Agenti Intelligenti

Appunti delle lezioni di
Paolo Alfano

Note Legali

Liberatoria, aggiornamenti, segnalazione errori:

Quest'opera viene pubblicata in formato elettronico senza alcuna garanzia di correttezza del suo contenuto. Il documento, nella sua interezza, è opera di Paolo Didier Alfano
e viene mantenuto e aggiornato dallo stesso, a cui possono essere inviate eventuali segnalazioni all'indirizzo *paul15193@hotmail.it*
Ultimo aggiornamento: 03 giugno 2017

Indice

07/03/2017

1 Introduzione

Nel corso della storia dell'informatica sono stati perseguiti diversi obiettivi

- Ubiquità: nel tempo, il ridursi dei costi degli apparecchi elettronici ha portato ad una diffusione capillare degli strumenti tecnologici
- Interconnessione: la capacita' di far comunicare parti distribuite. Nel momento in cui le parti possiedono una loro intelligenza, siamo in presenza di sistemi di agenti intelligenti distribuiti(un esempio di sistema di agenti intelligenti distribuiti è l'internet of things)
- Intelligenza: quando viene data una certa "intelligenza" agli agenti. Dovremo soffermarci in seguito su una migliore definizione di intelligenza
- Delega: lasciare che un sistema svolga dei compiti anche in situazioni critiche
- Orientamento al comportamento umano: l'avvicinamento tra la visione orientata alle macchine a quella pensata per l'uomo

Delegare e dare intelligenza agli agenti implica che gli agenti siano indipendenti e che cerchino di operare per ciò che è meglio per noi. L'interconnessione unita alla delega implica invece che i sistemi intelligenti siano in grado di accordarsi o competere tra loro.
Per tutti questi motivi diversi studiosi degli agenti intelligenti sostengono che gli agenti intelligenti siano la prossima evoluzione degli studi informatici.

Cerchiamo di dare una prima definizione di *agente*.
Potremmo dire che un agente è un sistema capace di svolgere azioni indipendentemente da un eventuale controllore.
Nel momento in cui abbiamo un'insieme di agenti, diremo che abbiamo un *sistema multiagente*. Un sistema multiagente può avere diversi obiettivi: cooperare, coordinare, negoziare. I primi due casi riguardano una collaborazione tra due sistemi, il terzo è un caso di accordo tra due agenti in competizione.
L'obiettivo di questo corso è mostrare

1. La strutturazione degli agenti
2. La loro capacita' di interagire con altri agenti

A questo punto un'opposizione che potremmo porre ai sistemi multiagente è che non sono altro che sistemi concorrenti/distribuiti: in realta' quello che conta nei sistemi multiagente è la loro capacita' di prendere decisioni indipendenti.
Allora potremmo pensare che i sistemi multiagente siano soltanto intelligenza artificiale: anche in questo caso non avremmo ragione perché nel caso di agenti

intelligenti vogliamo costruire agenti fisici.
Nel complesso possiamo dire che i sistemi multiagente si collocano tra argomenti diversi come: sistemi distribuiti, intelligenza artificiale, scienze sociali ed economia.

08/03/2017

Concludiamo questa introduzione andando a vedere i vari livelli che studieremo

1. Livello agente: è il livello piu' semplice e studia come sia strutturato un singolo agente
2. Livello di interazione: va a studiare come vari agenti operano tra loro senza bisogno di supervisione umana. L'interazione studia come gli agenti cooperino, come comunichino e come negozino tra loro
3. Livello organizzativo: studia l'assegnazione dei ruoli ai vari agenti e l'auto organizzazione di questi ultimi. In particolare con l'auto organizzazione intendiamo studiare il processo di adattamento a seguito di cambiamenti del goal o dell'ambiente. In diversi ambienti è anche importante considerare anche alcuni parametri come *reputazione*, *fiducia* e altri parametri ancora

Abbiamo anche altri due livelli un po' a sé stanti

- Infrastrutture e tecnologie di supporto: comprendere quali siano le tecniche e per sviluppare gli agenti. Studieremo questa parte in laboratorio e vedremo linguaggi basati sulla logica(Prolog-like). Parleremo anche diffusamente di un approccio tipico dello studio degli agenti detto *BDI* (Beliefs, Desires, Intentions)
- Metodi formali: analisi formale dei sistemi ad agenti multipli. Per studiare tali sistemi vengono utilizzati dei formalismi logici che descrivano sia gli stati mentali degli agenti che e le possibili interazioni nel sistema.

2 Agenti intelligenti

Diamo una definizione piu' formale di *agente*.
Un'agente è un sistema di computazione che esegue azioni in modo autonomo in un certo ambiente.
Mostriamo in Figura 1 come l'agente interagisce con l'ambiente

Dunque l'agente riceve un'insieme di input dall'ambiente e reagisce eseguendo delle azioni che operano sull'ambiente.
Un esempio di agente è il termostato. Sottolineiamo di nuovo che la differenza tra lo studio degli agenti e dell'IA è che nell'IA l'interazione con l'ambiente non è fondamentale.

Un *agente intelligente* è un sistema di computazione flessibile che è reattivo, proattivo e sociale:

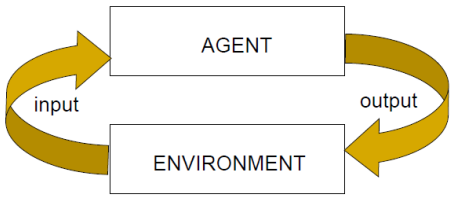

Figura 1: Relazione tra agente e ambiente

- Reattivo: capace di rispondere subito a uno stimolo
- Proattivo: ha un proprio obiettivo che cerca di raggiungere. Dunque l'agente non è solo guidato dagli eventi ma anche dall'obiettivo. Bilanciare la reattività e la proattività è una sfida importante nell'ambito degli agenti intelligenti.
- Abilita' sociale: capisce quale è il miglior comportamento da tenere in presenza di altri agenti per raggiungere un obiettivo. L'interazione tra agenti viene realizzata tramite un certo linguaggio di comunicazione tra agenti

Esistono delle ulteriori caratteristiche secondarie come, *mobilità*, *veracità*, *benevolenza*...
Notiamo che dalla definizione potremmo dire che anche l'essere umano è fondamentalmente un agente intelligente.
Esistono anche alcune differenze tra agenti e oggetti: gli agenti sono infatti autonomi, hanno una loro forma di intelligenza e sono attivi.
Un altro punto di vista interessante da dare è quello degli agenti come *sistemi intensionali*. Infatti quando spieghiamo l'attività umana lo facciamo cercando di mostrare un nesso tra le azioni del soggetto: "Janine prende un ombrello perché pensa che pioverà". Analogamente dovremmo cercare di fare lo stesso per gli agenti intelligenti. Si è lavorato molto sotto questo punto di vista, sviluppando linguaggi ad hoc e ponendosi dilemmi filosofici. Ad esempio se sia legittimo attribuire questi *stati mentali* agli agenti. Alcuni importanti studiosi sostengono che sia un passo richiesto solo nel caso in cui possa esserci utile in qualche misura, ovvero quando siamo di fronte a situazioni complesse(evitiamo di dare degli stati mentali al termostato).
Notiamo che questo procedimento è tipico dell'informatica: a fronte di situazioni più complesse cerchiamo di astrarre per rappresentare tali situazioni
Possiamo dare una struttura dell'architettura astratta di un agente e dell'ambiente.

- Stato attuale dell'ambiente: identificato tramite uno stato appartenente all'insieme degli stati $E = \{e, e^{'}, \ldots\}$
- Azioni applicabili: insieme di azioni $Ac = \{\alpha, \alpha', \ldots\}$

Un *run* è l'insieme degli stati attraversati tramite le varie azioni:
$run : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} e_n$
L'ultimo elemento del run puo' essere o uno stato o un'azione. Diremo allora che l'insieme dei run R si divide in due parti:

- R^{Ac}: insieme dei run che finiscono con una azione
- R^{E}: insieme dei run che finiscono con uno stato ambiente

A questo punto possiamo definire più formalmente l'*ambiente Env* come la tripla formata da

$$Env =< E, e_0, \tau >$$

dove E è l'insieme degli stati possibili dell'ambiente, e_0 è lo stato iniziale e τ è la funzione di trasformazione di stato

$$\tau : R^{Ac} \rightarrow p(E)$$

che a partire da un certo run che finisce per azione porta in un'insieme di stati dell'ambiente che dipende dall'azione eseguita.
La "funzione degli agenti", ovvero quello che un agente deve fare è in qualche modo simile:
$Ag : R^E \rightarrow Ac$
Notiamo quindi che la decisione dell'agente è basata sul run, quindi su tutta la storia precedente.
Di fatto diremo che la sequenza $(e_0, \alpha_0, e_1, \alpha_1, \ldots)$ rappresenta il run di un agente Ag in un ambiente Env se

1. e_0 è lo stato iniziale di Env
2. $\alpha_0 = Ag(e_0)$
3. $\forall u > 0, e_u \in \tau(e_0, \alpha_0, e_1, \alpha_1, \ldots)$ dove $\alpha_u = Ag((e_0, \alpha_0, e_1, \alpha_1, \ldots))$

L'idea poco realistica di questo modello è che l'agente per prendere una decisione vada a considerare l'intera storia passata.
Consideriamo il caso completamente opposto, quello di un agente che non considera quanto accaduto precedentemente. Tali agenti vengono detti *agenti puramente reattivi*. Anche in questo caso possiamo dire che il termostato è un esempio di agente puramente reattivo.
Riprendiamo lo schema mostrato in Figura 1 e definiamolo meglio per un agente puramente reattivo. Lo schema di tale agente viene mostrato in Figura 2
Forniamo invece in Figura 3 lo schema di un agente che abbia un proprio stato.
Notiamo che l'agente con stato sfrutta le funzioni

$$see : E \rightarrow perc \qquad next : \Delta \times perc \rightarrow \Delta \qquad action : \Delta \rightarrow Ac$$

dove Δ è l'insieme degli stati.
Nel complesso il nostro agente all'interno di questa architettura opera secondo il seguente ciclo:

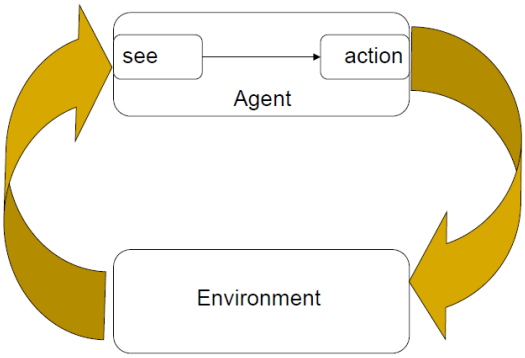

Figura 2: Schema di un agente puramente reattivo

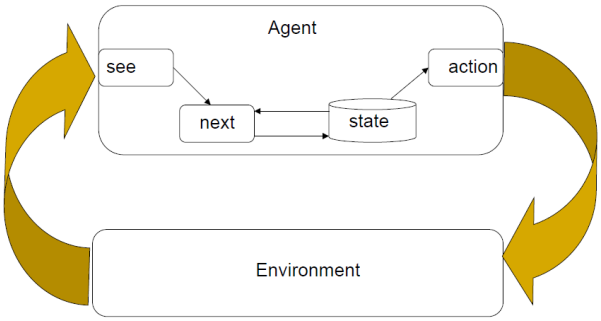

Figura 3: Schema di un'agente con stato

1. Parte da uno stato iniziale i_0
2. Osserva lo stato dell'ambiente e tramite la funzione *see* genera una percezione
3. Aggiorna lo stato tramite la funzione *next*
4. Seleziona l'azione da eseguire mediante la funzione *action*
5. Torna al punto 2

Come fare però a implementare un agente a cui dicendo cosa fare, capisca come farlo? Potremmo associare ad ogni stato una certa *utilità* ma questo non funziona molto bene nel caso il compito da eseguire sia realizzato da un run con utilità minima; con questo intendiamo dire che è difficile specificare una visione a lungo termine delle azioni da eseguire. È molto meglio specificare un utilità per l'intero run. Possiamo specificare una funzione ψ che dato un run restituisce 0 nel caso in cui il task non venga completato e che restituisce 1 se invece il task

è portato a termine. Ovvero

$$\psi : R \to \{0, 1\}$$

09/03/2017

3 Agenti a ragionamento deduttivo

Nel precedente capitolo abbiamo accennato alla differenza tra agenti puramente reattivi e a quelli con stato.
Possiamo dire allora che gli agenti sono di tre tipi:

- Reattivi
- Simbolici
- Ibridi

Originariamente tutti gli agenti erano basati su IA simbolica. Alla meta' degli anni '80 vengono proposti nuovi modelli ad agenti reattivi. Negli anni '90 emergono le prime soluzioni ibride.
L'idea degli agenti simbolici è quella di vedere gli agenti come un particolare tipo di sistema basato su conoscenza. Questa visione presenta pero' due problemi:

- Trasduzione: traduzione del mondo reale in un linguaggio simbolico.
- Ragionamento: come fare affinché la rappresentazione permetta di fornire dei risultati in tempi utili

Notiamo che comunque gli agenti simbolici non sono esclusivamente basati sulla logica visto che effettuano alcune operazioni (come il mantenimento del database) che non hanno nulla a che vedere con la logica.
Un tipo particolare di agenti simbolici sono gli agenti a ragionamento deduttivo. Agenti di questo tipo come stabiliscono le azioni da compiere? In modo semplice potremmo usare la logica. In particolare se abbiamo:

- ρ : l'insieme di regole detto *teoria*
- Δ: un database logico di fatti
- Ac: l'insieme delle azioni possibili

allora possiamo definire le seguenti funzioni: la funzione di percezione
$see : Env \to Per$
la funzione di modifica del database
$next : \Delta \times Per \to \Delta$
la funzione di selezione dell'azione
$action : \Delta \to Ac$

L'idea quindi è che data la logica possiamo stabilire quale azione eseguire. Se non è proibito eseguirla, allora la eseguiamo. In particolare seguiamo un semplice ciclo di questo tipo:

1. Verifichiamo se esiste un'azione esplicitamente prescritta, ovvero per cui valga la condizione
   $\Delta \vdash_\rho Do(a)$
   In questo caso andiamo ad eseguire l'azione a

2. Se non esiste una azione prescritta andiamo a cercare una qualunque azione che non sia esplicitamente vietata, ovvero per cui valga che
   $\Delta \nvdash_\rho \neg Do(a)$ In questo caso andiamo ad eseguire l'azione a

3. Se nessuna azione rispetta i due punti precedenti, allora nessuna azione è applicabile

Ad ogni modo questo non sarà il nostro approccio nello studio degli agenti intelligenti.

4 Practical reasoning

Cominciamo ad introdurre l'approccio intenzionale a BDI a cui abbiamo accennato in precedenza. Lo studiamo non tanto perché sia il modello piu' diffuso, ma perché è certamente il modello piu' noto.
In questo modello supponiamo di avere a disposizione degli agenti in grado di ragionare. Allora possiamo dire che la *practical reasoning* è la capacita' di un agente di ragionare sulle azioni. Si distingue dalla *theoretical reasoning* perché quest'ultima è guidata strettamente dalle credenze mentre la practical è guidata dalle azioni.
Il practical reasoning umano si struttura in due parti

1. Deliberazione(cosa): decidere quale stato vogliamo raggiungere. Questa fase produce in output le *intenzioni*

2. Means-ends(come): come raggiungere lo stato deciso in fase di deliberazione

Chiaramente le due fasi appena viste possiedono un costo computazionale di cui l'agente deve tener di conto. È anche per questo che ad un certo punto l'agente stabilirà un piano e cercherà di portarlo a termine.

14/03/2017

In tutto questo giocano un ruolo importante le intenzioni. Un agente che possieda una certa intenzione fara' di tutto per realizzarla e la abbandonera' solo nel momento in cui vengano meno le motivazioni per perseguirla.
Per essere più precisi le intenzioni portano diverse conseguenze

- pongono dei vincoli sui ragionamenti che dovrà compiere l'agente nel seguito

- Fungono da filtro per l'aggiunta di altre intenzioni(contrastanti) che potrebbero generare dei conflitti

- Gli agenti credono di poter portare a termine le proprie intenzioni

- introducono il problema per l'agente di capire come portarle a termine
- gli agenti potrebbero non comprendere tutte le conseguenze delle loro intenzioni. In particolare pur credendo che $\phi \Rightarrow \psi$ e sapendo che vale ϕ non necessariamente vorremo intendere anche ψ

Esempio 1 : nonostante io sappia che andare dal dentista implichi soffrire, e nonostante io sappia di doverci andare, questo non implica che intenda soffrire!

