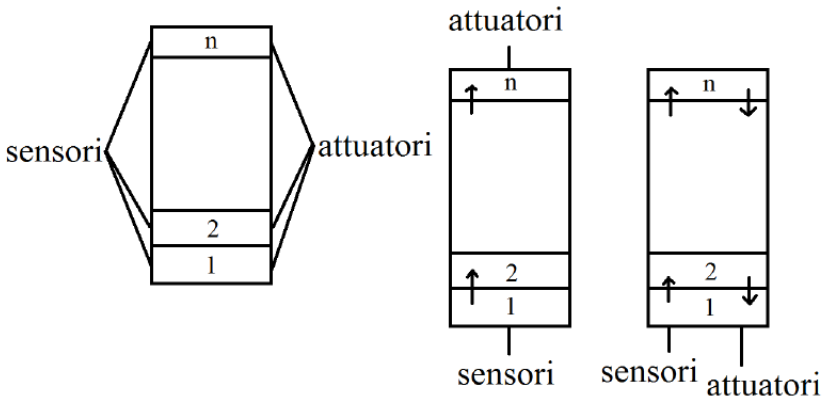



Figura 12: A sinistra un esempio di stratificazione orizzontale. Al centro e a destra due esempi di stratificazione verticale diversi

Figure 16.1: Tipi di Agenti Ibridi

**Esempio Orizzontale**

L'architettura di TOURINGMACHINES proposta da Ferguson consiste di sottosistemi di *percezione* e *azione* che si interfacciano direttamente con l'ambiente dell'agente e tre *control layers*, inseriti in un control framework, che media fra gli strati:

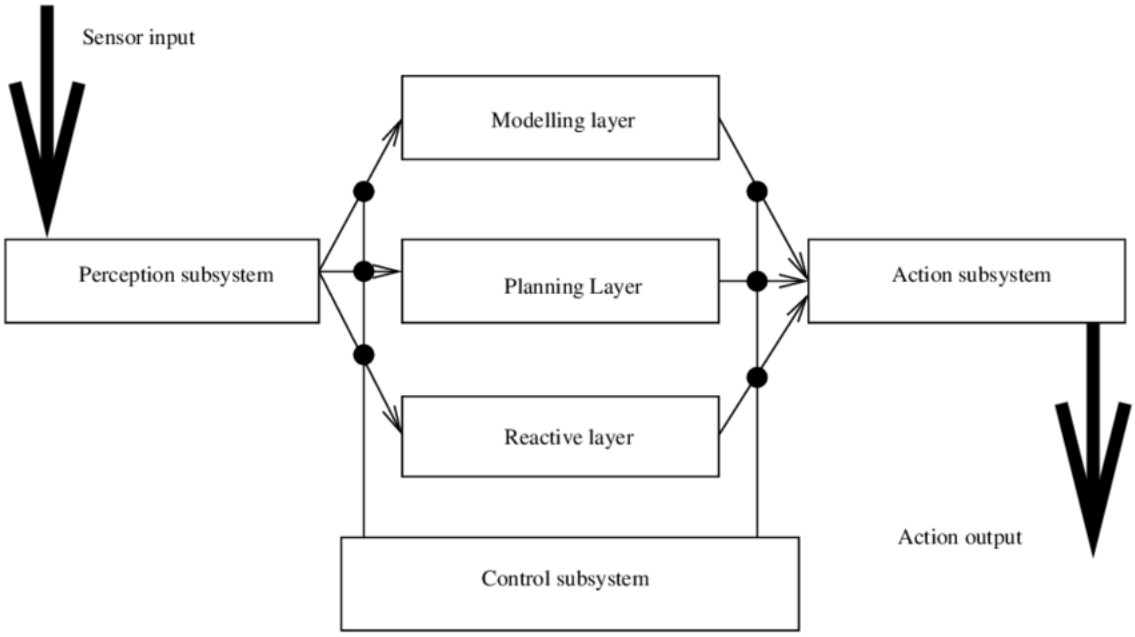



Figure 16.2: Architettura delle Touring Machines

Dove abbiamo:

- **Reactive Layer**: implementato come un insieme di regole situation-action, in maniera simile alla subsumption architecture;

- **Planning Layer**: costruisce piani e sceglie azioni da eseguire, in modo da raggiungere i goals dell'agente;
- **Modelling Layer**: contiene la rappresentazione simbolica del mondo e degli *stati cognitivi* di altre entità nell'ambiente dell'agente;
- **Control Subsystem**: permette ai precedenti 3 layer di comunicare tra di loro, inserendoli anche in un *control framework* che usa *control rules*.

