Agenti Intelligenti

Giacomo Grandi
A.A 2020-2021

5 trend che hanno caratterizzato la storia della computazione:

- Ubiquità
- Interconnessione
- Intelligenza
- Delega
- Human-orientation

UBIQUITÀ

La riduzione dei costi delle tecnologie ha permesso di introdurre capacità di elaborazione in luoghi e dispositivi sempre più comuni; elementi sofisticati e intelligenti diventano onnipresenti.

INTERCONNESSIONE

I sistemi informatici non esistono più da soli, ma collegati in rete in grandi sistemi distribuiti. Ad esempio internet. Proposti modelli teorici che ritraggono l'informatica come un processo di interazione.

INTELLIGENZA

La complessità dei compiti che siamo capaci di automatizzare e delegare ai computer è cresciuta costantemente.

DELEGA

I computer fanno di più per noi senza un nostro intervento. Stiamo dando il controllo ai computer anche in compiti critici per la sicurezza. In certi casi addirittura ci si può fidare più del giudizio di una macchina piuttosto che dell'esperienza del pilota (voli aerei ad esempio).

HUMAN-ORIENTATION

Spostare la programmazione (dal punto di vista espressivo) dalla macchina verso concetti più vicini al modo con cui noi umani comprendiamo il mondo.
I programmatori (e gli utenti) si relazionano ad una macchina in modo diverso.
I programmatori concettualizzano e realizzano software in termini di livello superiore (astrazione).

Delegazione e intelligenza implicano la necessità di costruire sistemi informatici in grado di agire in modo indipendente e che rappresentino nel migliore dei modi i nostri interessi.
Interconnessione e distribuzione accoppiate con la necessità di sistemi che rappresentino nel migliore dei modi i nostri interessi, portano a sistemi che possono cooperare e raggiungere accordi o competere con altri sistemi che hanno interessi diversi.

AGENTE
Sistema di computazione capace di agire in modo indipendente per conto di un utente o proprietario, capendo cosa dev'essere fatto per soddisfare gli obiettivi di progettazione.

SISTEMA MULTIAGENTE (MAS)
Serie di agenti che interagiscono l'uno con l'altro. Solitamente gli agenti agiscono per conto di utenti con differenti obiettivi e motivazioni. Richiedono la capacità di cooperare, coordinarsi e negoziare tra loro.
Il campo dei sistemi multiagente è influenzato e ispirato da: economia, filosofia, teoria dei giochi, logica, ecologia, scienze sociali.
Mentre l'IA classica ignora gli aspetti sociali (di interazione) dell'agire, questo concetto è fondamentale nei MAS.

AGENTE INTELLIGENTE
Sistema di computazione capace di agire autonomamente in un qualche ambiente con il fine di raggiungere gli obiettivi per cui è progettato. La principale caratteristica degli agenti è la loro autonomia.
Uno dei paradigmi principali: mettere sullo stesso piano **agente** e **ambiente**.
L'agente (processo) riceve input dall'ambiente (dati) e produce un output ovvero agisce sull'ambiente (dati).
Capace di esibire azioni autonome in modo flessibile in un ambiente:
- Reattivo (reactive)
- Proattivo (pro-active)
- Sociale (social)

AI forte: è possibile riprodurre l'intelligenza umana?
AI debole: esistono modi automatici per risolvere problemi che ad un essere umano richiedono intelligenza?

Il software non è né buono né cattivo ma può essere scritto male.

REATTIVITÀ
Se l'ambiente di un software è statico/fisso non è necessario preoccuparsi di esso.
Nel mondo reale l'ambiente è dinamico, le informazioni sono incomplete: è necessario considerare che l'esecuzione di azioni non abbia successo, o se vale la pena eseguire una certa azione.

AGENTE REATTIVO
È un sistema che mantiene una costante interazione con l'ambiente e risponde ai cambiamenti che occorrono su di esso (in tempo perché la risposta sia utile).

Può essere implementato tramite semplici regole condizionali.