4.1 Implementare un agente practical reasoning

Sapendo che un piano è una sequenza di azioni valide, possiamo dire che un agente che fa uso di practical reasoning dovrebbe eseguire il seguente ciclo

```
while (true)
   observe the world;
   update internal world model;
   deliberate about what intention to achieve next;
   use means-ends reasoning to get a plan to satisfy the intention;
   execute the plan
```

Delle due prime fasi non ci interessiamo, ma nel seguito andremo a raffinare tale ciclo fornendo ulteriori dettagli sulle altre fasi.
Supponiamo infatti che il nostro agente inizi la fase di delibera al tempo t_0, che inizi la fase di means-ends al tempo t_1 e cominci l'esecuzione al tempo t_2. Allora possiamo dire che il tempo di delibera e di means-ends sono:
$t_{del} = t_1 - t_0$
$t_{me} = t_2 - t_1$
Il rischio che corriamo è che mentre l'agente elabora t_{del} la situazione dell'ambiente circostante cambi, rendendo non ottimale la decisione presa. Quindi la cosa migliore è che t_{del} e t_{me} siano ridotti al minimo, oppure che il mondo circostante sia statico.
Adesso possiamo invece specificare meglio la fase di delibera. Si compone di due parti:

1. Generazione delle opzioni: viene definito l'insieme di possibili alternative.
2. Filtraggio: in questa fase l'agente sceglie tra varie alternative

In tutto questo un ruolo importante viene ricoperto anche dal *commitment*, ovvero l'impegno preso dall'agente nel soddisfare un'intenzione.
In particolare esistono delle strategie di commitment che mostrano come un agente cerchi di portare a termine i propri compiti, ovvero di soddisfare le sue intenzioni. Esistono tre principali strategie di commitment

- Blind: l'agente cerchera' di portare a termine il suo compito sempre e comunque fino a quando non lo conclude

- Single-minded: l'agente cerca di raggiungere l'obiettivo ma se si accorge che non è piu' soddisfacibile, si interrompe, è legato ai beliefs
- Open-minded: simile al precedente ma maggiormente legato ai goal

15/03/2017

Vediamo adesso come eseguire un piano. Dobbiamo considerare il piano come un'insieme di azioni. A questo punto la execute esegue una e una sola azione α alla volta.
Scomporre il piano in un'insieme di azioni ci permette di andare a valutare come cambia il mondo dopo ogni azione. In questo modo possiamo sapere se il resto del piano è ancora eseguibile. Nel caso non lo sia possiamo andare a calcolare ed eseguire un nuovo piano.
Questo tipo di agente puo' essere ben utilizzato con politiche di commitment single-minded a patto di introdurre una condizione che verifichi la raggiungibilita' del goal dopo l'esecuzione di ogni azione.
A questo punto potremmo pensare di creare un agente ancora piu' flessibile che va ad aggiornare le sue intenzioni durante l'esecuzione del piano. Forse pero' questo è un po' eccessivo perché cambiare continuamente le intenzioni di un agente risulterebbe troppo costoso e non è neanche detto che porterebbe a miglioramenti significativi. Cerchiamo quindi una via di mezzo introducendo una funzione *reconsider* che controlla se sia il caso di aggiornare le intenzioni. Ovviamente la *reconsider* deve costare poco perché abbia senso usarla. Inoltre non è sicuro che il piano basato sulle nuove intenzioni sia migliore del precedente.
Nel complesso, il nostro agente practical reasoning segue questo ciclo

```
B=B0                          // Initial beliefs
I=I0                          // Initial intentions
while (true){
   get next percept p
   B= brf(B, P)               // belief revision function
   D=options(B, I)            // options generation
   I=filter(B, D, I)          // options filtering
   pi = plan(B, I)            // plan generation
   /* we continue until the plan is over, or we are sure that
   *  1)we've reached our goal
   *  2)we'll never reach our goal*/
   while(not(empty(pi)||succeeded(B, I)||impossible(B, I))){
      a = head(pi)            // take first action of the plan
      execute(a)              // execute the action
      pi = tail(pi)           // update plan by removing a
      get next percept p
      B= brf(B, P)            //after an action we reconsider our beliefs
      if(reconsider(B, I)){
         D=options(B, I)
         I=filter(B, D, I)
      }
      if(not sound(pi, B, I)){   //Old plan not valid anymore
         pi = plan(B, I)
```

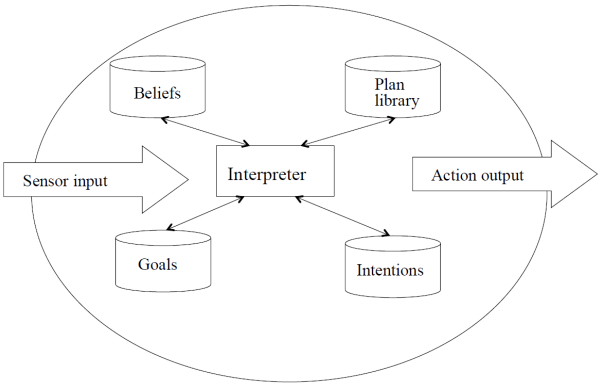

Figura 4: Architettura logica del linguaggio PRS

```
    }
  }
}
```

Ad ogni modo un esperimento ad opera di Kinny e Georgeff ha valutato l'effettiva utilita' del riconsiderare le strategie, distinguendo in due categorie gli agenti:

- Agenti bold: si fermano a riconsiderare le loro intenzioni molto raramente
- Agenti cautious: si fermano a riconsiderare le intenzioni dopo ogni passo

Senza grandi sorprese è stato verificato che quando l'ambiente tende ad essere statico, gli agenti bold si comportano meglio di quelli cautious, viceversa quando l'ambiente è dinamico gli agenti bold si comportano peggio

5 Linguaggi per agenti BDI

5.1 PRS

Il *Procedural Reasoning System* è uno dei primi linguaggi per BDI e ha un importante valore storico. Diversi linguaggi si sono basati sul PRS per implementare sistemi multiagente complessi come i sistemi per il controllo aereo, la diagnosi di sistemi. Inoltre accenniamo a tale linguaggio perché nel seguito ne vedremo un altro che posa le sue basi proprio sul PRS. Mostriamo in Figura 4 l'architettura logica di un agente PRS.

Gli input sono eventi che possono essere *esterni* o *interni*. L'output sono le azioni.
Ad ogni modo, in questo schema le due parti piu' interessanti sono i piani contenuti nelle *plan library* e le intenzioni contenute nella parte *intentions*.

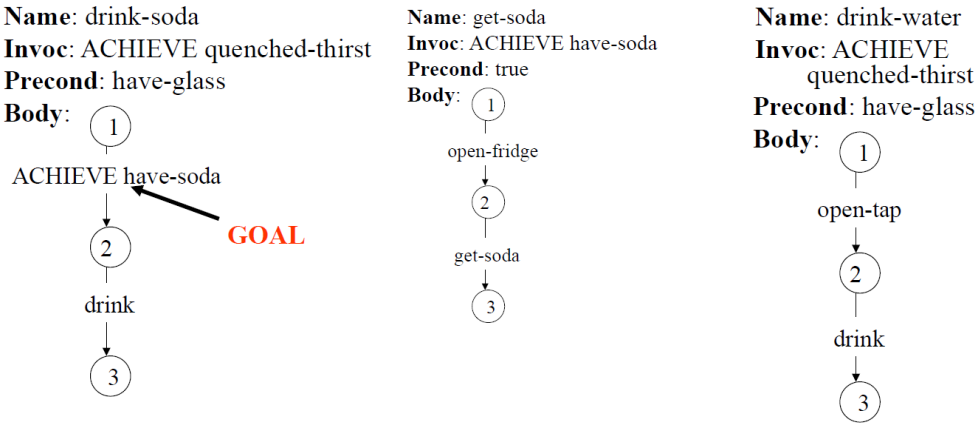

Figura 5: Due piani da eseguire in PRS

I piani sono composti da: un nome, una condizione che lo attiva, un insieme di precondizioni affinché il piano possa essere attivato. Infine possiede un corpo, formalmente strutturato come DAG[1], ma che possiamo anche vedere come un'insieme di azioni da eseguire. Formalmente gli elementi di un piano possono essere azioni atomiche o sottogoal. Vediamo un piano in un esempio

Esempio 1 : supponiamo di voler eseguire due piani, drink-soda e drink-water. La struttura dei piani viene mostrata in Figura 18

Notiamo che drink e open tap sono azioni basilari, mentre have-soda, che viene preceduta dalla label ACHIEVE è un altro goal da raggiungere per verificare quello corrente. Dunque anche have-soda ha un suo piano da eseguire

Il ciclo di un agente PRS si compone di cinque fasi

1. In base all'input vengono aggiornati goal e beliefs
2. I cambiamenti al goal e ai beliefs permettono l'eventuale attivazione di diversi piani
3. Uno o più piani eseguibili sono inseriti nella parte di intentions effettuando un'operazione di pattern matching tra gli eventi che si verificano e gli eventi scatenanti contenuti all'interno della descrizione di un piano(indicati tramite l'etichetta Invoc)
4. Viene selezionata una intention da portare a termine. Notiamo che i piani all'interno della parte di intentions sono inseriti come in uno stack. Supponiamo di inserire due piani nella parte intentions. Se eseguiamo il

[1] Directed Acyclic Graph

primo piano (che magari invoca diversi sottopiani) e abbiamo un fallimento, semplicemente prenderemo il successivo piano sullo stack e lo andremo ad eseguire.

5. Viene eseguito un passo del piano che realizza l'intention

16/03/2017

5.2 AOP

Un altro modo di programmare agenti è quello *AOP*, ovvero *Agent-Oriented Programming* che comprende un semplice linguaggio formale e un linguaggio di programmazione che è un'estensione di LISP, detto AGENT-0.
Nell'AOP abbiamo due categorie di entita': *mentali* e *non mentali*. Nelle categorie mentali troviamo i *belief* e i *commitment*. Nella categoria non mentale troviamo invece le *capability*. Vediamo queste entita' piu' nel dettaglio

- Belief: $B_a^t\phi$ indica che l'agente a crede ϕ al tempo t
- Commitment o obligation: $OBL_{a,b}^t\phi$ indica che al tempo t l'agente a esegue un commit verso l'agente b riguardo ϕ. Un caso particolare di commitment è la decisione, ovvero un commitment verso sé stessi: $DEC_a^t\phi = OBL_{a,a}^t\phi$
- Capability: $CAN_a^t\phi$ indica che al tempo t l'agente a è capace di ϕ.

Esempio 1 : vediamo un esempio di belief
$B_a^3 B_b^{10} like(a,b)^7$: indica che al tempo 3 l'agente a crede che al tempo 10 l'agente b crederà che al tempo 7 ad a piaceva b

I costrutti sintattici principali di un programma del linguaggio AGENT-0 sono:

- Fatti: (t atom)
- Azioni private: (DO t action)
- Azioni comunicative: (INFORM t a fact) oppure (REQUEST t a action)
- Condizioni mentali: che sono formate da pattern mentali del tipo
(B fatto) or ((OBL a)azione)
- Azioni condizionali: sono legate alle condizioni mentali infatti sono del tipo (IF mentalCondition action)
- Commitment: si verifica quando vengono soddisfatte alcune condizioni mentali e alcune condizioni sui messaggi. Ha forma (COMMIT messagecond mentalcond (agent action)*)

Nel complesso possiamo dire che ogni agente è formato da quattro componenti:

- Un'insieme di capabilities, ovvero quello che un agente può fare

- Un'insieme di credenze
- Un'insieme di commitment, ovvero quello che l'agente vuole fare
- Un'insieme di regole di commit: che determinano il comportamento dell'agente

Di fatto l'interprete esegue i seguenti passi: legge i messaggi correnti e aggiorna i suoi belief e commitment, dopo esegue il commitment basandosi sul tempo corrente. Notiamo quindi che in questo modello i belief vengono aggiornati dopo che un agente viene informato di qualcosa.

21/03/2017

6 Fondamenti logici

6.1 Problematiche della logica classica

Andiamo a vedere i fondamenti logici che ci serviranno per strutturare un agente intelligente. L'IA ha dato una spinta notevole allo sviluppo di nuovi formalismi per la logica. Tali formalismi ci permettono di rappresentare la conoscenza e di verificare certe proprieta' di modo che si possa andare a specificare il comportamento degli agenti.
Dobbiamo quindi costruire una logica che ci permetta di specificare il comportamento degli agenti. La logica classica potrebbe funzionare? In generale no. Consideriamo questo esempio

Esempio 1 : supponiamo di voler rappresentare il seguente fatto "John crede che stia piovendo". Nella logica classica potremmo rappresentarlo come segue
$Bel(John, piove)$
Purtroppo Bel non può essere visto come un operatore della logica classica questo perché il suo secondo argomento è una formula e non un termine come previsto dalla logica classica. Infatti nella logica classica gli operatori sono puramente funzionali: sono ovvero definiti dal valore di verita' dei loro argomenti. Ad esempio il valore di verita' di $p \wedge q$ è funzione esclusivamente di p e di q.
Invece Bel non è puramente funzionale visto che $Bel(John, piove)$ non dipende solo e soltanto dal valore di verita' di $piove$.

Abbiamo quindi due modi di procedere

1. Modificare la logica classica in qualche modo per ottenere un meta linguaggio
2. Usare una logica modale che fornisca operatori non puramente funzionali

Noi ci concentreremo maggiormente sulla seconda opzione.

6.2 Logica modale

La semantica della logica modale è stata sviluppata da Kripke ed è espressa in termini di *mondi possibili*. Di fatto ogni mondo possibile rappresenta uno stato. Quello che vedremo è che esisteranno delle situazioni *certamente vere* e altre *possibili*.
Chiaramente la logica modale non è completamente scollegata dalla logica classica infatti esistono alcuni concetti comuni

- L'utilizzo di una proposizione a
- L'utilizzo della disgiunzione $a \vee b$
- L'utilizzo della negazione $\neg a$
- L'esistenza dei *modelli*. Un modello è infatti un'istanziazione possibile delle variabili proposizionali. Chiaramente avendo a disposizione n predicati avremo 2^n modelli possibili. Diremo inoltre che una formula è *soddisfacibile* se esiste un modello che la soddisfi. Diremo invece che una formula è *valida* quando tutti i modelli la soddisfano
- L'utilizzo della conseguenza logica $KB \models \alpha$ che vale solo se per ogni modello in cui vale KB vale anche α
- L'utilizzo di leggi ben note come il *modus ponens* o la *regola di risoluzione*

La logica modale aggiunge invece due nuovi operatori: $\Box$ e $\Diamond$

- $\Box$: l'argomento è necessariamente vero
- $\Diamond$: l'argomento è possibilmente vero

Questi due operatori sono detti *modali*. Notiamo che sono uno il duale dell'altro. Infatti:
$\Box(\phi) = \neg\Diamond(\neg\phi)$
Potremmo allora usare uno dei due simboli, ma non lo faremo solo per avere una notazione piu' coincisa.
In questa logica il corrispettivo del modello è un'insieme di mondi possibili. Piu' precisamente un modello è una tripla:
$< W, R, L >$ dove:

- W è un'insieme di mondi collegati in qualche modo da una R
- $R \subseteq W \times W$ è una relazione di accessibilita' che lega due mondi
- $L : W \to 2^P$ è l'insieme delle proposizioni vere in un certo mondo

22/03/2017

Essendo W rappresentabile da un'insieme di mondi legati fra loro da relazioni, possiamo allora rappresentare un modello tramite un grafo. L'idea è di asserire che per una certa proprieta' ϕ vale che $\Box\phi$ per un mondo x se e solo se ϕ vale in tutti i mondi accessibili da x.
Vediamolo meglio tramite un esempio:

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>}

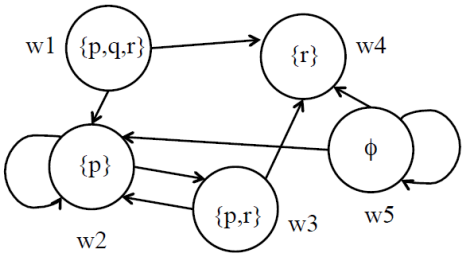

Figura 6: Rappresentazione grafica di un modello nella logica modale

Esempio 1 : mostriamo in Figura 6 un modello nella logica modale

Consideriamo diverse proprieta':

- $\Box p$ è vera in w_2 poiché p vale in tutti i mondi raggiungibili da w_2, ovvero w_2 e in w_3
- $\Diamond r$ è verificata in w_2 infatti r è vera in almeno un mondo accessibile da w_2, ovvero w_3.
- $\Diamond(r \wedge \Box q)$ è vera in w_1. Questa è un poco piu' complicata da verificare ma procediamo come segue. Affinché $\Diamond(r \wedge \Box q)$ sia vera, deve essere vera $(r \wedge \Box q)$ in un qualche mondo accessibile da w_1. I due mondi accessibili da w_1 sono w_2 e w_4. Per w_2 non è vero che $(r \wedge \Box q)$ visto che già solo r in w_2 non è verificato.
  $(r \wedge \Box q)$ è invece verificato per w_4. Infatti in w_4 abbiamo che r è banalmente vera mentre $\Box q$ è vera poiché w_4 non ha mondi accessibili e dunque la proprieta' è verificata[2].

