Agenti Intelligenti

Lorenzo D'Isidoro
lorenzo.disidoro@gmail.com

Intelligenza Artificiale e Sistemi Informatici A.A. 2022/2023

Contents

1 Introduzione

Il corso ha l'obiettivo di introdurre gli aspetti principali dei sistemi multiagente, ossia sistemi composti di elementi computazionali che interagiscono, noti come agenti. Gli agenti sono sistemi computazionali capaci di eseguire azioni in modo autonomo, e di interagire con altri agenti svolgendo attività sociali come cooperazione, coordinamento, negoziazione. I sistemi multi agente costituiscono una metafora naturale per modellare un ampio spettro di "artificial social systems". Nella prima parte del corso vengono forniti gli strumenti metodologici per comprendere i sistemi multiagente discutendo le architetture di singoli agenti e le principali problematiche legate all'interazione fra agenti. La seconda parte del corso presenta alcuni linguaggi e ambienti specifici per sistemi multiagente, in modo da consentire agli studenti di implementare alcuni esempi significativi. Pagina del corso.

2 Introduzione agli agenti intelligenti

Parte di questo documento fa riferimento ai lucidi del corso che sono parzialmente basati sul libro di testo del corso "M. Wooldridge, An Introduction to Multi-Agent Systems, Wiley, 2009" e sul materiale fornito da Alberto Martelli dell'Università degli Studi di Torino.

2.1 La Computazione

Prima di introdurre gli agenti intelligenti sarà utile discutere dei cinque trend che hanno caratterizzato la storia della computazione, ovvero:

- Ubiquità: La continua riduzione dei costi di elaborazione ha permesso la diffusione di sempre più dispositivi.
- Interconnessione: I sistemi informatici oggi non esistono più da soli, ma sono collegati in rete in grandi sistemi distribuiti, quindi sono sempre più interconnessi (e.g. LAN, Internet, ...)
- Intelligenza e Delega: I computer fanno di più per noi, senza un nostro intervento (e.g. sistemi a guida autonoma, ...)
- Human Oriented: Spostarsi da una vista orientata macchina della programmazione verso concetti e metafore che sono più vicine al modo con cui noi comprendiamo e vediamo il mondo, quindi l'utilizzo di astrazioni più familiari per l'uomo (e.g. Object Oriented Programming per gli sviluppatori)

Delegazione e intelligenza implicano la necessità di **costruire sistemi informatici in grado di agire efficacemente per nostro conto**.

- La capacità di agire dei sistemi informatici in modo **indipendente**
- La capacità dei sistemi informatici di agire in un modo che **rappresentino nel migliore dei modi i nostri interessi** durante l'interazione con altri esseri umani o sistemi
  - Interconnessione e distribuzione, accoppiate con la necessità di sistemi che rappresentino nel migliore dei modi i nostri interessi, portano a sistemi che possono cooperare e raggiungere accordi o competere.

Tutte queste tendenze hanno portato alla nascita di un nuovo campo in informatica: **Sistemi multi-agente**.

2.2 Gli Agenti

Un **agente** è un sistema di computazione capace di agire in modo **indipendente** per conto di un utente o proprietario, capendo cosa deve essere fatto per

soddisfare gli **obiettivi** di progettazione, piuttosto che essere costantemente informato.

Un **sistema multi agente (MAS)** consiste in una serie di agenti, che interagiscono l'uno con l'altro. Per interagire con successo, richiedono la capacità di **cooperare**, **coordinarsi** e **negoziare** tra loro, in modo analogo a quanto fanno le persone. Un MAS quindi ha il compito di rispondere a queste domande:

- Come possiamo creare agenti capaci di agire in modo indipendente, autonomo, in modo che possano svolgere con successo i compiti che gli deleghiamo?
- Come possiamo creare agenti capaci di interagire (cooperare, coordinare, negoziare) con altri agenti con il fine di portare a termine con successo i compiti delegati, soprattutto quando non si può presumere che gli altri agenti condividano gli stessi interessi/obiettivi?

Il primo problema è di progettazione/design di un agente, quindi micro, il secondo 'e di progettazione/design di una società di agenti, quindi macro.

2.3 Esempio: Curiosity

Curiosity è un rover robotico delle dimensioni di un'auto che esplora il cratere Gale su Marte come parte della missione Mars Science Laboratory (MSL) della NASA. Curiosity è stato lanciato da Cape Canaveral il 26 novembre 2011.

Gli obiettivi del rover includono: indagini sul clima marziano e geologia; valutazione del fatto che il sito selezionato offra condizioni ambientali favorevoli per vita microbica, inclusa un'indagine sul ruolo dell'acqua; etc.

Curiosity ha operato in modalità autonoma per circa 25 giorni.

Per far ciò si è deciso di basare la sonda su **algoritmi di pianificazione** e non solo su sistemi di controllo ingegneristici.