PROATTIVITÀ

In generale si vuole che gli agenti facciano cose per noi (delega di compiti): comportamenti guidati dagli obiettivi.
Proattività: tentare di raggiungere gli obiettivi, non solo guidati dagli eventi, prendere l'iniziativa. Riconoscere le opportunità.

AGENTE PROATTIVO

Mantengono uno stato che contiene due tipi di conoscenza:

- L'agente necessita info su come l'ambiente evolve
- L'agente necessita info su come le proprie azioni impattano sull'ambiente

L'agente necessita inoltre info sull'obiettivo (goal) in modo che possa agire per raggiungerlo.

REATTIVO VS PROATTIVO

Si desidera che gli agenti siano reattivi e che rispondano ai cambiamenti per tempo ma allo stesso tempo che lavorino in modo sistematico verso obiettivi a lungo termine. Queste due caratteristiche sono un po' in conflitto. Trovare un buon bilanciamento tra reattività e proattività è ancora un problema di ricerca aperto.

ABILITÀ SOCIALI

Il mondo reale è un ambiente multi-agente: non possiamo raggiungere i propri obiettivi senza la collaborazione di altri.
Allo stesso modo i sistemi sono immersi in una rete di computer.
Per social ability di un agente si intende l'abilità di interagire con altri agenti (anche umani) attraverso un qualche linguaggio di comunicazione (**agent-communication language, ACL**) e cooperare con essi.

Agente intelligente o generico programma?

Un agente autonomo è un sistema situato in un ambiente che può percepire ed agire su di esso nel tempo, con il fine di perseguire una propria agenda e così facendo influire nelle successive percezioni. (Franklin e Graesser).

RAGIONAMENTO SIMBOLICO

Il classico approccio per costruire agenti intelligenti è quello di vederli come casi particolari di sistemi basati sulla conoscenza (**symbolic AI**).
Architettura di un agente deliberativo:

- Contiene una esplicita rappresentazione (**modello simbolico**) dell'ambiente
- Prende decisioni (ad esempio quale azione eseguire) attraverso un **ragionamento simbolico**

Per cui sono due i problemi da affrontare:

- **Problema della trasduzione**: problema della traduzione del mondo reale, dell'ambiente, in una descrizione simbolica accurata, **in tempo** perché possa essere utile
- **Problema della rappresentazione/ragionamento**: problema di come rappresentare simbolicamente le informazioni e come fare in modo che gli agenti ragionino con queste informazioni **in tempo** perché i risultati possano essere utili.

Entrambi questi problemi non sono facilmente risolvibili, poiché la maggior parte degli algoritmi di manipolazione simbolica sono **altamente intrattabili**.

Sostanzialmente una macchina si occupa di calcolare funzioni da naturali a naturali e lo fa codificando la funzione stessa come un numero naturale. Il vero problema non è allargare il numero di funzioni da calcolare, bensì trovare il modo di definire quella funzione, in altre parole programmarla. Le metodologie informatiche e l'IA facilitano la ricerca di quella funzione che calcola quello che desideriamo noi.

AGENTI RAZIONALI

AGENTI COME SISTEMI INTENZIONALI

Spesso per spiegare comportamenti umani si attribuiscono atteggiamenti (attitudes come credere, volere ecc.) legati a una psicologia popolare. Queste attitudini sono chiamate **nozioni intenzionali**.

Sistema intenzionale: entità il cui comportamento può essere previsto dal metodo di attribuzione di credenze, desideri, acume razionale (filosofo Daniel Dennet).
Secondo McCarthy questa definizione in certe occasioni può essere attribuita anche a sistemi informatici (specialmente sistemi informatici complessi).
Infatti in un sistema molto complesso una spiegazione meccanicistica del suo comportamento potrebbe essere non praticabile. Mentre una visione intenzionale può semplificarne la descrizione: poiché il sistema crede questo, si comporterà così.
Più i sistemi diventano complessi più hanno bisogno di astrazioni e metafore per spiegare il loro funzionamento. Le posizioni intenzionali sono una tale astrazione.
(Y. Shoham)
Gli agenti intelligenti e, in particolare, i sistemi intenzionali rappresentano una potente astrazione.