Notiamo dunque che alcune proprieta' che potrebbero intuitivamente sembrare vere in realta' non lo sono. Ad esempio
$\Box\alpha \nRightarrow \Diamond\alpha$
Questo succede perché nel caso di nodi "pozzo"(come il nodo w_4 dell'esempio precedente) $\Diamond\alpha$ non è detto che sia verificato, visto che $\Diamond$ è quantificato come un quantificatore esistenziale.
Anche altre proprieta' sono piu' difficili da dimostrare. Ad esempio la soddisfacibilita' di un modello dipende da un certo mondo w: $M \models_w \phi$
Invece la validita' di un modello dipende da tutti i mondi.
Ad ogni modo esistono delle proprieta' che sono sempre valide:

[2]Ricordiamo infatti che in logica la semantica del $\forall\emptyset$ è equivalente a true mentre $\exists\emptyset$ è equivalente false

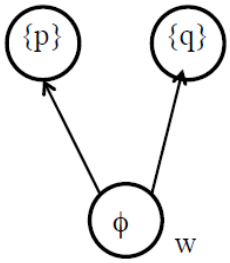

Figura 7: Modello dell'esempio 2

- $\Box(\phi \Rightarrow \psi) \Rightarrow (\Box\phi \Rightarrow \Box\psi)$
- $\phi \Rightarrow \Box\phi$
- $\Box(\phi \wedge \psi) \equiv (\Box\phi \wedge \Box\psi)$
- $\Box(\phi \vee \psi) \not\equiv (\Box\phi \vee \Box\psi)$

Esempio 2 : mostriamo come $\Box$ non distribuisca sul $\vee$. Consideriamo il modello in Figura 7

Risulta evidente che in questo modello è vero che $\Box(p \vee q)$ in w ma non è vero che $\Box p \vee \Box q$

6.3 Logica modale per l'IA

Cerchiamo adesso di capire come si possa utilizzare la logica modale per i nostri agenti. Come abbiamo detto un problema della logica classica è che gli argomenti non potevano essere delle formule. Nella logica modale questo non succede e dunque tramite essa possiamo andare a specificare i beliefs degli agenti.
Useremo allora una notazione leggermente differente in cui $B\phi$ va a rappresentare $\Box\phi$. Ovviamente modificare solo la notazione non puo' bastare. Andiamo allora a modificare qualcosa nella logica modale per adattarla.
Modificando la logica modale possiamo ottenere diversi tipi di logica

- Logica epistemica (conoscenza): i cui componenti fondamentali sono
  - $K_a\phi$ l'agente a sa che ϕ è vero
  - $B_a\phi$ l'agente a crede che ϕ sia vero
- Logiche BDI: in cui abbiamo
  - $B_a\phi$ l'agente a crede che ϕ sia vero
  - $D_a\phi$ l'agente a desidera ϕ
  - $I_a\phi$ l'agente a ha l'intenzione ϕ
- Logiche deontiche

- Logiche temporali
- Logiche dinamiche

Possiamo dire quali sono le modifiche che potremmo aspettarci. Ad esempio una buona proprieta' che ci aspetteremmo essere verificata è che se $\Box$ rappresenta la conoscenza, vorremmo che la logica avesse la proprietà che tutto ciò che è conosciuto è vero.
Questo può essere espresso aggiungendo l'assioma $\Box\phi \Rightarrow \phi$.
In alternativa all'aggiunta di assiomi, è possibile restringere la classe dei modelli richiedendo che la relazione di accessibilità R abbia certe proprietà:

- Riflessiva: $\forall w \in W.(w, w) \in R$
- Seriale: $\forall w \in W.\exists w' \in W.(w, w') \in R$
- Transitiva: $\forall w, w', w'' \in W.((w, w') \in R \wedge (w', w'') \in R) \Rightarrow (w, w'') \in R$
- Euclidea: $\forall w, w', w'' \in W.((w, w') \in R \wedge (w, w'') \in R) \Rightarrow (w', w'') \in R$

Ad ogni modo esiste una *teoria della corrispondenza* che mette in relazione ognuna di queste proprieta' con un assioma corrispondente nella logica modale. Infatti vale che

Sigla	Modale	Proprietà
T	$\Box\phi \Rightarrow \phi$	Riflessiva
D	$\Box\phi \Rightarrow \Diamond\phi$	Seriale
4	$\Box\phi \Rightarrow \Box\Box\phi$	Transitiva
5	$\Diamond\phi \Rightarrow \Box\Diamond\phi$	Euclidea

Come gia' avevamo accennato, combinando le varie proprietà otteniamo diversi tipi di logiche ognuna con diverse caratteristiche. Tali logiche sono legate tra loro e possono essere inserite in un reticolo.

23/03/2017

6.4 Logiche epistemiche

A questo punto possiamo cercare di analizzare meglio le logiche a cui abbiamo accennato. Cominciamo con le logiche di tipo epistemico.
Come avevamo gia' accennato in queste logiche indichiamo con $K\phi$ il fatto che l'agente sappia ϕ e indichiamo con $B\phi$ il fatto che l'agente creda ϕ. Dal punto di vista semantico $K\phi$ è equivalente a $\Box\phi$. Per questo motivo possiamo avere delle formule analoghe a quelle gia' viste in precedenza, ovvero che:

- $K(\phi \Rightarrow \psi) \Rightarrow (K\phi \Rightarrow K\psi)$ (assioma K)
- se ϕ è valida anche $K\phi$ lo è (assioma di necessita')

Questi due assiomi hanno una loro utilita'. Supponiamo ad esempio di sapere che $\phi \Rightarrow \psi$. Dall'assioma di necessita' sappiamo che $K(\phi \Rightarrow \psi)$. Ma allora dall'assioma K abbiamo che $(K\phi \Rightarrow K\psi)$.
Vediamo come invece le quattro proprieta' si applichino alle logiche epistemiche

- Riflessiva: $K(\phi) \Rightarrow \phi$
  ovvero che quello che l'agente sa è vero
- Seriale: $K(\phi) \Rightarrow \neg K(\neg\phi)$
  cioè che l'agente non possiede informazioni contraddittorie. Se è a conoscenza di ϕ non puo' essere anche a conoscenza di $\neg\phi$
- Transitiva: $K\phi \Rightarrow KK\phi$
  ovvero che se l'agente conosce ϕ allora sa di conoscere ϕ
- Euclidea: $\neg K\phi \Rightarrow K(\neg K\phi)$
  che vuol dire che se l'agente non sa ϕ allora sa di non saperlo

Notiamo che le ultime due proprieta' garantiscono all'agente di essere a conoscenza di quello che sa e non sa.
In generale la logica epistemica puo' anche contenere variabili e quantificatori. Per utilizzare i quantificatori dobbiamo pero' estendere la semantica dei mondi fornendo un dominio e una funzione che associ ogni termine al dominio. Quello che pero' rende un po' difficile l'utilizzo dei quantificatori è che ad ogni modo si debba associare un dominio diverso.
Vediamo un esempio per capire meglio come applicare la logica epistemica

Esempio 1 : supponiamo che due bambini dopo aver giocato nel fango vogliano sapere chi si è sporcato la fronte. Quello che sanno è che almeno uno di loro ha del fango sulla fronte e ogni bambino puo' vedere la fronte dell'altro ma non la propria.
Il bambino B dice "Non so se ho la fronte infangata"
Il bambino A dice "Io so di avere la fronte infangata"

Mostriamo in Figura 8 la serie di passaggi che dobbiamo effettuare per ottenere un risultato.

Possiamo quindi concludere che entrambi i bambini sono sporchi di fango in fronte.

6.5 Logiche deontiche

Adesso vediamo brevemente le logiche deontiche. In queste logiche abbiamo due operatori fondamentali O(obligation) e P(permission). Questi due operatori corrispondono agli operatori $\Box$ e $\Diamond$. Queste logiche possiedono solitamente due assiomi

- $O_p \Rightarrow P_p$
- $F_p \equiv \neg P_p$ dove F è un operatore di divieto

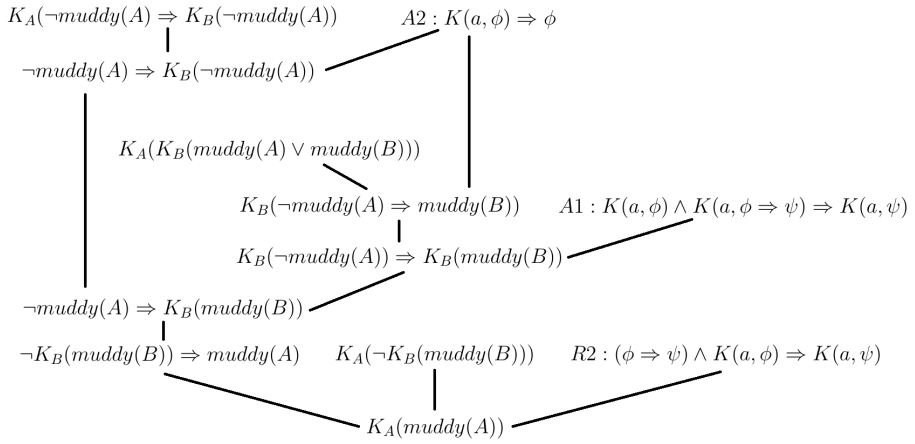

Figura 8: Esempio dei bambini sporchi di fango

6.6 Logiche temporali

La logica temporale è una logica modale del tempo. Ne esistono molte varianti. In particolare il tempo puo' essere discreto, avere un momento iniziale ed essere infinito nel futuro. Gli operatori sono solo rivolti ad eventi futuri. In particolare esistono due logiche temporali molto importanti

- Linear Time logic: la struttura del tempo è lineare
- Branching Time logic: la struttura del tempo è un albero

28/03/2017

Studieremo con maggiore cura le logiche temporali perché gli agenti evolvono nel tempo e dunque è importante analizzare proprio il tempo.
Cominciamo andando a studiare la logica linear time. Gli elementi tipici di tale logica sono

- α: una normale proposizione che assume valore $true$ o $false$
- $\alpha \wedge \beta$
- $\neg\alpha$
- $X\alpha$(next time α): indica che α sara' valido nel momento successivo
- $\alpha \bigcup \beta$(α until β): indica che α è vero dal momento corrente fino ad un futuro momento in cui β sara' vero

Da queste componenti di base possiamo ricavarne due ulteriori che sono molto interessanti

- $F\alpha \equiv true \bigcup \alpha$: indica che ad un certo punto nel futuro si verifichera' α

- $G\alpha \equiv \neg F \neg \alpha$: indica che α è sempre verificato. Infatti $\neg F \neg \alpha$ indica che non è mai vero che ad un certo punto nel futuro sarà verificato $\neg \alpha$

Avendo a disposizione questi elementi possiamo dire che un modello M è una tripla del tipo:

$$< S, x, L >$$

dove

- S è un'insieme di stati
- x: $\mathbb{N} \to S$ è una sequenza infinita di stati
- L: $S \to 2^P$ fornisce l'insieme delle proposizioni vere in ogni stato

Nel seguito indicheremo anche x^i quella sequenza di stati a partire dallo stato $i - esimo$ in poi. Ovvero se $x = (s_0, s_1, \ldots)$ allora $x^i = (s_i, s_{i+1}, \ldots)$.

Notiamo che per come abbiamo definito questa logica stiamo definendo un tipo particolare di logica modale in cui vale che
$\forall w \in W. \exists! w' \in W. (w, w') \in R$
Con questo intendiamo dire che andando a rappresentare tramite automi il dominio, ogni mondo(come era stato definito nella logica modale) è collegato ad un solo altro mondo.
Consideriamo un paio di esempi per comprendere meglio quanto stiamo dicendo

Esempio 1 : la formula $G(p \Rightarrow Xq)$ è verificata da tutte quelle sequenze in cui in ogni stato in cui p è vero, nello stato immediatamente successiva q è vero

Esempio 2 : la formula $Gp \wedge F \neg p$ non è mai soddisfatta perché stiamo dicendo che nel futuro p sarà sempre vero ma allo stesso tempo stiamo dicendo che ad un certo momento nel futuro sarà anche falso e questo è impossibile

L'altro tipo di logica temporale è quella branch time in cui al tempo viene data una forma ad albero. La logica branch time generalizza la linear time. Infatti nella branch time abbiamo due tipi di formule:

- path formulas: sono formule che si riferiscono ad un solo percorso, come accadeva nella logica linear time. Sfruttano un operatore dedicato $E\pi$(exists) per verificare π su un certo path
- state formulas: sono formule che si riferiscono a tutti i cammini. Sfruttano un operatore dedicato $A\pi$ (all) per verificare π su tutti i percorsi

Una logica branching time è la logica CTL^* che è di fatto piu' generale di una qualunque logica linear time. Ad ogni modo visto che la logica CTL^* è troppo complicata da usare, si preferisce usarne una sua restrizione detta CTL. Non è invece possibile fare confronti di generalità tra CTL e le logiche linear time.

6.7 Ragionamento sulle azioni

Un agente intelligente deve essere in grado di ragionare riguardo gli effetti delle sue azioni sul mondo. Ragionamenti di questo tipo fanno parte del *situation calculus*.
Nel situation calculus abbiamo

- Situazioni: stati del mondo in un certo momento
- Fluenti: proposizioni il cui valore dipende dalla situazione
- Azioni: causano un cambiamento del mondo

Vediamo un esempio per chiarirci le idee

Esempio 1 : consideriamo la seguente situazione. All'inizio Fred è vivo. John aspetta un certo tempo carica la pistola e spara a Fred, uccidendolo. Vediamo le varie componenti del nostro problema.

Fluenti: *caricata* e *vivo*
Azioni: *aspetta, carica, spara.*

Abbiamo anche delle precondizioni: infatti *carica* è eseguibile solo se vale $\neg caricata$.
Gli effetti delle azioni sono:
carica causa *caricata*
aspetta non ha alcun effetto
spara causa $\neg vivo$

Il situation calculus puo' essere realizzato con la logica del primo ordine.
Visto che un'azione influenza solo una piccola parte dei fluenti, come possiamo esprimere in modo semplice che ogni altra cosa non cambia? Questo problema viene detto *frame problem*[3] e il modo migliore per risolverlo è stabilire che ogni fluente mantiene il suo precedente valore sempre che non sia inconsistente con il nuovo stato.
Un altro problema è quello della ramificazione. In questo caso abbiamo problemi a gestire gli effetti indiretti di una azione. Infatti le azioni hanno un effetto diretto ma possono avere anche degli effetti indiretti.