2.4 Caratteristiche degli Agenti

In ultima istanza possiamo vedere un agente come un **sistema di computazione** capace di agire **autonomamente**, in maniera indipendente e in modo **flessibile**, in un qualche **ambiente** con il fine di raggiungere gli **obiettivi** per cui è stato progettato.

Un esempio di agente intelligente triviale, quindi banale, non interessante, è il termostato. Infatti il termostato decide cosa fare, ovvero accendere, spegnere il riscaldamento o non fare nulla, sulla base della temperatura rilevata dall'ambiente.

2.5 Flessibilità

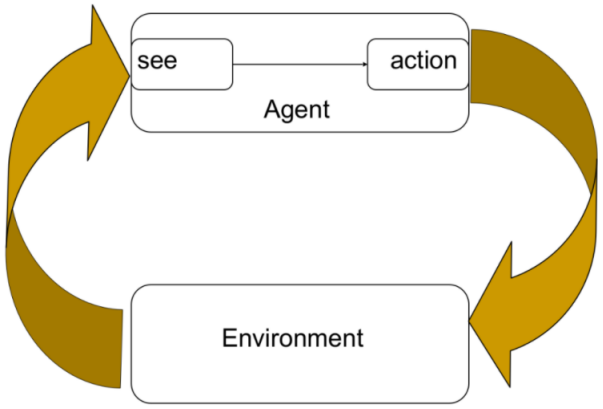
Come anticipato nel precedente paragrafo un agente intelligente è un sistema di computazione capace di esibire azioni autonome in modo *flessibile* in un ambiente. Con flessibile si intende:

- Reattivo
- Proattivo
- Sociale

2.5.1 Agente Reattivo

Un agente reattivo è un sistema che mantiene una costante interazione con l'ambiente e risponde ai cambiamenti che occorrono su di esso (in tempo perché la risposta sia utile). Bisogna tenere a mente però che l'ambiente reale è dinamico e le informazioni a volte possono essere incomplete.

Un agente reattivo può essere implementato tramite semplici regole condizionali del tipo: *"if car-in-front-is-braking then initiate-braking"*, se l'agente percepisce che l'auto di fronte sta frenando, allora inizia immediatamente a frenare per evitare una possibile collisione, senza perdere tempo a ragionare, in sintesi quello che viene fatto è, per dato evento/stimolo → una regola di risposta.



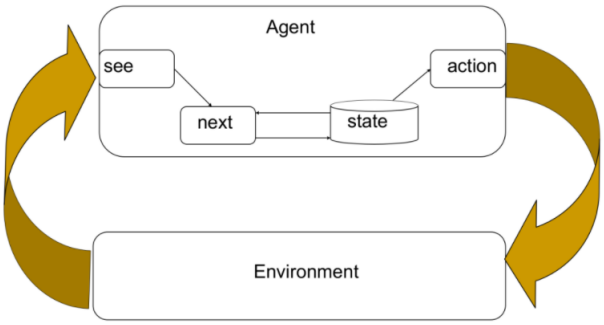
2.5.2 Agente Proattivo

Generalmente si vuole che gli agenti siano in grado di agire **autonomamente**, quindi generare e tentare di raggiungere gli obiettivi, non solo guidati dagli eventi, ma che siano in grado di prendere l'iniziativa.

Per agire autonomamente l'agente ha bisogno di informazioni:

- su come l'ambiente evolve
- su come le proprie azioni impattano sull'ambiente
- sull'obiettivo in modo tale che sia in grado di agire per raggiungerlo

Nel tentare di raggiungere un obiettivo l'agente deve essere in grado di ragionare su **piani**.



Si desidera che gli agenti siano reattivi e che rispondano ai cambiamenti per tempo e che lavorino in modo sistematico verso obiettivi di "lungo termine", ma *progettare un agente che bilanci reattività e proattività è ancora un problema di ricerca aperto.*

2.5.3 Agente Sociale

Per abilità sociale di un agente intelligente si intende l'abilità di interagire con altri agenti (anche umani) attraverso un qualche tipo di linguaggio di comunicazione (agent-communication language, ACL) e cooperare con essi.

2.5.4 Conclusioni

Un **programma non è un agente** perché il suo output non ha effetto su quello che lui percepisce in seguito e inoltre non c'è continuità temporale.