Posizione intenzionale: strumento di astrazione, modo conveniente di parlare di sistemi complessi che ci permette di prevedere e spiegare il loro comportamento senza dover capire come funziona effettivamente.

PRACTICAL REASONING

È il ragionamento sulle azioni, sul processo di capire cosa fare: valuta le varie opzioni in modo da decidere cosa compiere, nella valutazione ciò che conta è valutare i desideri e ciò che l'agente crede.

Nelle attività umane corrisponde a due attività:

- **Deliberazione (deliberation)**: decide quale stato di cose vogliamo raggiungere. L'output della deliberazione sono le **intenzioni**.
- **Pianificazione (means-ends reasoning** o **planning)**: decide come raggiungere questi stati di cose. Il fine del planning è ottenere un **piano** per raggiungere lo stato di cose scelto.

Deliberation e planning sono processi computazionali e come tali negli agenti reali avranno luogo in presenza di risorse limitate (tempo, calcolo).

Un agente per cui deve controllare il suo ragionamento: non sprecare tutta la sua potenza di calcolo e ad un certo punto deve prendere una decisione e impegnarsi nel raggiungere il proprio obiettivo.

Intenzioni: stato di cose che un agente ha scelto e al quale si impegna come sue intenzioni.

Il ruolo delle intenzioni è quello di favorire la proattività, cioè tendono a portare ad agire.

Bratman osserva che: le intenzioni hanno un ruolo molto più forte nell'influenzare l'azione rispetto ad altri atteggiamenti proattivi come il desiderio, infatti un'intenzione comunica che è già stato deciso che verranno eseguite determinate azioni.

Persistenza delle intenzioni: fintanto che la motivazione dell'intenzione persiste devo insistere e tentare di realizzarla

Inoltre le intenzioni vincolano il futuro practical reasoning: non sceglierò opzioni che non sono coerenti con le proprie intenzioni.

Per cui le intenzioni:

- Pongono problemi agli agenti che devono determinare modi per realizzarli
- Forniscono un filtro per l'adozione di altre intenzioni che non devono entrare in conflitto
- Gli agenti monitorano il successo delle loro intenzioni e sono inclini a riprovare se i loro tentativi falliscono
- Gli agenti credono che le loro intenzioni siano possibili
- Gli agenti non credono che non porteranno avanti le loro intenzioni
- In determinate circostanze gli agenti credono che realizzeranno le loro intenzioni
- Gli agenti non devono necessariamente avere intenzione su tutti gli effetti collaterali delle loro intenzioni

Agent control loop version 1 - *(agente che utilizza il practical reasoning)*

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

Problema: i processi di deliberation e means-ends reasoning hanno un costo in termini di tempo, impiegano risorse. Per cui l'intenzione ottenuta dalla deliberation era ottimale per il modello del mondo percepito dall'agente all'inizio del processo di deliberation, ma nel mentre il modello del mondo potrebbe essere cambiato e l'intenzione selezionata precedentemente potrebbe non essere più quella ottimale. (*Calculative rationality)*
La fase di deliberazione è solo metà del problema, l'agente deve ancora determinare come realizzare l'intenzione. Quindi l'agente avrà un comportamento ottimale solo nelle seguenti circostanze:

- Quando il tempo impiegato per i processi di deliberazione e di planning è incredibilmente piccolo
- Quando il mondo rimane statico mentre l'agente esegue i processi di deliberazione e planning
- Quando un'intenzione ottimale se raggiunta al tempo t0 (cioè il momento in cui l'agente osserva il mondo) è garantita rimanere ottimale fino al tempo t2 (ovvero quando l'agente ha determinato le azioni per raggiungere l'intenzione)

Agent control loop version 2

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

brf: belief revision function, serve per rivedere le proprie credenze in base alla percezione appena catturata.
Uso il nuovo insieme di belief per deliberare e ottenere l'intenzione.
Dopodiché costruisco un piano di azioni tramite l'insieme di belief e l'intenzione ed eseguo il piano.