### 16.2.2 Stratificazione Verticale

Input dei sensori e output delle azioni sono gestiti al massimo da uno strato ciascuno. Un solo livello è collegato ai sensori e un solo livello agli attuatori che possono o meno coincidere. Tra questi due livelli collochiamo dei livelli intermedi che contribuiscono a decidere quale azione intraprendere. Possiamo avere due tipi di stratificazione a seconda che l'informazione proceda ad una sola passata o a doppia passata. Ha come aspetto negativo che **non è fault tolerant rispetto ad un errore di un livello**.

#### Esempio Verticale

Come tipo di architettura ibrida con stratificazione verticale con doppia passata abbiamo l'InteRRaP proposto da Muller. Gli strati sono simili a quelli di TOURINGMACHINES:

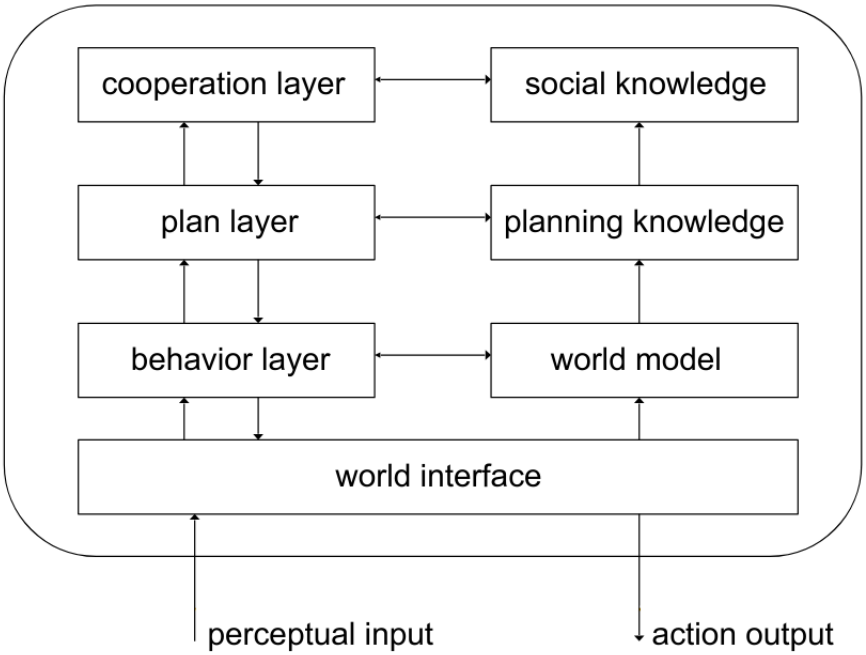



Figure 16.3: Architettura InteRRaP.

# Chapter 17

# Teoria Dei Giochi

## 17.1 Introduzione

Siamo in un ambiente multi-agente con agenti indipendenti ed **egoisti** nel senso che ognuno persegue i propri obiettivi individuali e ci chiediamo come possa un agente prendere un'**azione razionale** in questo contesto. Nei giochi normalmente studiati nei corsi di IA i giocatori giocano a turno con informazione perfetta, usando la ricerca minimax per trovare le mosse migliori. Qua vediamo invece solo **giochi simultanei** in cui ogni giocatore fa la mossa allo stesso istante e in particolare la loro versione più semplice possibile in cui ci sono solo 2 giocatori e che consistono di una sola mossa prima della terminazione. La teoria dei giochi nel contesto degli agenti può essere usata in almeno due modi:

- **Progettazione di agenti**: si può determinare le azioni che l'agente deve eseguire in base alla loro utilità attesa all'interno del gioco in modo che abbia un comportamento razionale;
- **Progettazione di meccanismi**: dato un certo ambiente con più agenti, andiamo a determinare quale sia il gioco che devono giocare gli agenti per massimizzare il bene comune. Ad esempio la teoria dei giochi può aiutarci a progettare un protocollo per una collezione di router su Internet in modo che ciascuno di essi sia incentivato ad agire per ottimizzare l'uso globale dalla rete. (Progettazione di Meccanismi con Teoria dei Giochi 18)