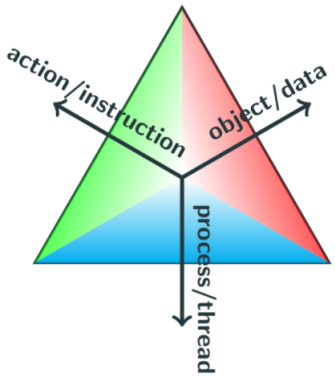
3 Il Triangolo della Computazione

L'ingegneria del Software aspira a un software di qualità, quindi un software che cerca di raggiungere i seguenti goal:

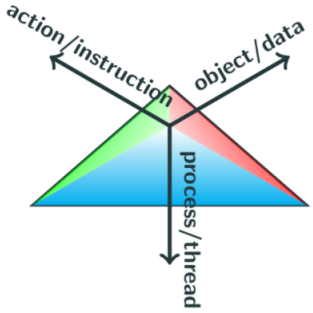
- Correttezza (correctness)
- Robustezza (robustness)
- Estensibilità (extensibility)
- Riusabilità (reusability)

Un ingrediente cruciale per raggiungere questi obiettivi è una corretta **modularizzazione** dell'architettura del software. Il triangolo di Meyer fornisce un terreno comune per confrontare diversi approcci alla modularizzazione, dove le "forze" in gioco sono:

- Process o Thread: CPU fisica, un processo o un thread.
- Action o Istruction: operazioni che compongono il calcolo (linguaggio macchina, operazioni fino alle subroutine).
- Object: strutture dati a cui le azioni si applicano.



3.1 Decomposizione su Process o Thread



Pro:

- Semplice ed intuitivo: decomposizione un programma in funzioni, sotto-funzioni e procedure.
- Algorithmic oriented: appropriato quando viene specificato un unico obiettivo principale.

Contro:

- Difficilmente manutenibile: difficile incorporare nuovi "obiettivi principali", quindi allo **sviluppo incrementale**.
- Difficilmente scalabile in presenza di dati condivisi e processi concorrenti.

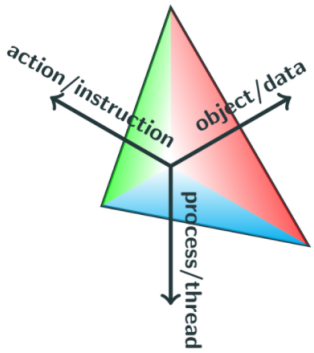
3.1.1 Business Processes

Si tratta di un modello che crea una rappresentazione esplicita delle attività di una società/azienda e descrivere come un insieme di attività interconnesse porta a un risultato preciso e misurabile in risposta a un evento esterno. Può essere definito come una sorta di decomposizione funzionale dove il goal si raggiunge a seguito dell'esecuzioine di un processo, diviso in sotto-processi.

Un contro è quello dovuto al fatto che l'enfasi è sul processo (vista incentrata sull'attività):

- Se il dato deve influenzare il processo non funziona.
- Nessuna astrazione adeguata per acquisire dati manipolati lungo i flussi.

3.2 Decomposizione su Oggetti e Dati



Pro:

- Si adatta bene allo sviluppo incrementale, quando i requisiti non sono noti a priori.
- Gli oggetti hanno una vita propria, indipendente dei processi li utilizza.

- Operazioni sui dati: fornire le azioni di su cui è possibile operare su di essi.

Contro:

- Gli oggetti sono passivi: i processi esterni prendono le decisioni su quali azioni invocare oggetti.
- Nessun disaccoppiamento tra l'uso di un oggetto e la gestione di tale oggetto.
- Per la gestione dei thread si deve ovviare associando ad ogni thread un oggetto.

3.2.1 Actors e Active Object

Data la complessità del tentativo di coordinare gli accessi a un oggetto da più thread, a volte ha più senso evitare del tutto il multi threading e si può utilizzare il pattern degli **Actors e Active Object**.

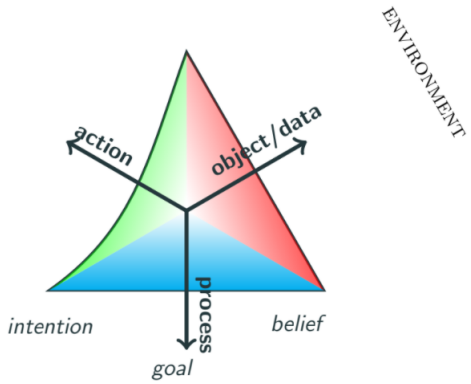
Active Object in generale è un pattern per gestire la concorrenza in cui si tenta di separare l'invocazione di un metodo dalla sua esecuzione.

Gli attori sono a un livello di astrazione più elevato rispetto a threads, infatti un **Actor** è un semplice oggetto che riceve dei messaggi e reagisce ad essi con un azione esegua sul thread corrente, quindi ad un Actor è associato il suo thread. E' possibile che l'attore decida di eseguire l'attività di azione su un pool di thread.

Uno dei problemi del modello ad attori è quello che non affronta adeguatamente il problema del coordinamento (sebbene sono state proposte estensioni). Inoltre non supporta la progettazione e la modularizzazione dei processi che utilizzano questi oggetti, perché processi esterni agli Attori.