6.8 Logica dinamica

È una logica modale per le azioni. Questo tipo di logica permette di ragionare riguardo i programmi regolari. Infatti il linguaggio L_D fa uso del linguaggio L_R dei programmi regolari. In particolare gli elementi di L_D sono

- α
- $\phi \wedge \psi$

[3] Dal fatto che tra un frame e il successivo nei film non cambia quasi niente

- $\neg\phi$
- $[p]\phi$: il suo significato intuitivo è che nel momento in cui p termina, il suo stato di terminazione deve soddisfare ϕ
- $\langle p\rangle\phi$: il suo significato intuitivo è che è possibile eseguire p affinché termini in uno stato che soddisfi ϕ

28/03/2017

7 Linguaggi logici per agenti

Adesso che abbiamo visto più formalmente le logiche possiamo chiederci come utilizzarle per implementare sistemi di agenti. Abbiamo fondamentalmente due scelte

- Rifinire manualmente la specifica in una forma eseguibile utilizzando processi di raffinamento informali. Tecniche di questo tipo si avvicinano all'ingegneria del software
- Tradurre o compilare la specifica in una forma computazionale utilizzando delle tecniche di traduzione automatica. Mostriamo nel seguito alcuni linguaggi che possono essere eseguiti direttamente

7.1 AgentSpeak(L)

Proposto come implementazione per sistemi BDI(in particolare PRS), in questo linguaggio gli agenti operano utilizzando clausole simili a quelle di Horn. In questo linguaggio abbiamo *Beliefs*, *Goals* e *triggering events*. In particolare questi ultimi aggiungono e rimuovono goal. I piani in questo linguaggio sono della forma

$$triggeringEvent : preconditions <- body$$

Ogni agente possiede dei propri beliefs, un insieme di piani di eventi di azioni e di intenzioni. Inoltre ogni agente possiede tre funzioni di selezione S_E, S_O ed S_I: S_E seleziona un evento da processare, S_O sceglie un piano attivato dall'evento e infine S_I seleziona un intenzione da eseguire

7.2 GOLOG

Presentato nel 1994, GOLOG specifica le azioni dando un insieme di precondizioni e di effetti delle stesse. In particolare le azioni primitive sono

- Azione: $Do(a,\ s,\ s') =_{def} Poss(a,\ s) \wedge s' = do(a,\ s)$
  dove $Poss(a, s)$ indica che valgono le precondizioni per eseguire l'azione a nello stato s e $do(a, s)$ indica l'esecuzione dell'azione a nello stato s
- Azione di test: $Do(\phi?,\ s,\ s') =_{def} \phi \wedge s = s'$

- Sequenza di azioni:
  $Do([\delta 1;\ \delta 2],\ s,\ s') =_{def} \exists s''.Do(\delta 1,\ s,\ s'') \wedge Do(\delta 2,\ s'',\ s')$
- Scelta non deterministica:
  $Do((\delta 1 \mid \delta 2),\ s,\ s') =_{def} Do(\delta 1,\ s,\ s') \vee Do(\delta 2,\ s,\ s')$
- Scelta non deterministica di azioni
- Iterazione non deterministica: particolare perché è una definizione del secondo ordine visto che le chiusure transitive non possono essere definite nella logica del primo ordine

Queste azioni e la formalizzazione di sequenze più complesse di azioni si basano sulla logica dinamica.
Basandosi su queste azioni primitive, su ALGOL e sui principi della programmazione logica, eseguire un programma vuol dire stabilire la seguente dimostrazione
$Axioms \models \exists S.Do(program, S_0, S)$
dove S_0 è la situazione iniziale.
In particolare l'esecuzione di un programma GOLOG restituisce una istanziazione delle variabili che soddisfi la situazione finale S.
In questo senso GOLOG è molto simile a PROLOG in quanto l'obiettivo di entrambi è quello di legare variabili quantificate esistenzialmente a dei valori che soddisfino il problema. Infatti GOLOG è un dimostratore automatico di teoremi per la logica del secondo ordine.
Sono state realizzate diverse versioni di GOLOG: CONGOLOG che implementa la concorrenza, IndiGOLOG che può essere eseguito intervallato ad azioni, pianificazione ed eventi esogeni e infine DyLog che sfrutta le logiche modali andando a definire procedure che sono interpretate come assiomi logici.

7.3 Concurrent METATEM

È un linguaggio basato sulla diretta esecuzione di formule temporali. In particolare sulle logiche temporali lineari.
Un sistema METATEM è formato da vari agenti che comunicano tramite messaggi asincroni in broadcast. Il comportamento è definito tramite logica temporale.
In METATEM il tempo è modellato come una serie infinita di stati discreti, con un punto iniziale. Alcuni operatori sul "futuro" sono

- $\bigcirc\phi$: ϕ deve essere soddisfatto nello stato seguente
- $\phi \cup \psi$: ϕ deve essere vero fino a quando non sarà vero ψ
- $\Diamond\phi$: ϕ deve essere soddisfatto ad un certo stato nel futuro
- $\Box\phi$: ϕ deve essere soddisfatto in tutti gli stati futuri

Alcuni operatori per il "passato" sono

- $\phi S \psi$: ϕ era soddisfatto fino a ψ

- $\circ\phi$: ϕ era soddisfatto nello stato precedente

Vediamo qualche esempio

Esempio 1 : vediamo qualche esempio con gli operatori appena visti
$\Box importante(agenti)$: significa che ora e sempre gli agenti saranno importanti
$\Diamond importante(ConcurrentMetatem)$: indica che in un certo momento futuro, concurrentMetatem diventerà importante $\bigcirc scusa(tu)$: significa che nello stato successivo tu ti scuserai

Un programma METATEM è composto da regole del tipo

$\Box(formule\ sul\ passato\ o\ sul\ presente \Rightarrow formule\ sul\ presente\ o\ sul\ futuro)$

L'interprete METATEM esegue continuamente il ciclo:

1. Controlla quali regola hanno precondizioni soddisfatte
2. Congiunge i conseguenti di tali regole
3. Riscrive le congiunzioni in forma disgiuntiva e scegli una disgiunzione da eseguire

Se viene trovata una contraddizione effettua backtracking a una scelta precedente.
I messaggi sono invece rappresentati da *predicati* che possono essere: di ambiente(per i messaggi in ingresso), di componente(per messaggi in uscita) e interni.

29/03/2017

8 Model Checking

Il model checking è un metodo per la verifica di proprieta' di sistemi concorrenti che in particolare puo' essere usato per la verifica di programmi multiagente.
Il model checking si basa sulle logiche temporali e preso un modello M in una logica L vuole verificare che una certa formula ϕ sia verificata in M.
In particolare noi useremo la logica temporale lineare.
L'analisi di un programma concorrente avviene tramite lo studio di un *transition system* che possiamo vedere come una struttura di Kripke in cui tracciamo i vari mondi possibili come nelle logiche modali. I cammini all'interno di tale struttura sono le computazioni possibili.
Purtroppo l'utilizzo di logiche temporali lineari porta con sé qualche problema perché i cammini in una logica lineare sono infiniti e quindi per verificare una certa ϕ dobbiamo verificarla per tutti i cammini infiniti. Vediamolo con un esempio

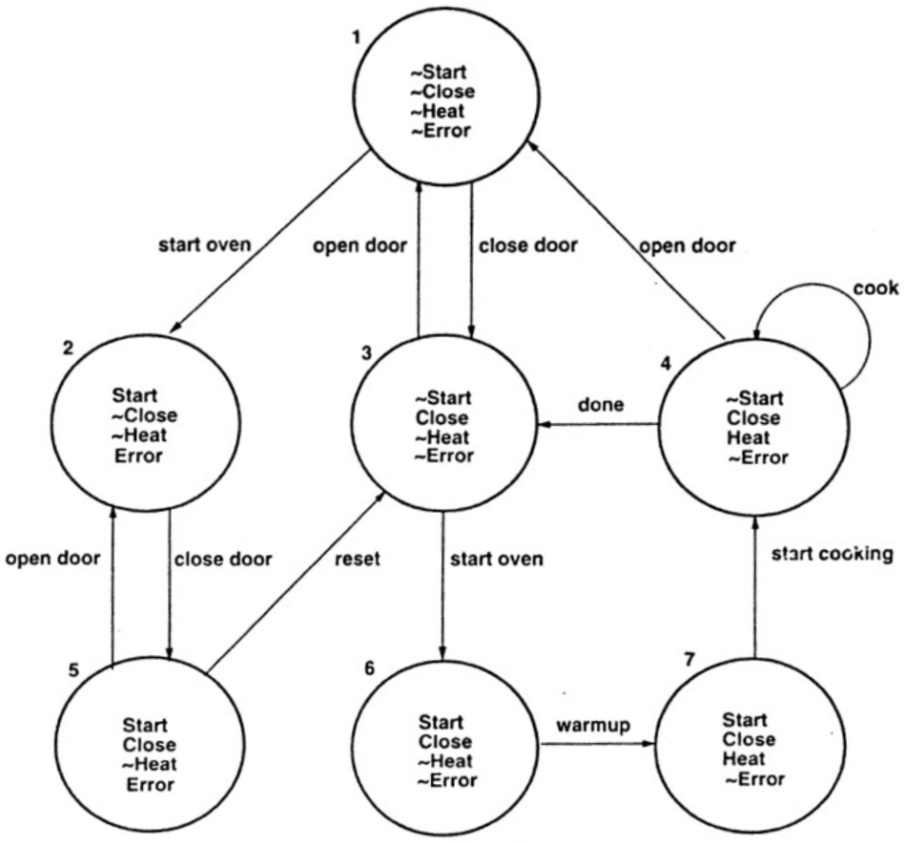

Figura 9: Transition system dell'esempio 1

Esempio 1 : supponiamo di modellare un forno a microonde secondo il transition system mostrato in Figura 9

Supponiamo di voler dimostrare se valga $G(Start \Rightarrow FHeat)$. Dunque dobbiamo verificare che per ogni cammino da ogni stato in cui vale $Start$, in un qualche momento futuro varra' $Heat$.
È abbastanza facile capire che questa proprieta' non è vera visto che è sufficiente considerare un cammino che si sposta ciclicamente fra gli stati 2 e 5. In tal modo $Heat$ non sara' mai verificato e dunque la formula è falsa.

Ad ogni modo cerchiamo di mostrare ora quale sia il procedimento generale per dimostrare una formula logica linear time.
Per farlo costruiamo un *automa di Büchi*. In questi automi abbiamo la classica quintupla:

$$< \Sigma, S, \delta, S_0, F >$$

dove Σ è un'insieme di simboli, S l'insieme degli stati, $\delta : S \times \Sigma \times S$ è una funzione di transizione, S_0 l'insieme degli stati iniziali mentre F l'insieme degli stati finali.
La particolarita' degli automi di Büchi è che tutti gli stati sono considerati finali e che accetta stringhe di lunghezza infinita. Questo ci sara' utile perché nel seguito diremo che una stringa w viene accettata se un certo stato f con $f \in F$ appare infinite volte nel percorso seguito tramite w. Possiamo anche generalizzare gli automi di Büchi prendendo qualcosa dai transition systems. Infatti possiamo fare in modo che in un certo stato di un'automa di Büchi valgano alcune proposizioni. Inseriamo negli automi di Büchi la L che apparteneva alla tripla $< W, L, R >$ delle logiche modali.
Con questo nuovo formalismo misto possiamo andare a rappresentare sia il modello di un sistema che le formule da verificare.
Come fare dunque a dimostrare che una certa formula ϕ è vera? Come succede in altre branche verifichiamo ϕ per refutazione.
Operativamente procediamo con i seguenti passi

1. Costruiamo i due automi $B(\pi)$ e $B(\neg\phi)$ dove $B(\pi)$ è l'automa che rappresenta il sistema e $B(\neg\phi)$ che rappresenta la formula da verificare per refutazione. L'automa del sistema è facile da costruire(come nell'esempio del microonde). L'algoritmo che genera un automa a partire da una formula LTL è abbastanza complicato, non lo vedremo

2. Dobbiamo poi costruire l'automa intersezione(ovvero il prodotto) dei due automi $B(\pi)$ e $B(\phi)$

3. Se l'intersezione è vuota la formula è certamente verificata. Se invece l'automa intersezione non è vuoto potrebbe esistere un run che genera un controesempio, quindi dobbiamo verificare se esista all'interno dell'automa prodotto un cammino che termini con un loop contenente uno stato $f \in F$. Se tale cammino esiste allora ϕ non è verificato. Altrimenti ϕ è verificato.

Notiamo che comunque queste tecniche possono anche essere usate per pianificare visto che potremmo verificare se $F(\phi)$(cioè se ϕ è prima o poi verificata). Chiaramente i piani che si ottengono in questo modo sono di lunghezza infinita. Per ottenere piani di lunghezza finita è sufficiente aggiungere un loop fittizio nello stato finale.

Un tool di verifica che esegue tutte queste operazioni si chiama SPIN, basato sul linguaggio PROMELA e fornisce un ambiente per la verifica di programmi concorrenti.

05/04/2017

9 Logiche per modelli di agenti BDI

Presentiamo due logiche per modellare in modo formale gli agenti BDI. Sono logiche simili fra loro il cui scopo principale è quello di fornire un supporto

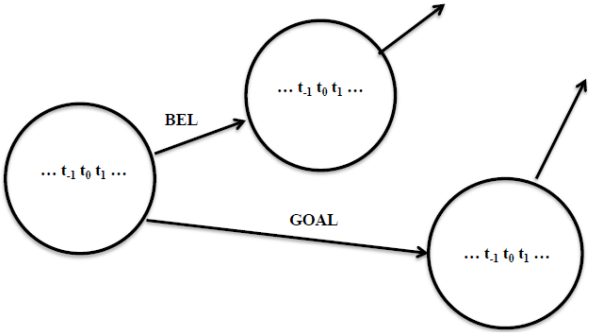

Figura 10: Esempio di schema logico nella logica di Cohen-Levesque

teorico formale agli agenti BDI.
La prima logica che vediamo è quella di Cohen-Levesque.
Questa logica mette insieme alcuni elementi della logica modale con quelli delle logiche temporali linear time.
Abbiamo quattro operatori primari modali

- $(BEL\ i\ \gamma)$: l'agente i crede γ
- $(GOAL\ i\ \gamma)$: il goal dell'agente i è γ
- $(HAPPENS\ \alpha)$: la prossima azione eseguita sara' α
- $(DONE\ \alpha)$: è appena accaduta l'azione α

Possiamo quindi osservare che sui primi due predicati possiamo andare a costruire i predicati piu' complessi della logica. Gli altri due predicati sono temporali. La logica di Cohen-Levesque come dicevamo mette insieme la logica modale e quella linear time. La semantica è un misto delle due logiche. Infatti abbiamo i mondi possibili come nelle logiche modali ma allo stesso tempo ad ogni mondo associamo una sequenza discreta di eventi nel tempo. Dunque cambia anche il concetto di formula valida. Infatti una formula per essere vera deve essere verificata in un certo mondo ad un certo momento. I collegamenti tra un mondo e un altro sono dati dagli operatori BEL e $GOAL$. Mostriamo in Figura 10 lo schema di una possibile situazione nella logica di Cohen Levesque

Un'altra particolarita' di questa logica è che sono stati creati degli operatori per permettere una "traduzione" piu' facile dai concetti a parole alla logica. Operatori di questo tipo sono il $BEFORE$ e il $LATER$ che sono derivabili dai gia' noti operatori $\Box$ e $\Diamond$. In particolare abbiamo che

$(LATER\ \phi) \equiv \neg\phi \wedge \Diamond\phi$
$(BEFORE\ \phi\ \psi)$: ovvero ϕ è verificato prima di ψ

Come vengono espresse le intenzioni di un'agente? Per capirlo dobbiamo prima

definire un *achievement goal*.
L'achievement goal è un obiettivo che l'agente intende raggiungere. Per questo motivo l'agente crede che nello stato corrente l'obiettivo non sia verificato ma sperabilmente nel futuro lo sara', ovvero:

$$(\text{A-GOAL } i\ \phi) \equiv (BEL\ i\ \neg\phi) \wedge (GOAL\ i\ (LATER\ \phi))$$

In particolare un *goal persistente* è un achievement goal che puo' essere rimosso solo per due motivi. O l'agente lo ha raggiunto, oppure si è reso conto che non è mai raggiungibile. Ovvero:

$$(P-GOAL\ i\ \phi) \equiv$$
$$(BEL\ i\ \neg\phi) \wedge$$
$$\wedge\ (GOAL\ i\ (LATER\ \phi)) \wedge$$
$$\wedge\ (BEFORE\ ((BEL\ i\ \phi) \vee (BEL\ i\ \Box\neg\phi))\ \neg(GOAL\ i\ (LATER\ \phi)))$$

A questo punto possiamo dire che un'*intenzione*, realizzata tramite l'operatore $INTEND$, dipende dal goal persistente dell'agente. Infatti un agente intende eseguire α se prima di eseguirla sa di volerla eseguire. In particolare vuole eseguire α nel caso in cui vi sia un goal persistente che lo ha portato in uno stato in cui crede di poter eseguire α e dunque la esegue:

$$(INTEND\ i\ \alpha) \equiv (\text{P-GOAL } i\ (DONE(BEL\ i\ (HAPPENS\ \alpha))))$$