## 17.2 Teoria dei Giochi

Definiamo dei concetti utili nella teoria dei giochi:

- **Giocatore**: il giocatore o agente sono le entità che prendono le decisioni nel gioco;
- **Strategia o Azione**: è la scelta che un giocatore fa all'interno di tutto il set delle possibili azioni. Ovviamente giocatori diversi potrebbero avere o meno lo stesso set di azioni possibili;

- **Strategia pura**: ogni giocatore deve adottare e poi eseguire una strategia pura, cioè una politica deterministica che specifica una particolare azione da intraprendere in ogni situazione;
- **Strategy profile**: specifica quale strategia pura è adottata da ogni giocatore e quindi definisce deterministicamente il **risultato** del gioco;
- **Utility functions**: possiamo descrivere le **preferenze** per ogni agente con le utility functions, una per ogni agente, che assegna ad ogni risultato e quindi ad ogni strategy profile, un numero che indica quanto è '*buono*' il risultato per quell'agente;
- **Matrice dei Payoff**: mette insieme tutta l'informazione precedente. La matrice si costruisce prendendo due giocatori e mettendo sulle righe le strategie pure che il primo giocatore può fare e sulle colonne le strategie pure del secondo giocatore. Il numero di righe e colonne dipende quindi dal numero di mosse che possono fare rispettivamente. In ogni entrata ci sarà uno *strategy profile* con associata l'utilità per entrambi i giocatori di terminare con quel risultato;
- **Soluzione**: una soluzione è un sottoinsieme delle entrate della tabella in cui ogni giocatore adotta una **strategia razionale**.

L'argomento più importante in teoria dei giochi è definire **cosa significa razionale** quando ogni agente sceglie solo una parte dello strategy profile che determina il risultato. Prima di presentare come esempio il *Dilemma del Prigioniero*, uno scenario immaginario, inventato dal Professor Albert W. Tucker di Princeton, che è senza dubbio una delle più famose rappresentazioni della Game Theory, diamo altre definizioni fondamentali:

- **Domina Debolmente**: diciamo che una strategia $s$ per il giocatore $p$ domina debolmente la strategia $s'$ se il risultato di $s$ è migliore o uguale (in termini di payoff) per $p$ di quello di $s'$ per ogni scelta di strategie da parte degli altri giocatori ed almeno in un caso $s$ è strettamente migliore di $s'$;
- **Domina Fortemente**: diciamo che una strategia $s$ per il giocatore $p$ domina fortemente la strategia $s'$ se il risultato di $s$ per $p$ è sempre meglio di quello di $s'$ per ogni scelta di strategie da parte degli altri giocatori;
- **Strategia Dominante**: per un utente $p$ è la strategia $s^\star$ che domina tutte le altre. E' irrazionale per un agente scegliere una strategia dominata ed è irrazionale non scegliere una strategia dominante quando ne esiste una. A seconda che la dominazione sia forte o debole si parlerà di **strategia fortemente dominante** e **strategia debolmente dominante**;
- **Debolmente Pareto Superiore**: dati due possibili risultati del gioco $w_1$ e $w_2$ possiamo dire che "$w_1$ è *debolmente Pareto superiore di* $w_2$" se ogni giocatore considera $w_1$ migliore o uguale in termini di utilità di $w_2$ e almeno un giocatore preferisce $w_1$;
- **Strettamente Pareto Superiore**: dati due possibili risultati del gioco $w_1$ e $w_2$ possiamo dire che "$w_1$ è *strettamente Pareto superiore di* $w_2$" se ogni giocatore considera $w_1$ migliore di $w_2$;

- **Pareto Optimal**: un risultato è Pareto ottimo se non c'è un altro risultato o strategy profile preferibile da tutti i giocatori, ossia non è possibile migliorare l'utilità di un soggetto, senza peggiorare il benessere degli altri soggetti;

- **Equilibrio di Nash**: uno strategy profile $S = (P_i : s_i)$ è un equilibrio di Nash se ogni giocatore $P_i$ in $S$ farebbe peggio se deviasse da $S$, assumendo che tutti gli altri giocatori si attengano a $S$. E' ampiamente riconosciuto dagli studiosi della teoria dei giochi che l'equilibrio è una condizione necessaria per qualunque soluzione ad un gioco. Si può dimostrare che un equilibrio ottenuto con strategie dominanti è un equilibrio di Nash, ma non tutti i giochi hanno strategie dominanti. Saremo comunque interessati ad individuare tali equilibri perchè sono quelli dove termina il gioco.