3.3 Il Paradigma Agent-Oriented

La programmazione orientata agli agenti (AOP) è un paradigma di programmazione in cui la costruzione del software è centrata sul concetto di agenti software. A differenza della programmazione orientata agli oggetti che ha gli oggetti, che forniscono metodi con parametri variabili, un agente ha uno stato che è costituito da componenti come **credenze (belief)**, **intenzioni (intention)**, capacità e **obblighi**; per questo motivo lo stato di un agente è chiamato stato mentale.



Il paradigma dell'agente vede se stesso, e l'ambiente circostante, come due astrazioni di prima classe:

- **Agenti sono orientati al processo**:
  - Il ciclo deliberativo, delle intenzioni, è finalizzato al raggiungimento di un goal (processo).
  - Sono in grado di percepire e manipolare il loro ambiente.
- **Ambiente è orientato ai dati/oggetti**: I dati, gli oggetti, sono le componenti che descrivono l'ambiente circostante.

Possiamo intuire facilmente che, a cavallo tra le *azioni* e il *processo* si collocano le **intenzioni (intention)**. Mentre tra il *processo* e i *dati* troviamo le **credenze (belief)**. Quindi per evidenziare la *differenza tra Agenti e Oggetti* sottolineiamo che gli Oggetti

- Non hanno il controllo sul proprio comportamento (passivo/reattivo)
- Non mostrano un comportamento flessibile (nessun ciclo deliberativo)
- Sono a thread singolo

La principale *differenza tra Agenti e Attori*: Gli attori non sono intenzionali/proattivi.

4 Agent Architectures

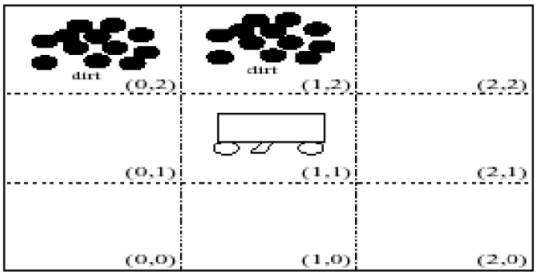
Il classico approccio per costruire agenti intelligenti è quello di vederli come casi particolare di sistemi basati sulla conoscenza, questo paradigma è noto come "symbolic AI". L'architettura di un agente deliberativo, infatti, contiene una esplicita rappresentazione (modello simbolico) dell'ambiente e prende decisioni (ad esempio, quale azione eseguire) attraverso un ragionamento simbolico.

Sono due i problemi da affrontare:

- Il problema della **trasduzione**: ovvero il problema della traduzione del mondo reale, dell'ambiente, in una descrizione simbolica accurata, in tempo perché sia utile (si pensi ai problemi di visione, riconoscimento del parlato, apprendimento, ...).
- 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.

4.1 Agenti con ragionamento deduttivo

Un esempio è quello del robot delle pulizie, il cui obiettivo è quello di pulire tutto lo sporco presente in un area della casa. Il primo passaggio è sicuramente quello di codificare l'ambiente, in questo caso rappresentiamo la stanza come una matrice.



Ma come un agente può decidere cosa fare utilizzando tecniche di ragionamento?

L'idea di base 'e usare la logica per codificare una teoria permettendo di ottenere la migliore azione da eseguire in qualsiasi situazione. Definiamo 3 predicati per risolvere il problema:

- In(x, y), l'agente è nella posizione (x, y).
- Dirt(x, y), c'è dello sporco nella posizione (x, y).
- Facing(d), l'agente è rivolto in direzione d.

Possibili azioni: Ac = turn, forward,suck.

Definiamo le regole per determinare cosa fare:

$$In(0,0) \land Facing(north) \land \neg Dirt(0,0) \Rightarrow Do(forward)$$

$$In(0,1) \land Facing(north) \land \neg Dirt(0,1) \Rightarrow Do(forward)$$

...

Possibili **problemi** possono essere come scoprire l'ingresso della videocamera in Dirt(0, 1) e che il processo decisionale basato sulla logica del primo ordine è indecidibile.

5 Agenti Razionali

Quando si spiega l'attività umana, è spesso utile fare dichiarazioni come le seguenti: *"Janine ha preso l'ombrello perché lei crede che possa piovere"*. Queste affermazioni fanno uso di una **psicologia popolare** (folks psychology) che fa riferimento al complesso di teorie, credenze e abilità cognitive e comprende la capacità di intuire il funzionamento della mente e di predirne il comportamento.

L'azione *"Janine ha preso l'ombrello"* a seguito della credenza/intuizione *"lei crede che possa piovere"*.

5.1 Sistema Intenzionale

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"*. Quindi secondo questa definizione è opportuno chiedersi se è legittimo o utile attribuire credenze e desideri ai sistemi informatici.