Vediamo invece la logica di Rao e Georgeff.
La differenza fondamentale tra questa logica e la precedente è che nella precedente cercavamo di mettere insieme logica modale con la logica linear time. In questa invece cerchiamo di mettere insieme logica modale e logica branch time. Come potremmo aspettarci la semantica di questa logica è basata su un'insieme di mondi strutturati ad albero detti *time trees* in cui i nodi figli ereditano le proprieta' dei nodi genitore. Le formule generate in questa logica sono simili a quelle generate in CTL^*.
L'idea che sta dietro al funzionamento di queste logiche è che si possa man mano andare a tagliare i rami dell'albero per ottenere dei risultati.
In questa logica valgono ancora alcuni degli operatori definiti nella logica di Cohen-Levesque, come BEL, $GOAL$, $INTEND$, ma anche altri come *succeeds*, $fails \ldots$
In particolare all'interno di questa logica definiamo l'operatore A che ha il seguente significato: "è inevitabile che"/"vale per tutti i futuri". Definiamo inoltre il predicato E che ha il seguente significato: "è possibile che"/"vale per qualche futuro". Tramite questi operatori possiamo dare una definizione più accurata di commitment:

- Blind committment:
  $INTEND(AF\phi) \Rightarrow A(INTEND(AF\phi) \bigcup BEL(\phi))$
- Single minded commitment:

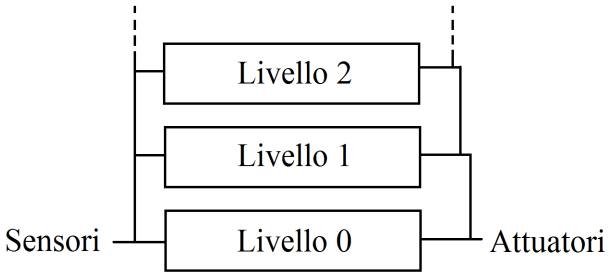

Figura 11: Struttura dell'architettura di Brooks

$(INTEND(AF\phi) \Rightarrow A(INTEND(AF\phi) \bigcup (BEL(\phi) \vee \neg BEL(EF\phi)))$
ovvero l'agente mantiene le sue intenzioni fino a quando crede che siano ancora possibili

- Open minded commitment:
$(INTEND(AF\phi) \Rightarrow A(INTEND(AF\phi) \bigcup (BEL(\phi) \vee \neg GOAL(EF\phi)))$
ovvero l'agente mantiene le sue intenzioni fino a quando queste sono ancora goal

06/04/2017

10 Agenti reattivi e ibridi

Ci sono diversi problemi irrisolti nel campo dell'AI simbolica. Uno dei maggiori problemi non è tanto compiere azioni, quanto interfacciarsi con l'esterno, ovvero quale input ricevere e in che modo riceverlo. Un noto studioso di tali problemi è Rodney Brooks.
In particolare Brooks nei suoi lavori sottolinea che la vera intelligenza deve essere testata nel mondo reale e non in sistemi astratti come i dimostratori di teoremi. In questo senso l'intelligenza dipende da come un'agente si muove nel mondo.
Per questo motivo Brooks propone un'architettura a livelli detta *subsumption architecture*.
In ogni livello sono contenute delle regole che possono essere attivate se valgono determinate condizioni. In generale i livelli piu' bassi sono quelli che eseguono le azioni piu' semplici, ovvero i livelli che reagiscono piu' velocemente.
Ad esempio un agente che si muove in un ambiente reale potrebbe avere al proprio livello 0 una regola che gli impedisca di entrare in contatto con altri oggetti. I livelli superiori potrebbero svolgere azioni piu' complesse come l'esplorazione o l'esecuzione di azioni specifiche.
La struttura dei livelli è mostrata in Figura 11

In realta' ogni livello è strutturato come una macchina a stati finiti per comprendere quale azione eseguire.

L'azione da eseguire è stabilita tramite una funzione di comportamento
$c : p \to a$
in cui p sono le percezioni mentre a sono le azioni. Nel caso in cui si attivino piu' regole da differenti livelli andiamo a considerare la regola proveniente dal livello piu' basso.

Esempio 1 : Brooks ha effettivamente implementato quanto dice su un robot in grado di muoversi in un ambiente roccioso. Consideriamo prima il caso banale di un singolo agente che debba svolgere dei compiti. I suoi livelli potrebbero essere

1. Se viene rilevato un ostacolo, è necessario cambiare direzione
2. Se stiamo trasportando dei campioni e siamo tornati alla base, rilasciare i campioni
3. Se stiamo trasportando dei campioni e non siamo alla base, muoviti verso la base
4. Se viene rilevato un campione, prelevalo
5. Se non c'è niente da fare, muoviti a caso esplorando l'ambiente

I numeri indicano il livello. Dunque prima verifichiamo se debba attivarsi la 1), poi la 2) e cosi' via..
Nel caso in cui il sistema sia multiagente dobbiamo rimuovere la regola 3) e aggiungere un paio di regole per fare in modo che i robot seguano i "sentieri" tracciati da altri robot.

I vantaggi del modello appena visto sono che

- È semplice
- È sufficientemente economico
- Il costo computazionale è abbastanza basso
- Molto resistente contro i fallimenti
- È elegante

Purtroppo questo approccio presenta anche degli svantaggi. Infatti

- Gli agenti che non hanno un modello dell'ambiente devono avere informazioni sufficienti sull'ambiente locale. Questo comporta che tali agenti non sappiano trattare problemi con informazioni non locali. Ovvero effettuano delle scelte sul "breve termine"
- È difficile implementare agenti che imparano
- Non abbiamo una metodologia precisa per implementare gli agenti
- È abbastanza complesso progettare agenti con molti comportamenti

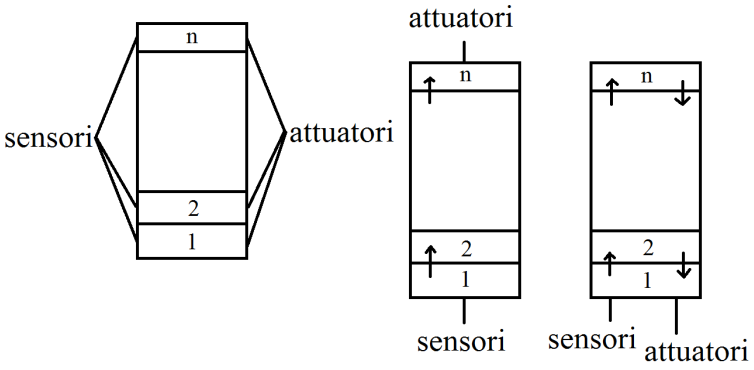

Figura 12: A sinistra un esempio di stratificazione orizzontale. Al centro e a destra due esempi di stratificazione verticale diversi

Per questo motivo preferiamo utilizzare un approccio *ibrido* in cui abbiamo un'architettura in due parti: una reattiva basata sull'architettura di Brooks e una deliberativa che contenga le informazioni sull'ambiente.
Le architetture ibride possono essere strutturate in due modi, orizzontalmente e verticalmente

- Stratificazione orizzontale: ogni livello è direttamente connesso ai sensori e agli attuatori. Ogni strato si comporta come un agente indipendente che suggerisce quale azione intraprendere
- Stratificazione verticale: un solo livello è collegato ai sensori e un solo livello agli attuatori. Tra questi due livelli collochiamo dei livelli intermedi che contribuiscono a decidere quale azione intraprendere

Le due architetture appena illustrate sono mostrate in Figura 12

Alcuni esempi di architetture ibride sono la *TOURINGMACHINES* che possiede una struttura orizzontale per il modulo reattivo e una rappresentazione simbolica del mondo per la parte deliberativa.
Un'altra architettura ibrida è la *InteRRaP* che invece possiede una struttura verticale.

11/04/2017

11 Sistemi multiagente

Nei sistemi multiagente abbiamo un insieme di agenti che comunicano tra loro. Solitamente tali agenti forniscono un'infrastruttura per la comunicazione e l'interazione, sono autonomi, distribuiti, e possono cooperare o essere in competizione fra loro.
Il protocollo di comunicazione specifica quali messaggi possono essere inviati

in relazione alle azioni(di proposta di un'azione, di accettazione, di rifiuto...). Inoltre consideriamo distribuiti i nostri agenti perché possono esserlo geograficamente, possono avere molte componenti...
Abbiamo quattro tecniche per gestire un problema di questo tipo: la modularità, la distribuzione, l'astrazione e l'intelligenza. Nel momento in cui un sistema di agenti implementa queste tecniche, abbiamo un un sistema di intelligenza artificiale distribuita che ci permette di risolvere problemi complessi lavorando in parallelo, in modo modulare e garantendo allo stesso tempo una certa tolleranza ai guasti.
Ovviamente per implementare tutto questo abbiamo bisogno di un *agent communication language*(ACL). In questa direzione si allinea il *Knowledge Sharing Effort* avviato dalla DARPA nel 1990, il cui scopo era quello di sviluppare delle tecniche per condividere e riutilizzare la conoscenza. Per farlo era stato sviluppato un linguaggio, *Knowledge Query and Manipulation Language*(KQML) e un linguaggio logico *Knowledge Interchange Format*(KIF) che esprimesse le proprietà degli oggetti nella base di conoscenza.

Esempio 1 : Le proprietà nel KIF vengono espresse come segue. "La temperatura è di 83 gradi celsius":

(= (temperature m1) (scalar 83 Celsius))

Lo scopo del KQML è quello di fornire un insieme di funzionalità ad alto livello indipendenti dal protocollo di trasporto(tcp/ip, rmi...) e dal linguaggio.
La sintassi di un messaggio KQML è basata su liste simili a LISP. Nel messaggio dobbiamo stabilire le seguenti informazioni: il "mittente", il "destinatario", l'"attuale mittente"(per eventuali intermediari), chi sia il "prossimo destinatario"(per eventuali intermediari), "in risposta a" per stabilire precisamente a quale messaggio si stesse rispondendo, il "linguaggio", il "contenuto"...
Viene anche definita una classe detta *communication facilitators* dedicata all'inoltro dei messaggi.
Inoltre, inizialmente KQML non aveva una propria semantica, ma in seguito sono state fornite per i messaggi delle *precondizioni* e delle *postcondizioni* dall'ovvio significato unite ad una *complete condition* che indica lo stato finale dopo che una certa interazione è stata completata.
Nel 1996 la Foundation for Intelligent Physical Agents (FIPA) propone un proprio ACL, detto FIPA ACL, simile a KQML. Questi due linguaggi si assomigliano molto eccetto per la semantica dove abbiamo qualche differenza e per la mancanza nel FIPA ACL di alcune primitive presenti nel KQML come i communication facilitators.
Ad ogni modo il FIPA ACL utilizza un linguaggio semantico a tre componenti:

- $B_i\gamma$: i crede che valga γ
- $U_i\gamma$: i non è sicuro che valga γ ma ritiene che sia più probabile che valga γ invece che $\neg\gamma$
- $C_i\gamma$: i vorrebbe realizzare γ.

Vengono anche introdotti degli operatori per ragionare riguardo gli agenti:

- $Feasible(a, \gamma)$: a può fare qualcosa che rende vero γ subito dopo
- $Done(a, \gamma)$: a ha eseguito l'azione che renderà vero γ subito dopo
- $Agent(i, a)$: i è l'unico agente che abbia mai eseguito l'azione a

Tramite quanto abbiamo detto finora, nel FIPA ACL vengono definite delle *feasibility preconditions* che devono essere vere per permettere al mittente di inviare correttamente delle informazioni, e vengono definiti dei *Rational effects* che sono gli effetti che un agente può ragionevolmente aspettarsi dall'esecuzione di una certa azione.
Un linguaggio che implementa tutte le specifiche del FIPA ACL è JADE, di cui parleremo nella sezione 13.
Ad ogni modo esistono altri formalismi per implementare i protocolli, come le reti di Petri, alcuni tipi di grammatiche e altri linguaggi che non possiamo mostrare qui.

Per concludere vediamo un tipico protocollo di cooperazione tra agenti: *contract net*.
In contract net esiste un agente che che vuole risolvere un compito. Tale agente viene detto *manager*. Gli agenti che potrebbero risolvere il compito sono detti *contractors*.
Il manager deve: annunciare il compito, ricevere le varie offerte da eventuali contractors, incaricare un certo contractor, ricevere i risultati da quest'ultimo.
Il contractor invece deve: ricevere l'annuncio del compito, valutare se sia possibile svolgerlo, rispondere, eseguire il compito se lo ha accettato, fornire i risultati.
Ogni agente può comportarsi sia da manager che da contractor. Infatti un certo contractor potrebbe fungere da manager andando a sua volta a richiedere di eseguire un certo sottotask.

Un modello alternativo a quello visto finora è quello del *sistema a lavagna* in cui esiste una base di conoscenza comune, la lavagna su cui operano gli agenti. Ogni agente aggiorna la lavagna fornendo parte della soluzione.

12/04/2017

12 Semantica sociale dell'ACL

La definizione di semantica nell'ACL è basata sugli stati mentali. Questo perché una comunicazione viene definita in termini degli stati mentali dei partecipanti.
In un sistema di agenti cooperativi questo approccio va bene ma se gli agenti devono competere tra loro questo approccio non funziona perché non è possibile fidarsi del tutto degli altri agenti.
In questi ambienti usiamo un altro approccio, detto *approccio sociale*, in cui si

riconosce la natura inerentemente pubblica della comunicazione e comunichiamo tramite *commitments*. Basandosi sul fatto che agenti diversi giocano ruoli diversi in un sistema multiagente, ogni agente esegue un commit rendendo vere certe condizioni o eseguendo certe azioni

Esempio 1 : consideriamo una versione semplificata del protocollo NetBill per acquistare prodotti su internet. Un acquirente richiede un certo bene, un venditore invia una propria proposta, l'acquirente accetta o meno, il venditore invia i beni, l'acquirente invia il pagamento, il venditore invia una ricevuta

Di fatto come rappresentare i commitments quindi? Tramite i *fluenti*.
Esistono diversi tipi di fluenti

- commitment basilari $C(ag1, ag2, p)$: indicano la comunicazione tra i due agenti $ag1$ e $ag2$
- commitment condizionali $CC(ag1, ag2, p, q)$: se la condizione p è verificata, $ag1$ informerà $ag2$ riguardo q

Esempio 2 : per specificare meglio il protocollo NetBill, possiamo specificare dei fluenti del tipo request, paid... e delle zioni come request-quote, send-accept... Vediamo come viene implementata la send-accept:

```
Action{
   send-accept(ct,mr),
      PRECOND: NOT replied AND CC(mr, ct, accept, goods)
      EFFECT: replied AND accept AND CC(ct, mr, goods, paid) }
```

In generale quindi, l'aggiunta di commit è causata dall'esecuzione delle azioni. Quando tutti i commitment sono *fullfilled* tramite una sequenza finita di azioni, allora abbiamo un *run* del protocollo. Specificate le azioni è possibile definire dei piani che permettano di raggiungere uno stato in cui non ci sono commitment pendenti.