### 17.2.1 Dilemma del Prigioniero

Questa è la sua formulazione:

> *Due presunti scassinatori, Alice e Bob, sono catturati con le mani nel sacco e interrogati separatamente dalla polizia. Entrambi sanno che, se confessano il crimine, verranno condannati a 5 anni di prigione per furto con scasso, ma se entrambi rifiutano di confessare potranno essere condannati solo a 1 anno per il crimine minore di possesso di oggetti rubati. Tuttavia, la polizia offre a ognuno di loro separatamente la possibilità di testimoniare contro il complice: in tal caso il testimone se ne andrebbe libero, mentre l'altro sarebbe condannato a 10 anni. Ora Alice e Bob sono posti di fronte al cosiddetto dilemma dei prigionieri: devono testimoniare o no? Essendo agenti razionali, Alice e Bob vogliono massimizzare la propria utilità. Supponiamo che Alice sia totalmente disinteressata del destino del complice: in questo caso la sua utilità decresce con il numero di anni che ella dovrà passare in prigione, ma quel che accade a Bob è del tutto ininfluente. Bob contraccambia tale sentimento.*

Per arrivare a una decisione razionale, entrambi si basano sulla seguente matrice di payoff:

| | Alice: testify | Alice: refuse |
|---|---|---|
| **Bob: testify** | A = -5, B = -5 | A = -10, B = 0 |
| **Bob: refuse** | A = 0, B = -10 | A = -1, B = -1 |

Table 17.1: Matrice dei Payoff nel Dilemma del Prigioniero

Sia Alice che Bob hanno una strategia dominante, quella di testimoniare. Presa ad esempio Alice, se fosse razionale, analizzerebbe la matrice con il seguente ragionamento:

- se Bob testimoniasse mi prenderei: 5 anni se testimoniassi anche io e 10 in caso contrario. Quindi in questo caso è meglio testimoniare;

- se Bob invece si rifiutasse di testimoniare mi prenderei: 0 anni nel caso in cui decidessi di testimoniare e 1 anno se mi rifiutassi. Anche in questo caso testimoniare è meglio.

In definitiva, in entrambi i casi è meglio testimoniare, per cui lo farà. Se Alice è furba, oltre che razionale, potrà continuare a ragionare come segue: anche per Bob la strategia dominante è testimoniare. Quindi, lui testimonierà ed entrambi saremo condannati a 5 anni. Il dilemma è che il risultato è peggiore, -5 per ciascun giocatore, del risultato che entrambi otterrebbero se si rifiutassero di testimoniare, cioè -1 per entrambi. La strategia $(A : refuse, B : refuse)$ è pareto ottima, mentre $(A : tesify, B : testify)$ non lo è.

*C'è un modo per Alice e Bob di arrivare al risultato ottimale?*

Certamente è possibile che entrambi si rifiutino di testimoniare, ma è molto improbabile. Ciascuno avrebbe la tentazione di testimoniare, perché guadagnerebbe 1 anno di prigione mentre l'altro si troverebbe con 10 anni. La ragione per cui la soluzione Pareto optimal è improbabile, è che non è un punto di equilibrio di Nash, mentre lo è lo scenario in cui entrambi testimoniano. Cambia il risultato se cambia la funzione di utilità di Alice e Bob. Se i due ladri fossero ad esempio parenti o amici stretti potrebbero considerare come utilità la somma degli anni di prigione di entrambi; in questo caso la soluzione pareto ottima diventa anche equilibrio di Nash.

### 17.2.2 Più punti di equilibrio e Strategia Mista

Ci sono scenari di giochi in cui non ci sono strategie dominanti, ma ci sono comunque più di un punto di equilibrio di Nash come nel seguente problema:

> Acme, un costruttore di hardware per videogiochi, deve decidere se il suo prossimo gioco userà DVD o CD. Intanto Best, produttore di software per giochi, deve decidere se produrre il suo prossimo gioco su DVD o CD. I profitti saranno positivi se entrambi sono d'accordo e negativi se non lo sono, come mostrato nella matrice seguente:

| | Acme: DVD | Acme: CD |
|---|---|---|
| **Best: DVD** | A = 9, B = 9 | A = -4, B = -1 |
| **Best: CD** | A = -3, B = -1 | A = 5, B = 5 |

Table 17.2: Matrice Dei Payoff ACME

In tal caso, come per ogni gioco, a noi interessa trovare una sola soluzione razionale per ogni giocatore. A questo punto gli agenti hanno un problema: nessuno di loro ha una strategia dominante e ci sono due diverse soluzioni accettabili, ma se ogni agente sceglie una soluzione differente, allora lo strategy profile risultante diventa pessimo.