L'accademico Jhon McCarthy afferma che ci sono occasioni in cui ascrivere una macchina come sistema intenzionale è appropriata, ad esempio, quando l'iscrizione ci aiuta a capire la struttura della macchina, il suo comportamento passato o futuro, o come ripararla o migliorarla.

Dobbiamo tenere a mente però che 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 spesso diventano impraticabili. **Le nozioni intenzionali sono quindi astrazioni**, che forniscono un modo semplice e familiare per descrivere, spiegare e prevedere il comportamento di sistemi complessi. Quindi qualsiasi sistema, più o meno complesso, può essere definito come sistema intenzionale (e.g. interruttore della luce, Amazon Alexa, robot aspirapolvere, ...) le quali nozioni intenzionali sono tendiamo a vederle come astrazioni, non ci preoccupiamo del loro funzionamento nel dettaglio.

Ma anche importanti sviluppi nella computazione sono basati su astrazioni, basti pensare ai linguaggi di programmazione che ci astraggono dal linguaggio macchina, e anche i concetti di:

- Oggetti
- Procedure/funzioni
- Tipi di dati

Gli agenti intelligenti e, in particolare, i sistemi intenzionali, rappresentano un ulteriore (e potente) astrazione.

5.2 Practical Reasoning (Ragionamento su Azioni)

Il practical reasoning è il ragionamento sulle azioni, sul processo di capire cosa fare, l'uso della ragione per decidere come agire. Il practical reasoning si distingue dal Theoretical Reasoning, che riguarda solo le credenze non le azioni.

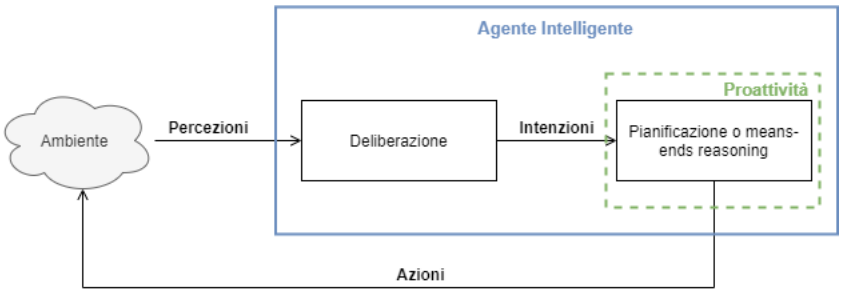
Il Practical Reasoning è caratterizzato da due attività:

- Deliberazione: Quale stato di cose vogliamo raggiungere (l'output sono le **intenzioni**).
- Pianificazione o means-ends reasoning: Come possiamo raggiungere gli obiettivi (l'output è il soddisfacimento delle intenzioni, raggiungere lo stato di cose).

Deliberazione e means-ends reasoning sono processi computazionali, come tali in tutti gli agenti reali questi processi avranno risorse limitate (e.g. il tempo, lo spazio, ...). Quindi il calcolo è una risorsa preziosa per gli agenti perché serve per controllare il suo ragionamento ed è importante che non agiscano a tempo indeterminato, perché anche il tempo è una risorsa. **Con intenzioni ci riferiamo allo stato di cose che un agente ha scelto di raggiungere**.

5.3 Proattività e Intenzioni

Anche se un agente è un sistema informatico è intelligente, non perché ha consapevolezza o capacità extra-programma, ma se riusciamo ad attribuirgli alcune caratteristiche, tra cui, la **proattività**. Il ruolo delle intenzioni è quello di favorire la proattività, cioè tendono portare ad agire, all'azione.



Il filosofo Brattman osserva che le intenzioni giocano un ruolo molto più forte nell'influenzare l'azione rispetto ad altri atteggiamenti proattivi come il desiderio. Infatti **le intenzioni non possono essere in contrasto, mentre invece i desideri si**.

Quindi possiamo dire che le intenzioni hanno le seguenti proprietà:

- **Problemi**: Le intenzioni pongono problemi agli agenti, se ho un intenzione $\varphi \Rightarrow alloco\ risorse : \exists piano\ per\ eseguire\ \varphi$

- **Le intenzioni vincolano il futuro practical reasoning**, quindi non posso generare intenzioni che vanno in conflitto tra loro: Se ho il desiderio di andare in piscina ma devo andare a lavoro, la piscina rimane un desiderio, il lavoro l'intenzione.
- **Persistenza**: Gli agenti monitorano il successo delle loro intenzioni e sono inclini a riprovare se i loro tentativi falliscono, solo se la motivazione dell'intenzione non sussiste più, è razionale abbandonare tale intenzione.
- Gli agenti credono che le loro intenzioni siano possibili. Se voglio andare in piscina credo che la piscina sia aperta e che io posso fare la mia attività.
- Gli agenti non credono che non porteranno avanti le loro intenzioni.
- In determinate circostanze, gli agenti credono che realizzeranno le loro intenzioni..
- **Effetti collaterali**: Gli agenti non devono necessariamente avere intenzione su tutti gli effetti. collaterali delle loro intenzioni (e.g. ho una carie, affronto il dolore per curarla).