Per essere sicuri che un protocollo soddisfi le proprietà che ci aspettiamo, definiamo delle

- Proprietà strutturali(terminazione, deadlock): verificabili tramite un automa che rappresenti il protocollo
- Proprietà semantiche(rispetto di certi vincoli): verificabili ad esempio tramite model checking se le proprietà sono espresse in termini di formule logiche temporali

19/04/2017

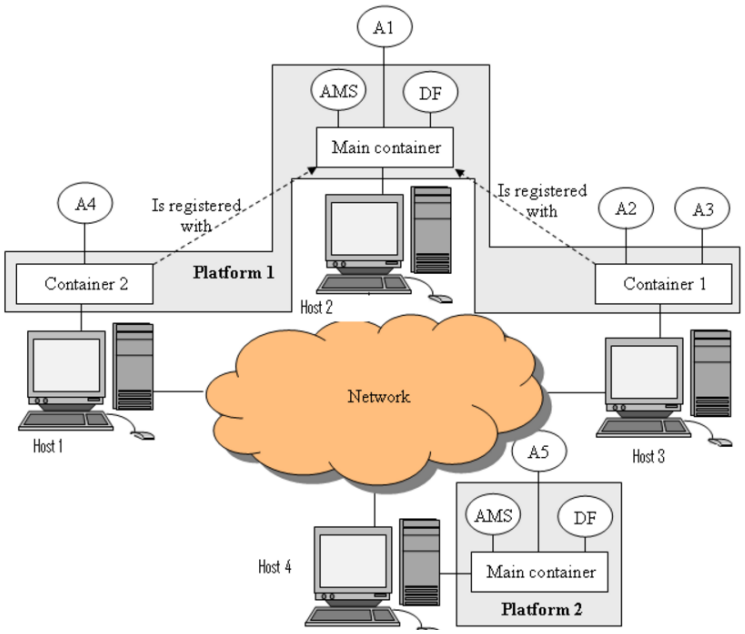

Figura 13: Due piattaforme JADE. Ogni piattaforma possiede uno o più contenitori e un main container

13 JADE

JADE è un linguaggio di programmazione basato su Java, sviluppato dalla FIPA (Foundation for Intelligent Physical Agents). Lo scopo della FIPA è determinare come gli agenti interagiscano fra loro ma non come gli agenti siano fatti(reattivi, BDI...)
All'interno di JADE viene definita la struttura dei messaggi scambiati, protocolli per i messaggi. L'obiettivo di JADE è quello di fornire un ambiente in cui far interagire gli agenti.
Gli agenti sono collocati all'interno di una piattaforma che è costituita da *conteiners*. Ogni contenitore può avere associati uno o più agenti. Ogni piattaforma deve possedere un *main-container*. Mostriamo questi elementi in Figura 13

Una volta scaricato JADE[4] possiamo avviarlo tramite la stringa

```
java -cp lib\jade.jar jade.Boot -gui
```

Avendo inserito l'opzione gui si avvierà l'interfaccia grafica mostrata in Figura 14.
Notiamo che, andando ad ossevare il contenuto del main container abbiamo tre

[4]Scaricabile presso il sito ufficiale
http://jade.tilab.com/download/jade/license/jade-download/?x=58&y=20

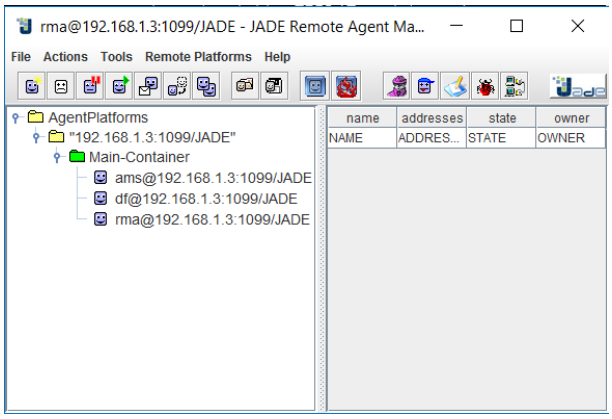

Figura 14: La struttura della GUI di JADE e i tre agenti di default

agenti presenti in ogni programma JADE. Uno gestisce il main container, un altro gestisce gli agenti mentre l'ultimo la comunicazione tra agenti. Mostriamo in tre agenti di default in Figura 14.

Vediamo adesso come implementare un agente in JADE. Come abbiamo detto, l'agente è sviluppato in Java e infatti gli agenti sono espressi tramite una classe che estende la classe di default Agent. Questa classe mette a disposizione alcuni metodi come `getLocalName()` che fornisce il nome dell'agente oppure `doDelete()` che causa la terminazione dell'agente. Vediamo qualche semplice esempio di agenti in JADE.

Esempio 1 : per rispettare la tradizione, cominciamo con il classico programma hello world. Il codice JADE è

```
import jade.core.Agent;
public class HelloWorldAgent extends Agent {
   protected void setup() {
      System.out.println("Hello World! My name is "+getLocalName());
      doDelete();      // Make this agent terminate
   }
}
```

Il metodo setup è quello che viene avviato all'inizio della computazione. Come dicevamo, il metodo getLocalName() è ereditato dalla classe Agent.

20/04/2017

13.1 Behaviour

Per definire i compiti che un certo agente deve portare a termine, dobbiamo parlare dei *behaviour*.
Un behaviour rappresenta un compito che l'agente deve portare a termine.

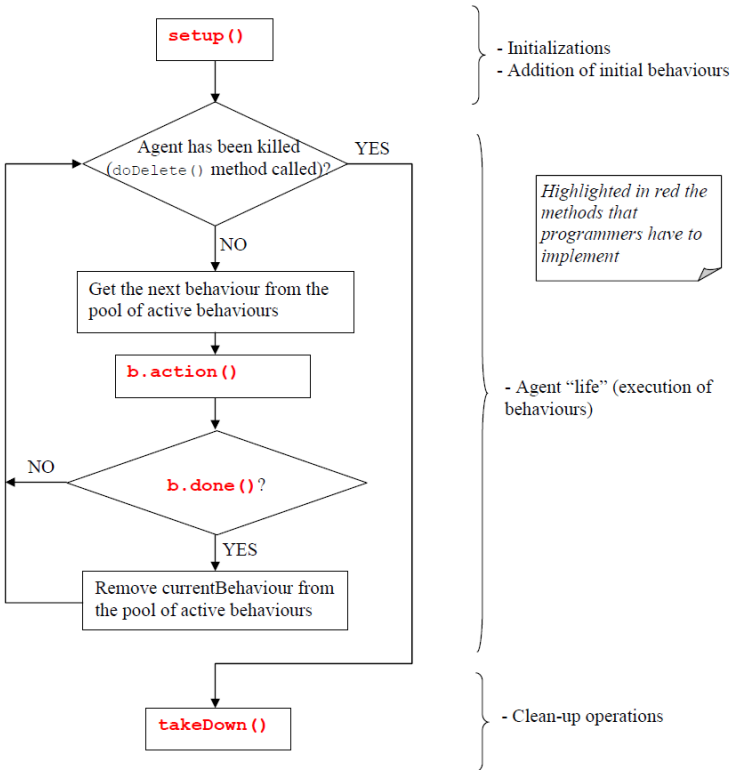

Figura 15: Diagramma di flusso del thread di un agente

Per aggiungere un behaviour ad un certo agente dobbiamo utilizzare il metodo `addBehaviour()` della classe Agent.
Inoltre ogni classe che estenda la classe Behaviour deve obbligatoriamente implementare i metodi

- `action()`: definisce le operazioni da eseguire quando il comportamento è in esecuzione
- `done()`: restituisce un booleano che specifica se il comportamento ha raggiunto il suo scopo e dunque deve essere rimosso dal pool dei comportamenti.

Il thread che esegue le azioni dell'agente segue un ciclo predefinito che mostriamo in Figura 15

Notiamo che abbiamo una fase di iniziallizzazione in cui viene invocato il metodo `setup()`. Dopo verifichiamo se sia necessario interrompere la computazione dell'agente, nel caso sia stato invocato un metodo `doDelete()`. Nel caso in cui l'agente debba continuare ad operare, prende il behaviour successivo dal pool dei behaviour e lo esegue. Viene invocata la `done()` e verifichiamo se il

behaviour ha raggiunto il suo obiettivo. Se l'obiettivo viene raggiunto possiamo rimuovere il behaviour dal pool, altrimenti lo manteniamo all'interno del pool. Vediamo un esempio in cui implementiamo un agente che riceva un ping e invii una risposta

Esempio 1 : il codice dell'agente è

```
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;
import jade.util.Logger;
public class PingAgent extends Agent {
   private Logger myLogger =
       Logger.getMyLogger(getClass().getName());
   private class WaitPingAndReplyBehaviour extends CyclicBehaviour {
      public WaitPingAndReplyBehaviour(Agent a){
         super(a);
      }
      public void action() {
         ACLMessage msg = myAgent.receive();
         if(msg != null){
            ACLMessage reply = msg.createReply();
               if(msg.getPerformative()== ACLMessage.REQUEST){
                  String content = msg.getContent();
               if ((content != null) && (content.indexOf("ping") !=
                   -1)){
                  myLogger.log(Logger.INFO, "Agent
                       "+getLocalName()+" - Received PING Request
                       from "+msg.getSender().getLocalName());
                  reply.setPerformative(ACLMessage.INFORM);
                  reply.setContent("pong");
               }
               else{
                  myLogger.log(Logger.INFO, "Agent
                       "+getLocalName()+" - Unexpected request
                       ["+content+"] received from
                       "+msg.getSender().getLocalName());
                  reply.setPerformative(ACLMessage.REFUSE);
                  reply.setContent("( UnexpectedContent
                       ("+content+"))");
               }
            }
            else {
               myLogger.log(Logger.INFO, "Agent "+getLocalName()+" -
                    Unexpected message
```

```
                    ["+ACLMessage.getPerformative(msg.getPerformative())+"]
                    received from "+msg.getSender().getLocalName());
                reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
                reply.setContent("( (Unexpected-act
                    "+ACLMessage.getPerformative(msg.getPerformative())+")
                    )");
             }
             send(reply);
          }
          else {block();}
       }
    } // END of inner class WaitPingAndReplyBehaviour

    protected void setup() {
       DFAgentDescription dfd = new DFAgentDescription();
       ServiceDescription sd = new ServiceDescription();
       sd.setType("PingAgent");
       sd.setName(getName());
       sd.setOwnership("TILAB");
       dfd.setName(getAID());
       dfd.addServices(sd);
       try {
          DFService.register(this,dfd);
          WaitPingAndReplyBehaviour PingBehaviour = new
              WaitPingAndReplyBehaviour(this);
          addBehaviour(PingBehaviour);
       } catch (FIPAException e) {
          myLogger.log(Logger.SEVERE, "Agent "+getLocalName()+" -
              Cannot register with DF", e);
          doDelete();
       }
    }
}
```

Esiste una particolare categoria di behaviour, detta dei *behaviour ciclici* a cui appartengono tutti quei behaviour che non terminano, dunque sono sempre presenti all'interno del pool.

26/04/2017

13.2 Comunicazione fra agenti

Lo scambio di messaggi tra agenti in JADE è asincrono.
Ogni agente possiede una propria coda di messaggi. Quando un agente riceve un messaggio, viene inserito in fondo alla coda. Questo schema viene mostrato in Figura 16

I messaggi che vengono scambiati fra gli agenti hanno un preciso formato stabilito dalla FIPA.
Nello specifico, i messaggi scambiati possiedono i seguenti attributi

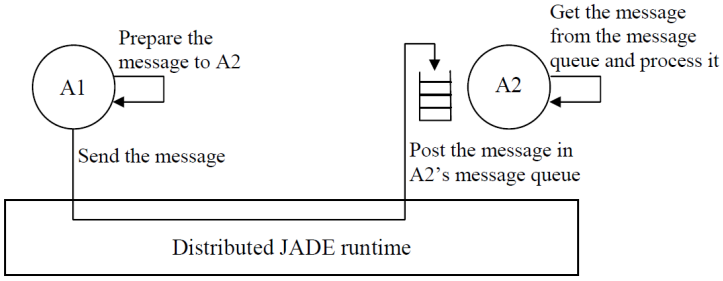

Figura 16: Struttura della coda dei messaggi di un agente

- Il mittente
- La lista dei destinatari
- La *performative*, ovvero lo scopo del messaggio. Nello specifico la performative può essere di 7 tipi diversi: REQUEST quando il mittente vuole che il destinatario faccia qualcosa, INFORM se il mittente vuole informare il destinatario di qualcosa, QUERY_IF quando il mittente vuole sapere se una certa condizione sia ancora valida, e infine il set di performative CFP(call for proposal), PROPOSE, ACCEPT_PROPOSAL, REJECT_PROPOSAL se mittente e destinatario devono negoziare qualcosa
- Il contenuto
- Il linguaggio del contenuto, per specificare ai destinatari quale sia la codifica del messaggio
- L'ontologia, ovvero il vocabolario dei simboli usati nel campo contenuto
- Alcuni altri campi per specificare ulteriori opzioni, come i timeout per ricevere una risposta

Vediamo un esempio di semplice invio di un messaggio

Esempio 1 : vediamo un esempio di invio di informazione tra agenti

```
package saluti;
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.Behaviour;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
public class Agente1_2 extends Agent {
    protected void setup() {
      System.out.println("Buongiorno, mi presento sono " +
          getName() +
```

```
        " e sono pronto ad operare!");
        addBehaviour(new Behaviour() {
           private int step = 0;
           public void action() {
              switch(step) {
                 case 0:
                    ACLMessage msg0 = new
                        ACLMessage(ACLMessage.QUERY_IF);
                    msg0.addReceiver(new AID("AgenteDueDue",
                        AID.ISLOCALNAME));
                    msg0.setLanguage("Italian");
                    msg0.setContent("Buongiorno, come sta?");
                    send(msg0);
                    System.out.println(getName() + ": inviata
                        QUERY_IF");
                    step++;
                    break;
                 case 1:
                    MessageTemplate mt0 =
                       MessageTemplate.MatchPerformative(ACLMessage.INFORM_IF);
                    ACLMessage reply0 = receive(mt0);
                         if (reply0 != null) {
                             if (reply0.getContent().equals("Bene,
                                  grazie. E lei?")) {
                                 System.out.println(getName() + ":
                                     rivevuta
                                  INFORM_IF");
                                 ACLMessage msg1 =
                                     reply0.createReply();
                                 msg1.setPerformative(ACLMessage.INFORM);
                                 msg1.setContent("Bene, grazie.");
                                 send(msg1);
                                 System.out.println(getName() + ":
                                     inviata INFORM");
                                 step++;
                             }
                         } else {
                             block();
                         }
                    break;
                 }
                  }
              public boolean done() {return step == 2;}
          });
       }
    }
```

Notiamo che per ricevere un messaggio dobbiamo utilizzare la `receive()`. Cosa succede se la coda è vuota al momento dell'invocazione? Semplicemente la

`receive()` restituisce NULL.
Come fare a ricevere un messaggio se non sappiamo quando il messaggio verrà inviato? Tecnicamente potremmo inserire la receive in un while, ma questo comporterebbe un notevole spreco di risorse. Per questo motivo in realtà usiamo un metodo particolare che ha nome `block()`. Invocando block, l'agente si sospende fino a quando non riceve un nuovo messaggio. In generale una buona pratica programmativa è la seguente

```
public void action() {
   ACLMessage msg = myAgent.receive();
   if (msg != null) {
      // Messaggio ricevuto correttamente. E' possibile processarlo
      ...
   }
   else {
      block();
   }
}
```

Vediamo un esempio con scambi di messaggi multipli

Esempio 2 : vediamo uno scambio di saluti fra due agenti

```
package saluti;
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.Behaviour;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
public class Agente2_2 extends Agent {
    protected void setup() {
        System.out.println("Buongiorno, mi presento sono " +
            getName() +
            " e sono pronto ad operare!");
        addBehaviour(new Behaviour() {
            private int step = 0;
            public void action() {
                switch(step) {
                    case 0:
                        MessageTemplate mt0 =
                            MessageTemplate.MatchPerformative(ACLMessage.QUERY_IF);
                        ACLMessage msg0 = receive(mt0);
                        if (msg0 != null) {
                            if (msg0.getContent().equals("Buongiorno,
                                come sta?")) {
                                System.out.println(getName() + ":
                                    rivevuta QUERY_IF");
                                ACLMessage reply0 =
                                    msg0.createReply();
                                reply0.setPerformative(ACLMessage.INFORM_IF);
```

```
                        reply0.setContent("Bene, grazie. E
                            lei?");
                        send(reply0);
                        System.out.println(getName() + ":
                            inviata
                         INFORM_IF");
                        step++;
                    }
                }
                else{
                    block();
                }
                break;
            case 1:
                MessageTemplate mt1 =
                   MessageTemplate.MatchPerformative(ACLMessage.INFORM);
                ACLMessage msg1 = receive(mt1);
                if (msg1 != null) {
                    if (msg1.getContent().equals("Bene,
                         grazie.")) {
                        System.out.println(getName() + ":
                            rivevuta INFORM");
                        step++;
                    }
                }
                else{
                    block();
                }
                break;
        }
      }
         public boolean done() {return step == 2;}
    });
  }
}
```