*Quale punto di equilibrio scegliamo tra quelli possibili?*

- Una possibilità potrebbe essere quella di scegliere la soluzione migliore, quella ottima di Pareto. In questo caso **DVD, DVD**.

*Ma se questa non esistesse? Se tutti i punti di equilibrio avessero stesso valore di utilità?*

- Generalmente per poter trattare una situazione di questo genere gli agenti devono **comunicare**, o stabilendo una convenzione che ordina le soluzioni prima dell'inizio del gioco, o **negoziando** per raggiungere durante il gioco un accordo che soddisfi entrambi.

*E se abbiamo un gioco privo di punti di equilibrio classico con strategie pure?*

- Questo è il caso dei **giochi a somma zero**, ossia giochi in cui la somma dei payoff in ogni cella della matrice dei payoff è zero e la cui soluzione risiede nella definizione di **strategie miste**. A differenza della strategia pura deterministica che abbiamo definito prima, questa è non deterministica, in particolare se un giocatore ha $k$ scelte possibili $s_1, s_2, ..., s_k$ una strategia mista su queste scelte assegna una distribuzione di probabilità $p_1, p_2, ..., p_k$ la cui somma è 1 e ad ogni passo campiona la mossa da eseguire. Nash dimostrò che ogni gioco in cui ogni giocatore ha un insieme finito di strategie possibili ha un equilibrio con strategie miste. Nonostante questo trovare tale equilibrio può essere molto complesso, così come definire un algoritmo in grado di calcolarlo. Esistono solo alcuni risultati per particolari classi di giochi.

### 17.2.3 Ripetizione del Gioco

*Cosa succederebbe se un agente venisse a conoscenza di dover incontrare più volte un altro agente?* Dobbiamo tener in conto che l'altro agente possa considerare come si sia comportato il nostro agente precedentemente. Questa situazione è stata studiata matematicamente e ne è emerso che:

- **Si Coopera** se un agente ne incontra un altro $\infty$ volte allora è sempre meglio cooperare, infatti persistere in un comportamento di non cooperazione comporta una diminuzione del guadagno di tutti;
- **Non Si Coopera** se un agente ne incontra un altro $n$ volte con $n$ finito e conosciuto da entrambi gli agenti. Questo succede perché con $n$ finito ogni agente potrebbe essere tentato di non cooperare sull'ultima interazione con l'altro agente per massimizzare i propri guadagni. Però, se all'ultima interazione non conviene cooperare, allora non conviene neanche alla penultima e così via fino alla prima. Questo è il problema della **backward induction**.

Ad ogni modo nel 1984 Axelrod ha dato il via ad un torneo per stabilire quale fosse la strategia migliore relativa al dilemma del prigioniero iterato. Dopo aver sviluppato diverse strategie, le ha fatte interagire tra di loro in "match" differenti. Ogni match era costituito da 200 round. Ne è risultato che la strategia migliore, ossia quella che si è comportata complessivamente meglio contro tutte le altre, è la *tit-for-tat* in cui l'agente al round 0 coopera con l'altro, ma in ogni altro round l'agente sceglie la soluzione scelta dall'altro agente nel round precedente. Questo permette di reagire abbastanza prontamente contro le strategie che non cooperano, ma permette di avere un grande guadagno con tutte quelle strategie che invece tendono a cooperare. Ovviamente questa strategia perdeva contro quella che non coopera mai come affermato precedentemente, ma complessivamente risultava la migliore.

# Chapter 18

# Progettazione di Meccanismi - Teoria dei Giochi

## 18.1 Introduzione

Se in un gioco tipico della Teoria dei Giochi 17 non si riesce a trovare una soluzione con 1 solo punto di equilibrio di Nash, occorre avviare la negoziazione tra agenti. Con progettazione di meccanismi s'intende la progettazione di **protocolli** (sequenze di azioni) che gli agenti possono eseguire per governare un'interazione multiagente.