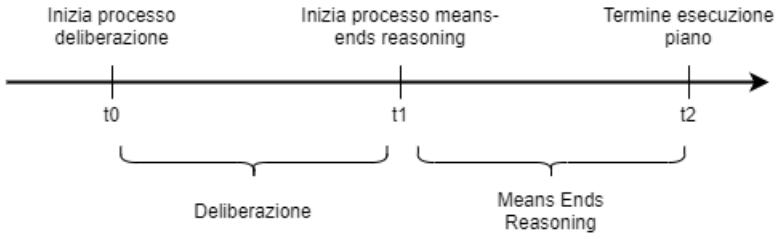
6 Realizzazione di Agenti che Sfruttano il Practical Reasoning

6.1 Agent Control Loop Version 1

```
while true do:
    # deliberazione
    observe the world;
    update internal world model;
    deliberate about what intention to achieve next;

    # pianificazione
    use means–ends reasoning to get a plan;
    execute the plan;
```

I processi di deliberazione e means-ends reasoning non sono istantanei, hanno un costo in termini di tempo, questo significa che quando ottengo un piano potrebbe non essere più quello che mi serve. Osservando questa timeline possiamo fare alcune deduzioni:



Definiamo come tempo per deliberare:

$$T_{deliberazione} = t_1 - t_0$$

$$T_{means-ends-reasoning} = t_2 - t_1$$

La **deliberazione è ottimale** se al tempo t_1, l'agente ha selezionato l'intenzione da raggiungere che sarebbe stata ottimale se fosse stata raggiunta all'istante t_0. A meno che il tempo $T_{deliberazione}$ sia estremamente piccolo, l'agente corre il rischio che l'intenzione selezionata non sia pi'u ottimale nel momento in cui l'agente l'ha deliberata. Ma la fase di deliberazione è solo metà del problema: l'agente deve ancora determinare come realizzare l'intenzione.

Quindi, **l'agente avrà un comportamento complessivo ottimale** nelle seguenti circostanze:

- $T_{deliberazione} \to 0$: Quando il tempo impiegato per i processi di deliberazione e di means-ends reasoning è incredibilmente piccolo.
- Quando l'ambiente è garantito rimanere statico mentre l'agente sta eseguendo i processi di deliberazione e di means-ends reasoning.
- Quando un'intenzione ottimale se raggiunta al tempo t_0 (momento in cui si osserva l'ambiente) è garantita rimanere ottimale fino al tempo t_2 (momento in cui l'agente ha determinato le azioni per raggiungere l'intenzione).

6.2 Agent Control Loop Version 2

```
# initial beliefs B := B0;
while true do
    get next percept ρ;
    B := brf(B, ρ);
    I := deliberate(B);
    π := plan(B, I);
    execute(π)
```

Analizziamo lo pseudo codice soprastante con particolare attenzione alla funzione **brf**, che sta per Belief Revision Function, ovvero la funzione che si occupa della revisione delle credenze:

- *get next percept ρ;* registriamo le credenze rilevati in qualche modo (e.g. tramite l'utilizzo di sensori).
- *B := brf(B, *ρ*);* Aggiorna i beliefs B sulla base di quelli correnti e delle credenze (non monotona perché devo rivedere le mie credenze o eliminarne qualcuna).
- *I := deliberate(B);* deliberiamo le intenzioni basate sulla credenze aggiornate.
- π := *plan(B, I);* pianificazione di un piano π.
- *execute(π)* esegue i piani.

6.3 BDI: Belief Desire Intention

6.3.1 Agent Control Loop Version 3

Belief Desire Intention (BDI) è un tipo di architettura per agenti, applicata nella seguente implementazione, che inizia cercando di capire quali sono le opzioni a sua disposizione (desires) sulla base delle proprie informazioni e credenze (beliefs), in ultimo sceglie tra le opzioni possibili e definisce delle intenzioni (intentions).

```
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);
    execute(π)
```

A differenza della versione precedente (v2) dovremmo definire dei desires D basandoci su B e I. Inoltre assumiamo che a t_0 di avere beliefs B e intentions I definite. Analizziamo le novità rispetto al codice precedente:

- *D := options(B, I);* l'agente genera un insieme di possibili alternative, desideri D.
- *I := filter(B, D, I);* Quando un'opzione è restituita da **flter** e quindi è scelta dall'agente come intenzione I, diciamo che l'agente ha fatto un **commitment**, ovvero ha preso un impegno, verso tale opzione.