27/04/2017

14 Interazione multiagente: risorse scarse

L'allocazione di risorse scarse è un tema centrale nella programmazione multiagente. Le risorse possono essere di vario tipo: oggetti fisici, risorse computazionali... Nel momento in cui le risorse a disposizione sono poche, è necessario fare un'*asta* per assegnarle.
In un'asta abbiamo dei compratori che cercano di ottenere un certo bene pagando una certa quantità di soldi. Ogni compratore ha un proprio *prezzo limite*. Se lo eccede, commette una *perdita*. A sua volta, un venditore che vende un bene

a meno del valore limite dei compratori commette una perdita.
Ovviamente il prezzo limite influenza il comportamento dei compratori. Esistono tre distinti modelli per valutare come il prezzo limite influenzi il comportamento degli agenti

- A valore privato: il valore che attribuisco all'oggetto è indipendente dal valore che gli danno gli altri compratori
- A valore comune: il valore che attribuisco all'oggetto è lo stesso per tutti ma viene stimato differentemente dai vari agenti. Un esempio potrebbe essere la vendita di un vaso pieno di monete. Il valore sarà meramente lo stesso per tutti ma ogni agente tenderà a valutarlo diversamente
- A valore correlato: il valore che diamo all'oggetto è legato al valore che gli altri danno all'oggetto. Se qualcuno è disposto a pagare di più per quell'oggetto, probabilmente anche noi saremo maggiormente propensi ad aumentarne il valore

Diremo che se un compratore propone un prezzo, abbiamo un'*offerta*. Nel caso in cui sia un venditore a indicare un prezzo, la chiameremo *richiesta*. Secondo la definizione di Dan Friedman, un'asta è un istituzione di mercato in cui vengono scambiati dei messaggi inerenti il prezzo di un certo bene. In questo scambio di messaggi viene data la preferenza alle offerta a maggior prezzo, e alle richieste a minor prezzo.
Possiamo dividere le aste in:

- Monodimensionali/multidimensionali: monodimensionali se il contenuto di un'offerta riguarda semplicemente un bene e la quantità da comprarne. Multidimensionale se include altri aspetti, come il tempo di consegna...
- Unilaterali/bilaterali: unilaterali se abbiamo un venditore e molti compratori oppure un compratore e molti venditori. Bilaterali se possiamo avere molti compratori e venditori allo stesso tempo
- Ad offerta aperta/chiusa: aperta se le offerte che vengono fatte sono note a tutti. Chiusa se solo il venditore conosce le offerte
- Ad unità singola/multipla: singola se esiste un solo esemplare dell'oggetto o se comunque i vari esemplari vengono venduti uno alla volta. Multipla se oltre al prezzo viene fornita anche una quantità
- Primo prezzo/k-esimo prezzo: primo prezzo se il vincitore paga il prezzo più alto. k-esimo prezzo se paga il k-esimo prezzo in ordine di valore
- Ad oggetto singolo/multioggetto: singolo se esiste un solo esemplare indivisibile che viene venduto. Multioggetto se viene venduto un "pacchetto" unico contenente beni diversi

Vediamo invece quattro diversi tipi di asta:

1. Asta inglese: è il tipo di asta nota a tutti. Vengono fatte delle offerte, partendo da un prezzo di base. Termina dopo un certo tempo o quando non ci sono più offerte. Chi vince paga il prezzo della sua offerta. Secondo le caratteristiche che abbiamo dato prima questo tipo di asta è: monodimensionale, unilaterale, ad offerta aperta, ad unità singola, primo prezzo, ad oggetto singolo

2. Asta olandese: l'oggetto parte da un prezzo alto. Il venditore abbassa periodicamente il prezzo. Il primo compratore che ritiene il prezzo accettabile lo compra per quella cifra segnalandolo al venditore.
   Esiste anche una versione che unisce le due aste appena viste ed è l'asta di Amsterdam. Comincia con un'asta inglese, ma quando restano solo due offerenti viene avviata un'asta olandese che parte dal doppio del prezzo a cui era giunta l'asta inglese.

3. Asta primo prezzo ad offerta chiusa: ogni possibile compratore fa un'offerta senza sapere cosa offrono gli altri. L'offerta più alta vince

4. Asta di Vickrey: è un'asta segreta in cui i vari offerenti presentano la propria offerta senza sapere cosa offrono gli altri.
   L'offerta vincente è la più alta ma chi vince paga il prezzo della seconda offerta migliore. Sembrerebbe che un meccanismo di questo tipo svantaggi il venditore ma è anche vero che gli agenti tenderanno ad alzare maggiormente la posta. Questo tipo di asta comunque spinge gli agenti ad offrire il valore che davvero vorrebbero pagare senza sovrastimarlo o sottostimarlo. Infatti se un compratore tende a sovrastimare rischia di avere una perdita consistente ma se invece tende a sottostimare diminuisce la sua probabilità di vincita.

02/05/2017

15 Interazione multiagente: teoria dei giochi

La teoria dei giochi non è può essere il tema centrale di un intero corso ma possiamo certamente parlarne. Infatti nella teoria dei giochi abbiamo un insieme di agenti che interagiscono per raggiungere un certo obiettivo.
Gli agenti hanno un loro obiettivo e devono stabilire quali siano le decisioni più razionali da eseguire. Le mosse sono eseguite simultaneamente e per semplificare il nostro modello considereremo giochi in cui viene eseguita una sola mossa.
Anche se queste semplificazioni potrebbero far sembrare la teoria dei giochi fin troppo semplice, in realtà può essere usata per risolvere situazioni molto complesse.
La teoria dei giochi può essere studiata in due modi diversi:

1. Design dell'agente: determiniamo le azioni che l'agente deve eseguire in base all'utilità delle azioni all'interno del gioco.

2. Design del meccanismo(del protocollo): dato un certo ambiente con più agenti, andiamo a determinare quale sia il gioco che debbano giocare gli agenti per massimizzare il bene comune

Notiamo che avendo solo due giocatori che possono eseguire una e una sola mossa, possiamo andare a definire una *payoff matrix* contenente il risultato della funzione di utilità per ogni giocatore in base alle mosse scelte. Vediamolo tramite qualche esempio

Esempio 1 : due giocatori P e D giocano a una versione modificata della morra in cui possono solo tirare una o due dita. Nel momento in cui la somma delle dita è pari, P riceve da D tanti dollari quante le dita coinvolte nel tiro. Se invece la somma delle dita è dispari, D riceve da P tanti dollari quante le dita coinvolte nel tiro. In pratica la payoff matrix è:

	D: uno	D:due
P:uno	$P = 2,\ D = -2$	$P = -3,\ D = 3$
P:due	$P = -3,\ D = 3$	$P = 4,\ D = -4$

Sulla prima riga possiamo vedere le mosse che può scegliere D mentre sulla prima colonna possiamo vedere le mosse che può scegliere P. Prendiamo ad esempio la mossa in cui D getta un dito mentre P li getta entrambi. In questo caso il risultato è dispari, quindi P fornisce a D tre dollari visto che la somma delle dita è tre.
In questo caso, non è possibile stabilire una strategia vincente, tutti i casi sono equiprobabili e il valore medio del guadagno è pari a zero.

Esempio 2 : due scassinatori, Alice e Bob vengono catturati in possesso della refurtiva. Interrogati separatamente gli vengono date le seguenti informazioni

- Se entrambi non si accusano, la loro pena sarà di un anno
- Se entrambi si accusano, la pena sarà di cinque anni per entrambi
- Se uno solo accusa l'altro, l'accusatore sarà libero ma l'accusato riceverà dieci anni di pena

Sapendo che sia Alice che Bob cercheranno di minimizzare il numero di anni di carcere senza avere a cuore il destino dell'altro, quale potrebbe essere una strategia efficace per ognuno di loro?
La payoff matrix è

	Alice: accusa	Alice: non accusa
Bob: accusa	$A = -5,\ B = -5$	$A = -10,\ B = 0$
Bob: non accusa	$A = 0,\ B = -10$	$A = -1,\ B = -1$

In questo caso, accusare l'altro è la strategia migliore. Consideriamo il punto di vista di Alice. Nel caso Bob accusi, entrambi ricevono una pena pari a cinque anni. Se Alice non avesse confessato, avrebbe ricevuto una pena pari a 10 anni.
Consideriamo il caso in cui Bob non accusi. In questo caso Alice può uscire subito. Se invece non avesse accusato, avrebbe ricevuto un anno.
Dunque, qualunque cosa scelga Bob, se Alice lo accusa ottiene un risultato migliore.

Una particolarità dell'esempio 2 è che è possibile delineare una *strategia fortemente dominante* per il gioco.
Una strategia s viene detta fortemente dominante su ogni altra strategia s' se il risultato di s è migliore del risultato prodotto da s' per ogni possibile mossa eseguibile da ogni altro giocatore.
Talvolta, una strategia dominante non esiste(è il caso dell'esempio 1).
Ad ogni modo, la strategia scelta da Alice non è invece *Pareto ottimale*.
Una strategia viene detta Pareto ottimale se non esiste un altro risultato in cui un qualche giocatore ottiene un risultato migliore senza far peggiorare il risultato di un altro.
Riprendendo l'esempio 2 è la soluzione in cui nessuno accusa la Pareto ottimale, infatti non esiste un'altra situazione in cui uno dei due scassinatori possa ridurre la propria pena senza aumentare quella del proprio compare. Potrebbero pensare di applicare un algoritmo che porti ad una soluzione Pareto ottimale entrambi? Certamente, ma solitamente questo non accade perché la soluzione in cui entrambi non testimoniano non è un *punto di equilibrio*.
Diremo infatti che date due strategie di due giocatori diversi, $S = (s1, s2)$ è un punto di equilibrio se, fissata la strategia $s1$, l'altro giocatore non può fare meglio che scegliere $s2$ e viceversa.

03/05/2017

Quanti punti di equilibrio possono esistere all'interno di un gioco? Possono anche non esisterne? La risposta è sì. Ad esempio nel gioco della morra non esiste alcun punto di equilibrio. Per dimostrarlo dobbiamo verificare che ogni stato possibile non è un punto di equilibrio.
Avere un punto di equilibrio è chiaramente molto importante per stabilire una strategia.
In generale, l'esistenza di una strategia dominante determina l'esistenza del punto di equilibrio. Non è invece vero che se esistono dei punti di equilibrio, esista anche una strategia dominante. Vediamolo tramite un esempio

Esempio 3 : supponiamo che esista una casa di produzione A di console e una di sviluppo di videogiochi B. Entrambe devono decidere se supportare/utilizzare i cd o i dvd. Anche in questo caso ogni azienda non sa cosa sceglierà l'altra. La payoff matrix è:

In questo caso non esiste una strategia dominante. Se B sceglie di utilizzare i DVD, A avrà un guadagno positivo scegliendo i DVD e negativo se sceglie

	A: DVD	A: CD
B: DVD	$A = 9,\ B = 9$	$A = -4,\ B = -1$
B: CD	$A = -3,\ B = -1$	$A = 5,\ B = 5$

i CD. Viceversa però, se B sceglie i CD, i risultati si invertono. Dunque non c'è una scelta migliore a prescindere.
Ad ogni modo esistono due punti di equilibrio, quello in cui entrambi scelgono di utilizzare lo stesso supporto(o tutti e due CD o tutti e due DVD).

Quando esistono due punti di equilibrio come nell'esempio precedente, come fare a sceglierne uno? In generale possiamo usare come secondo criterio l'ottimalità secondo Pareto. Ad ogni modo se i due punti di equilibrio presentano lo stesso guadagno il criterio di ottimalità secondo Pareto non ci aiuta.

Consideriamo adesso un ulteriore elemento: quale è la probabilità che una certa strategia venga selezionata da un altro agente? Considerare la probabilità ci sposta nel territorio delle strategie miste. Vediamolo tramite un esempio

Esempio 4 : consideriamo la situazione in cui un calciatore debba tirare un rigore e un portiere debba pararlo. Le scelte sono solo se tirare a destra/-sinistra per il calciatore, e gettarsi a destra/sinistra per il portiere. La payoff matrix è:

	A: destra	A: sinistra
B: destra	$A = -1,\ B = 1$	$A = 1,\ B = -1$
B: sinistra	$A = 1,\ B = -1$	$A = -1,\ B = 1$

Quale strategia è meglio applicare? Associando una certa probabilità ad ogni azione emerge un risultato molto intuitivo. È meglio tirare sia a destra che a sinistra con probabilità $p = 0.5$. Questo ci permette di avere un punto di equilibrio nella strategia

In generale comunque è ancora un problema aperto come trovare un punto di equilibrio data una strategia mista

Un altro problema generale nella teoria dei giochi è se cooperare o meno. Considerando il caso dei prigionieri, la cooperazione sarebbe consigliabile perché porta ad un miglioramento della condizione di entrambi. Purtroppo cooperare comporta dei rischi, anche nel mondo reale. Ad esempio il problema del disarmo atomico è un problema di cooperazione che comporta dei rischi, dovrebbe essere applicato da entrambe le parti che voglio ridurre il proprio armamento, ma se una delle due non procede al disarmo...
Chiaramente questo discorso sussiste fintanto che entrambi gli agenti cercano

di massimizzare il proprio guadagno senza tener conto delle perdite dell'altro agente.
Cosa succederebbe se per esempio un nostro agente venisse a conoscenza di dover incontrare più volte un altro agente? Dovremmo tener di conto che l'altro agente possa considerare come il nostro agente si è comportato precedentemente. Questa situazione è stata studiata matematicamente e ne è emerso che:

- Se un agente ne incontra un altro ∞ volte allora è sempre meglio cooperare, infatti persistere in un comportamento di non cooperazione comporta una diminuzione del guadagno di tutti
- Se un agente ne incontra un altro n volte con n finito, allora non conviene cooperare. Questo succede perché con n finito ogni agente potrebbe essere tentato di non cooperare sull'ultima interazione con l'altro agente. Questa scelta servirebbe per massimizzare il proprio guadagno. 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 a un torneo per stabilire quale fosse la strategia migliore relativa al problema degli scassinatori. Dopo aver sviluppato diverse strategie ha fatto interagire ogni strategia con le altre in "match" differenti. Ogni match era costituito da 200 round. Ne è risultato che la strategia migliore 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.

04/05/2017

16 Interazione multiagente: negoziazione

Visto che la cooperazione è tanto importante, come è possibile far raggiungere un accordo a due agenti che hanno obiettivi diversi?
In generale la negoziazione deve essere guidata da dei protocolli. Nelle aste per esempio c'è un protocollo ben preciso: il banditore avvia l'asta, qualcuno fa un'offerta, il banditore la accetta e così via.
Un buon protocollo di negoziazione dovrebbe avere le seguenti proprietà

- Successo garantito: certamente viene raggiunto un accordo
- Massimizzazione dell'utilità globale
- Efficienza di Pareto: non esiste un altro risultato che permette a qualcuno di migliorare la propria situazione senza che quella di qualcun altro peggiori

- Razionalità individuale: seguire il protocollo è la cosa migliore che un agente possa fare
- Stabilità(equilibrio secondo Nash)
- Semplicità
- Distribuzione

Nel seguito definiremo un protocollo con le caratteristiche appena elencate in un dominio *task-oriented*. Un dominio task-oriented è formato da

$$< T,\ Ag,\ c >$$

dove:
T è l'insieme dei task
$Ag = \{1, \ldots, n\}$ è l'insieme degli agenti
$c : p(T) \to \mathbb{R}^+$ è il costo per eseguire una certo insieme di task

Diremo inoltre che un *encounter* è un insieme di task del tipo $< T_1, \ldots, T_n >$ per cui $\forall i \in Ag : T_i \subseteq T$
Restringendosi al caso più semplice in cui abbiamo solo due agenti, dato un encounter $< T_1, T_2 >$ diremo che un *deal* è una allocazione dei task $T_1 \bigcup T_2$ agli agenti 1 e 2.
Il costo di un deal $\delta =< D_1, D_2 >$ dove D_i è una possibile redistribuzione dei task per l'agente i è pari a $c(D_i)$ per l'agente i.
In questo senso è molto utile andare a definire una funzione di utilità per il deal. Viene definita come segue

$$utility_i(\delta) = c(T_i) - c(D_i)$$

Quando non si riesce a trovare un deal per gli agenti, siamo nella situazione di *conflict deal*. Questa situazione, solitamente indicata con Θ è quella in cui gli agenti smettono di cooperare tra loro. Notiamo che in questo caso

$$\forall i \in Ag,\ utility_i(\Theta) = 0$$

Esempio 1 : supponiamo di avere due agenti A e B. Ogni agente possiede due figli che frequentano 3 scuole diverse Sx, Sy, Sz. La posizione delle scuole è mostrata in Figura 17

I task eseguiti dagli agenti sono:
A_1: accompagna il figlio 1 di A nella scuola Sx
A_2: accompagna il figlio 2 di A alla scuola Sz
B_1: accompagna il figlio 1 di B alla scuola Sx
B_2: accompagna il figlio 2 di B alla scuola Sy

Sappiamo che spostarsi da un nodo all'altro costa 2, fermarsi in un nodo costa 1. Ad esempio $c(\{B_2\}) = 5$ e $c(\{A_1, B_1\}) = 5$.