## 18.2 Meccanismi

Un buon protocollo di interazione dovrebbe avere le seguenti proprietà:

- **Successo garantito**: il protocollo garantisce che prima o poi verrà raggiunto un accordo;
- **Massimizzare il welfare sociale**: un accordo che massimizza la somma delle utilità dei partecipanti;
- **Pareto efficiency**: il risultato della negoziazione deve essere Pareto optimal, ossia non esiste un altro risultato che permette a qualcuno di migliorare la propria situazione senza che quella di qualcun altro peggiori;
- **Individual rationality**: i partecipanti hanno interesse nel seguire le regole del protocollo perchè per loro è una strategia razionale;
- **Stability**: il protocollo dà un incentivo a comportarsi in un certo modo come ad esempio quello di giungere in un punto di equilibrio di Nash;
- **Simplicity**: la strategia del protocollo è "ovvia".

Formalmente un meccanismo consiste di:

1. **Linguaggio** per la descrizione delle possibili strategie adottabili dagli agenti;

2. **Regola** per i risultati che determina le vincite degli agenti dato un profilo di strategie adottabili.

A prima vista, il problema di progettare un meccanismo potrebbe sembrare banale: se ogni agente massimizza la propria utilità, questo porterà automaticamente alla massimizzazione dell'utilità globale. Sfortunatamente questa soluzione non funziona, perché le azioni di ogni agente possono influenzare il benessere degli altri in modo tale da ridurre l'utilità globale. Un esempio è la **tragedy of commons**, una situazione in cui tutti i contadini portano le loro bestie a brucare gratis sui prati comunali, con il risultato di distruggerli e giungere così ad un'utilità negativa per tutti. Ogni contadino ha agito singolarmente in modo razionale, ragionando che l'uso del prato comune era libero. Un approccio standard per gestire simili problemi è "*far pagare*" ad ogni agente l'uso delle risorse collettive.

### 18.2.1 Meccanismi di Asta

Un meccanismo importante è quello delle aste. Assumiamo che ci sia un singolo bene in vendita, che ha un **valore pubblico** per tutti i partecipanti e un **valore privato** che può essere diverso da un partecipante ad un altro. I tipi di aste possono essere:

- **asta inglese**: il banditore incrementa via via il prezzo dell'articolo a partire da un lower bound, controllando che ci siano ancora acquirenti interessati, finché non ne rimane solo uno. L'asta inglese ha anche la caratteristica che i partecipanti hanno una semplice strategia dominante: continuare a puntare finché il costo corrente è inferiore al valore personale privato;
- **asta olandese**: il banditore parte offrendo il bene a un prezzo molto alto (upper bound) e continua abbassando il prezzo finché un qualche partecipante accetta l'offerta del banditore. Ha la stessa strategia dominante del precedente, accettare l'offerta quando il costo è il valore personale.

Una proprietà indesiderabile di queste due tipi di aste è il loro alto costo di comunicazione: i partecipanti devono essere tutti nella stessa stanza o devono disporre di linee sicure ad alta velocità. Si tratta infatti di un gioco cooperativo. Nascono quindi altri tipi di asta non cooperative:

- **asta in busta chiusa**: ogni partecipante comunica al venditore segretamente una singola offerta, e quella più alta vince. Con questo meccanismo, la strategia di offrire il valore reale non è più dominante;
- **asta Vickrey**: detta anche asta in busta chiusa al secondo prezzo, in cui come nel caso precedente vince chi fa l'offerta più alta, ma paga un prezzo corrispondente alla seconda offerta più alta e non alla sua propria. In questo caso si può verificare che esiste una strategia dominante molto semplice: offrire il proprio valore privato. In generale, se il tuo valore privato è $v_i$ e hai offerto $b_i$, è semplice verificare che non si ottiene un risultato migliore se $b_i \neq v_i$ analizzando i due casi possibili:

  1. $b_i > v_i$: puoi risultare primo, ma non hai garanzia e rischi solo di pagare una cifra maggiore di $v_i$;

2. $v_i < b_i$: hai meno probabilità di vincere rispetto a offrire $v_i$ e, se vinci, la cifra da pagare sarà sempre quella che pagheresti offrendo $v_i$ , ossia quella proposta dal secondo vincitore. Conviene quindi puntare $v_i$ ed aumentare le possibilità di vincita.