Il **commitment** implica **persistenza temporale**, un'intenzione, una volta adottata, non dovrebbe immediatamente essere abbandonata. Quindi viene automatico chiedersi quanto a lungo un intenzione dovrebbe persistere? Il meccanismo che una agente usa per determinare quando e come un'intenzione possa

essere abbandonata è conosciuta come **commitment strategies**. Sono state identificate tre principali strategie:

1. **Blind Commitment (Fanatico)**: Un agent blind committed continuerà a mantenere un'intenzione fino a quando non crederà che l'intenzione sia stata effettivamente raggiunta. Il blind commitment è talvolta indicato anche come fanatical commitment.
2. **Open-Minded Commitment**: Un agente open-minded committed manterrà un'intenzione fintanto che la ritiene, la crede possibile.
3. **Single-Minded Commitment (Risoluto)**: Un agente single-minded committed continuerà a mantenere un'intenzione perché non ritiene che l'intenzione sia stata raggiunta, o che non sia più possibile raggiungerla.

Agent Control Loop Version 3 è un agente **overcommited**, cioè impegnato in modo eccessivo ad ultimare le sue intenzioni, questo perché il piano π potrebbe richiedere troppo tempo senza considerare i cambiamenti esterni.

6.3.2 Agent Control Loop Version 4: Ripianificazione

La ripianificazione, nel caso in cui qualcosa vada storto o se le intenzioni I non sono più corrette, riduce il commitment dell'agente. Introduciamo la funzione **sound** che simula i passi di un piano e verifica che siano ancora corrette. Questa funzione ha costo lineare a differenza della *plan* che è più complessa.

```
B := B_0;
I := I_0;
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) do
        # head(π) restituisce la prima azione del piano
        α := head(π);
        execute(α);

        # nuovo piano: le restanti azioni (la coda)
        π := tail(π);
        get next percept ρ;
        B := brf (B, ρ);
        if not sound(π, I , B) then
            π := plan(B, I )
```

L'agente è ancora **overcommited** rispetto le intenzioni: non si ferma mai valutare se le sue intenzioni siano o meno adeguate.

6.4 Riconsiderare le Intenzioni

6.4.1 Agent Control Loop Version 5

Per ridurre ulteriormente il commitment modifichiamo l'agente della versione 4 per far si che si fermi a determinare se le intenzioni hanno avuto successo o se sono diventate impossibili da soddisfare (single-minded commitment). Quindi l'agente può **riconsiderare le sue intenzioni I** ogni volta che il controllo ci dice che:

- **not empty(π)**: Ci sono azioni nel piano.
- **succeeded(I , B)**: Ritiene raggiunte con successo le sue attuali intenzioni.
- **impossible(I , B)**: Crede che le sue attuali intenzioni non siano più possibili.

```
B := B_0;
I := I_0;
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
        # head(π) restituisce la prima azione del piano
        α := head(π);
        execute(α);

        # nuovo piano: le restanti azioni (la coda)
        π := tail(π);
        get next percept ρ;
        B := brf (B, ρ);
        if not sound(π, I , B) then
            π := plan(B, I)
```

6.4.2 Agent Control Loop Version 6

Possiamo migliorare la versione precedente facendogli riconsiderare le intenzioni I dopo l'esecuzione di ogni azione del piano.

```
B := B_0;
I := I_0;
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
        # head(π) restituisce la prima azione del piano
        α := head(π);
        execute(α);

        # nuovo piano: le restanti azioni (la coda)
        π := tail(π);
        get next percept ρ;
        B := brf (B, ρ);
        D := options(B, I);
        I := filter(B, D, I);

        if not sound(π, I , B) then
            π := plan(B, I)
```

C'è da sottolineare il fatto che riconsiderare delle intenzioni costa? Un agente che non si ferma per riconsiderare le proprie intenzioni abbastanza spesso continuerà a tentare di raggiungere le sue intenzioni anche se sono impossibili da raggiungere, d'altro canto un agente che riconsidera costantemente le sue intenzioni potrebbe dedicare un tempo insufficiente a raggiungerle effettivamente.

6.4.3 Agent Control Loop Version 7

In questa implementazione incorporiamo una esplicita componente di controllo di meta-livello che decide se eseguire o meno la riconsiderazione, il metodo **reconsider**. Recondider è una componente di controllo che decide sulla base di un euristica (insieme di strategie, tecniche e procedimenti) che dipende dal dominio.

```
B := B_0;
I := I_0;
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
        # head(π) restituisce la prima azione del piano
        α := head(π);
        execute(α);

        # nuovo piano: le restanti azioni (la coda)
        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)
```

6.4.4 Possibili interazioni

Le possibili interazioni la componente di controllo di meta-livello, ovvero la funzione *reconsider*, e deliberazione sono:

Situation number	Chose to deliberate?	Changed intentions?	Would have changed intentions?	*reconsider*(...) optimal?
1	No	—	No	Yes
2	No	—	Yes	No
3	Yes	No	—	No
4	Yes	Yes	—	Yes

1. L'agente non ha scelto di deliberare, e come conseguenza, non ha scelto di cambiare le intenzioni, se avesse scelto di deliberare, non avrebbe cambiato le intenzioni. In questa situazione, la funzione *reconsider* si sta comportando in modo ottimale perché mi risponde di non deliberare.