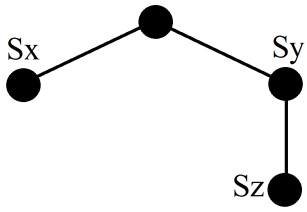

Figura 17: Posizione delle scuole dell'esempio 1

Nel caso di conflict deal, abbiamo che:
$c(\{A_1, A_2\}) = 14$, $c(\{B_1, B_2\}) = 10$
Consideriamo il deal $D :< \{A_1, B_1, A_2\}, \{B_2\} >$.
In questo caso i costi sono:
$c(\{A_1, B_1, A_2\}) = 14$, $c(\{B_2\}) = 5$
L'utility è:
$utility_A(D) = c(T_A) - c(D_A) = 0$
$utility_B(D) = c(T_B) - c(D_B) = 5$

In presenza di più deal possibili, diremo che un deal δ_1 *domina* un deal δ_2 se valgono le due seguenti proprietà:

1. Il deal δ_1 si comporta tanto bene quanto δ_2 su ogni agente, ovvero che:
   $\forall i \in Ag : utility_i(\delta_1) \geq utilty_i(\delta_2)$

2. Il deal δ_1 si comporta meglio di δ_2 per almeno un agente, ovvero che:
   $\exists i \in Ag : utility_i(\delta_1) > utilty_i(\delta_2)$

Indicheremo questa situazione tramite $\delta_1 \succ \delta_2$.
Diremo invece che δ_1 *domina debolmente* δ_2 nel caso in cui valga solo la prima delle due condizioni. Indicheremo questa situazione con $\delta_1 \succeq \delta_2$.
Nel seguito diremo inoltre che un deal δ è *individualmente razionale* se domina il conflict deal.
Diremo inoltre che un deal che non è dominato da nessun altro deal è *Pareto ottimale*. È su queste due categorie di deal che gli agenti effettuano i loro negoziati.
Nel precedente esempio gli unici due deal di questo tipo erano:
$D2 =< \{A2, B2\}, \{A1, B1\} >$ con $utility_A(D2) = 4$ e $utility_B(D2) = 5$
$D3 =< \{A1, B1\}, \{A2, B2\} >$ con $utility_A(D3) = 9$ e $utility_B(D3) = 0$

Vediamo adesso il protocollo. Il suo nome è *Protocollo di concessione monotona*. Le sue regole sono le seguenti: la negoziazione procede per round. Al round 1 ogni agente propone il suo deal per l'altro agente. Un agente accetta il deal quando crede che l'offerta dell'altro agente sia migliore o pari a(con utilità maggiore o uguale)quella da lui proposta.
Se la negoziazione prosegue al round successivo, un agente non può offrire all'altro un deal "peggiore" di quello che aveva fornito al round precedente. Per

questo motivo il protocollo viene detto monotono. Perché ad ogni round l'utilità dell'agente con cui stiamo negoziando deve seguire una funzione monotona crescente.
Se dopo un certo numero di round non si riesce a trovare un deal, la negoziazione termina e ogni agente prosegue con il proprio conflict deal.
Questo protocollo pone tre domande le cui risposte vanno sotto il nome di *strategia di Zeuthen*:

- Quale dovrebbe essere il primo deal proposto da un agente? Quello da lui preferito
- Ad ogni round chi dovrebbe concedere un miglioramento dell'utilità dell'altro agente? L'agente che corre un maggiore rischio, ovvero quello che potrebbe rimetterci maggiormente se ottenesse un conflict deal
- Nel caso un agente debba concedere, quando deve concedere? Il minimo indispensabile per ribaltare la situazione di rischio

In generale un certo agente tenderà a rischiare un conflict deal quanto meno sta guadagnando dalla situazione attuale. Più formalmente diremo che il *rischio* per un certo agente i è pari a:

$$risk_i = \frac{utility_i(\delta_i) - utility_i(\delta_j)}{utility_i(\delta_i)}$$

In modo un po' controintuitivo, un alto valore di rischio indica poco da perdere nel conflitto. Viceversa un basso valore di rischio indica che un certo agente ha molto da perdere. Notiamo che infatti se l'altro agente ci offre un deal con il quale non guadagniamo niente($utility_i(\delta_j) = 0$) allora il nostro rischio è pari a 1, ovvero non abbiamo niente da perdere.
Notiamo infine che la strategia di Zeuthen è in equilibrio secondo Nash: se un agente decide di applicarla l'altro agente non può fare di meglio che usarla a sua volta.

09/05/2017

17 JASON

Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net(JASON) è un linguaggio di programmazione basato sull'architettura PRS di cui abbiamo parlato nella sottosezione 5.1. Di fatto JASON è una semplificazione del PRS.
Per poter parlare di JASON dobbiamo però riprendere AgentSpeak cui avevamo accennato nella sottosezione 7.1, un linguaggio sviluppato nel 1996 che si propone di colmare la lacuna formatasi tra teoria e implementazione degli agenti BDI.
JASON è di fatto la prima implementazione di AgentSpeak in Java.
Dobbiamo quindi parlare un po' di AgentSpeak per poter procedere. AgentSpeak specifica il comportamento di un agente inclusi i beliefs i desires e le

intentions tipiche del modello BDI.

In AgentSpeak i concetti atomici sono:

- beliefs: che corrispondono ai fatti della programmazione logica.
- piani: dipendenti dal contesto, invocabili dall'utente, consentono di decomporre i goal in modo gerarchico e sono sintatticamente simili alle clausole della programmazione logica
- azioni
- intentions
- eventi
- goal

All'interno di AgentSpeak possiamo usare variabili, costanti, simboli funzionali, simboli predicativi, simboli di azione, connettivi, quantificatori, simboli di punteggiatura.
I connettivi possono essere quelli della logica classica(and, or, not...) più tre connettivi speciali

- ! connettivo di achievement
- ? connettivo di test
- ; connettivo di sequenza

Un altro elemento in comune con i linguaggi di programmazione logica è che le variabili del linguaggio vengono indicate con la lettera maiuscola.

Studiamo le componenti appena elencate in maggior dettaglio partendo dai beliefs.
Per rappresentarli usiamo la seguente notazione:

$$b(t_1, \ldots, t_n)$$

è un belief atom solitamente indicato con $b(t)$, dove b è un simbolo predicativo e $t_1, \ldots, t_n$ sono termini.
Quando un certo belief atom non possiede variabili libere, è detto *belief atom ground*. Nel caso in cui vi siano più beliefs atom legati da un connettivo, vengono chiamati semplicemente beliefs. Istanze ground di belief sono dette *base beliefs*
Vediamo cosa rappresentare tramite i beliefs in un esempio

Esempio 1 : supponiamo di simulare un autostrada. Nell' autostrada abbiamo: 4 corsie adiacenti. Inoltre sappiamo che le corsie possono essere percorse da automobili, in ciascuna corsia può essere presente della spazzatura. Il robot deve raccogliere la spazzatura e depositarla nel cestino,

posizionato in una delle 4 corsie, e infine che mentre pulisce, il robot non deve trovarsi in una corsia in cui è presente un'auto, per evitare di essere distrutto. Tramite i beliefs potremmo rappresentare la configurazione delle corsie, il posizionamento del robot, la posizione delle auto nelle corsie, e la posizione del cestino per la raccolta dei rifiuti. Ad esempio potremmo avere che

adjacent(X,Y)
location(robot,X)
location(car,X)

Andando a scrivere adjacent(a,b), location(robot,c) creiamo delle base beliefs

Il goal in AgentSpeak è uno stato del sistema che l'agente vuole raggiungere. Esistono due tipi di goal:

- Achievement goal: della forma !g(t), sono quei goal che l'agente vuole raggiungere
- Test goal: della forma ?g(t), sono quei goal che l'agente vuole verificare se sono veri

Vediamo invece i *triggering events*. Goal e beliefs possono essere aggiunti soltanto quando vengono attivati alcuni eventi speciali detti triggering events. Abbiamo quattro possibili triggering events:

1. Aggiunta di un belief
2. Rimozione di un belief
3. Aggiunta di un goal
4. Rimozione di un goal

I triggering events sono importanti perché possono attivare condizioni che modifichino il comportamento dell'agente o l'ambiente stesso. Le aggiunte e le rimozioni di goal/belief vengono segnalate rispettivamente dai simboli + e −
Per essere più precisi diremo che dati un belief b(t) e i due goal !g(t) e ?g(t) allora +b(t), -b(t), +!g(t), -!g(t), +?g(t), -?g(t) sono triggering events.

Esempio 2 : riprendendo quanto detto nell'esempio precedente, potremmo dire che:

!cleared(b) : l'agente vuole pulire la corsia b
?location(car,b): l'agente vuole verificare l'eventuale presenza di un auto nella corsia b.
Potremmo segnalare tramite un triggering event la presenza di spazzatura in un certa corsia

+location(waste, X)

Oppure l'acquisizione di un goal per ripulire una certa corsia:

+!cleared(X)

Ciò che modifica l'ambiente sono le *azioni*. Come per i goal e i beliefs, anche le azioni sono della forma $a(t_1, \ldots, t_n)$.
I *piani* servono invece per stabilire come un agente potrebbe raggiungere un determinato obiettivo. Solitamente un piano è strutturato come segue:

$$piano := head < -corpo$$

dove corpo è l'insieme di azioni da eseguire per raggiungere l'obiettivo, mentre head è

$$head := triggering\ event : contesto$$

In particolare il triggering event specifica come mai sia stato attivato il piano e il contesto specifica quali beliefs devono essere soddisfatti nel set delle credenze dell'agente quando il piano viene attivato.

Esempio 3 : volendo realizzare un piano che si attiva quando la spazzatura si trova in una certa corsia, e il robot è in tale corsia. L'obiettivo del piano è quello di portare la spazzatura al cestino. Un possibile piano potrebbe essere

```
+location(waste, X):
   location(robot,X) & location(bin,Y)
   <-pick(waste);
   !location(robot, Y);
   drop(waste).
```

Se invece volessimo realizzare un piano per consentire al robot di spostarsi tra le corsie

```
+!location(robot,X):
   location(robot,X)
   <-true.
```

Vediamo invece un po' di semantica operazionale di AgentSpeak(L). A run time un agente può essere visto come un insieme di beliefs, piani, intenzioni, eventi, azioni e funzioni di selezione.
Quando viene generato un triggering event. Gli eventi possono essere *esterni* se modificano l'ambiente o *interni* se modificano lo stato dell'agente. I piani attivabili a seguito di un evento sono detti *piani rilevanti*. Il piano da eseguire viene scelto tra quelli rilevanti dalla funzione di selezione

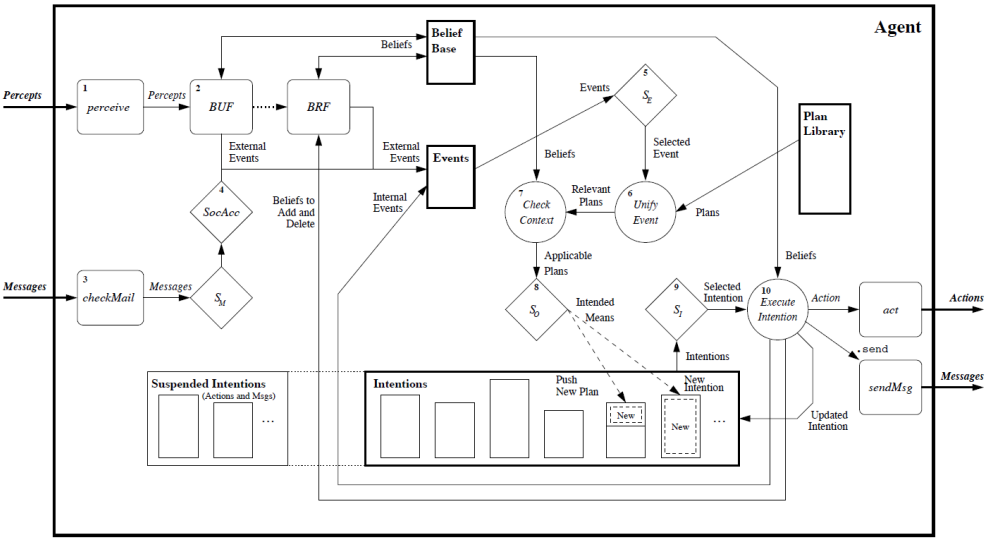

Figura 18: Ciclo di esecuzione di Jason

10/05/2017

Veniamo invece all'implementazione di AgentSpeak(L), ovvero a JASON. Interpreta AgentSpeak(L) aggiungendo alcune funzionalità come la gestione del fallimento di piani, il supporto per lo sviluppo di ambienti Java, l'aggiunta della negazione forte...
In modo simile ad AgentSpeak, in Jason abbiamo i piani, la cui struttura è

triggering_event : context <- body.

Il triggering event è l'evento che il piano deve gestire, il contesto indica le circostanze in cui il piano viene attivato, il body è la sequenza di azioni da effettuare per considerare il piano concluso.
Il ciclo di esecuzione di Jason, che mostriamo in è strutturato come segue:

1. Riceviamo una percezione dall'ambiente
2. La percezione passa al BUF (Belief Update Function), tramite cui aggiorniamo la base di conoscenza. L'aggiunta di fatti può scatenare la generazione di eventi
3. Riceviamo eventualmente dei messaggi da altri agenti, che quindi sono considerati da Jason come qualcosa di diverso rispetto alle percezioni. Vengono infatti gestiti tramite meccanismi diversi

4. Selezioniamo i messaggi socialmente accettabili tramite la componente S_M. L'arrivo di messaggi può scatenare la generazione di eventi

5. Selezioniamo uno degli eventi attivati tramite la componente S_E. Di default viene preso l'evento in testa alla coda degli eventi

6. Selezionato l'evento cerchiamo un piano relevant che abbia come triggering event proprio l'evento scelto. Questa ricerca viene fatta tramite un procedimento di unificazione tra i valori dell'evento e le variabili dei piani

7. Troviamo i piani applicabili, ovvero quelli che soddisfano il contesto

8. Selezioniamo uno dei piani trovati al punto precedente tramite la componente S_O, e lo inseriamo nelle intenzioni. Se il piano serve per gestire una nuova percezione, allora abbiamo una new intention, altrimenti se il piano è stato creato come sottogoal di un goal principale, creiamo uno stack di piani invocati ricorsivamente. Per capire in quale caso dei due precedenti ci troviamo, nella struttura degli eventi Jason è contenuto una *annotazione*, ovvero un campo che indica se l'evento è generato da una percezione o come sottogoal

9. Viene selezionata un'intenzione tramite la componente S_I. Di default la scelta viene fatta tramite round robin delle intenzioni, per fare in modo che la computazione di tutte le intenzioni proceda senza priorità particolari

10. Eseguiamo un passo dell'intenzione. Se abbiamo una azione la eseguiamo, se è un messaggio lo spediamo con .send, se è una modifica dei belief o un evento interno torniamo all'inizio del ciclo, se è la modifica di un'intenzione, torniamo alla scelta delle intenzioni

Come dicevamo, possiamo associare delle annotazioni ai predicati tramite parentesi quadre. Due esempi di annotazioni sono [percept] che indica la percezione di una informazione dall'ambiente e [self] che indica che l'informazione è stata aggiunta dall'agente stesso.
La gestione di un fallimento di un certo piano +!g avviene tramite la definizione di un piano -!g che viene invocato nel caso in cui +!g fallisca.
Le azioni interne in JASON sono definite sintatticamente da un punto che le precede. Ad esempio .send(), .print()...

In JASON un sistema multiagente viene configurato tramite un file specifico in cui indichiamo il nome della società di agenti cui gli agenti appartengono, i nomi degli agenti della società e l'ambiente in cui si collocano gli agenti(definito tramite una classe Java)