2. L'agente non ha scelto di deliberare, ma se lo avesse fatto avrebbe cambiato le intenzioni. In questa situazione, la funzione *reconsider* non si sta

comportando in modo ottimale.

3. L'agente ha scelto di deliberare, ma non ha cambiato intenzioni. In questa situazione, la funzione reconsider non si comporta in modo ottimale.

4. L'agente ha scelto di deliberare e ha cambiato intenzioni. In questa situazione, la funzione reconsider si sta comportando in modo ottimale.

Quindi il comportamento della funzione *reconsider* dipende da quanto cambia l'ambiente che porta ad un cambio delle intenzioni.

6.4.5 Riconsiderazione ottimale delle intenzioni

Kinny e Georgeff investigano in maniera sperimentale l'efficacia delle strategie, nello specifico su due diversi tipi di strategia di riconsiderazione sono stati utilizzati:

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

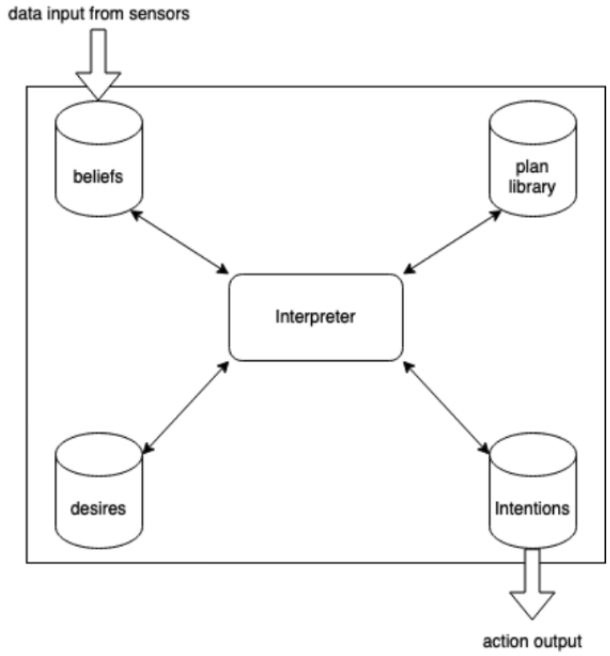
Possiamo definire γ come il grado di dinamismo dell'ambiente dove, se γ è basso l'ambiente non cambia rapidamente invece se γ è alto l'ambiente cambia rapidamente:

- γ **alto, ambiente dinamico**: Gli agenti audaci preformano meglio perché quelli cauti perdono tempo a riconsiderare i propri impegni mentre agenti audaci sono impegnati a lavorare per raggiungere le loro intenzioni.
- γ **basso, ambiente statico**: Gli agenti cauti tendono a preformare meglio perché sono in grado di riconoscere quando le intenzioni non sono più corrette, traendo anche vantaggio da situazioni fortuite e nuove opportunità quando si presentano.

6.5 PRS: Procedural Reasoning System

PRS, che sta per Procedural Reasoning System, fu la prima architettura per agenti sviluppata ad includere il paradigma BDI (Belief, Desire, Intention) ed ha diverse applicazioni, come ad esempio il sistema di controllo del traffico aereo all'aereoporto di Sydney.

L'Architettura può essere rappresentata come segue:



E' un interprete che utilizza una librerie di piani precompilati e un insieme di desideri in combinazione con dei belief per determinare delle azioni basate su intenzioni.

Ponendo l'attenzione sulla **libreria di piani** possiamo pensare al comportamento di un semplice programma che, con i suoi blocchi condizionali, va a identificare specifici comportamenti da attuare durante la sua esecuzione e si comporta come un *pianificatore*, maggiori sono i blocchi condizionali più saranno i casi d'uso che il programma riuscirà a gestire. Le architetture PRS anziché utilizzare un pianificatore utilizza una libreria di piani che sono precompilati dai programmatori, che sono costituiti da:

- **Goal**: Post condizione del piano.
- **Contesto**: Precondizione del piano (prerequisiti).
- **Corpo**: Sequenza di azioni

Quindi **il piano viene scelto dall'interprete nella libreria sulla base del Goal e del Contesto**. Ma in PRS i piani possono essere più complessi, in

particolare possono contenere a loro volta altri Goal. Quindi un agente PRS ha un insieme di piani, alcuni belief iniziali riguardo l'ambiente (e.g. rappresentati come fatti Prolog) e un goal iniziale.