BELIEF, DESIRE, INTENTION (BDI)

Un agente delibera cercando di capire quali sono le opzioni a sua disposizione (**desires**) sulla base delle proprie informazioni e credenze (**beliefs**). Sceglie tra le opzioni possibili e si impegna (**commit**) verso queste. Le opzioni scelte sono le **intenzioni**.

Agent control loop version 3

```
B := B0;
I := I0;
while true do
      get next percept ρ;
      B := brf (B, ρ);
      D := options(B, I);
      I := filter(B, D, I);
      π := plan(B, I);
      execute(π);
```

La funzione di deliberazione viene scomposta in due componenti:

- **Option generation**: l'agente genera un insieme di possibili opzioni, desires.
- **Filtering**: l'agente sceglie tra le opzioni possibili per determinare il nuovo insieme di intenzioni.

COMMITMENT STRATEGIES

Strategie che un agente può adottare per raggiungere le proprie intenzioni:

- **Blind commitment** *(fanatico)*: l'agente continuerà a mantenere un'intenzione fino a quando non crederà che l'intenzione sia stata effettivamente raggiunta.
- **Open-minded commitment**: l'agente manterrà un'intenzione fintanto che la ritiene possibile.
- **Single-minded commitment**: l'agente continuerà a mantenere un'intenzione finché non ritiene che l'intenzione sia stata raggiunta o che non sia più possibile raggiungerla.

L'agente si impegna sia al fine (ends, cioè i desideri da realizzare) che ai mezzi (means, cioè il modo con cui l'agente desidera raggiungere l'intenzione). Attualmente il nostro ciclo di controllo dell'agente (version 3) è **overcommitted** (impegnato in modo eccessivo) sia nei mezzi che nei fini: o completa il piano o le intenzioni non verranno riconsiderate.

Agent Control Loop Version 4

```
B := B0;
I := I0;
while true do
      get next percept ρ;
      B := brf (B, ρ);
      D := options(B, I );
      I := filter (B, D, I );
      π := plan(B, I );
      while not empty(π) do
            α := hd(π);   //Seleziona la prima azione di π
            execute(α);
            π := tail(π);   //Aggiorna π, levando la prima azione
            get next percept ρ;
            B := brf (B, ρ);
            if not sound(π, I, B) then
                  π := plan(B, I);
```

Mi chiedo se il piano calcolato rispetto alle intenzioni e alle nuove credenze è corretto.
Se il piano non mi porta più a risolvere le mie intenzioni, creo un nuovo piano.
Pertanto dopo ogni azione eseguita controllo tramite una nuova percezione se il piano corrente è ancora quello adatto, altrimenti ripianifico.
Questo però risolve solamente l'overcommitment rispetto ai mezzi (al piano), l'agente è ancora overcommited rispetto alle intenzioni.

Agent Control Loop Version 5

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

Pertanto occorre fermarsi per determinare se le intenzioni hanno avuto successo o se sono diventate impossibili da soddisfare. *(Single-minded commitment)*
Tutta via questo approccio (Version 5) limita il modo in cui viene concesso a un agente di riconsiderare le sue intenzioni, può farlo solo quando:

- Ha completato il piano
- Ritiene di aver raggiunto le sue attuali intenzioni
- Crede che le sue attuali intenzioni non siano più possibili

Per cui potremmo riconsiderare le intenzioni dopo l'esecuzione di ogni azione.

Agent Control Loop Version 6

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

Ma la riconsiderazione delle intenzioni costa.
Dilemma: se non riconsidera abbastanza spesso le intenzioni, l'agente continuerà in quella direzione anche se sarà palese che non è più conveniente proseguire verso quelle intenzioni; tuttavia se riconsidera costantemente le sue intenzioni può dedicare troppo tempo a questo e non raggiungerle mai effettivamente.
Pertanto conviene incorporare una specifica componente di controllo di *meta-livello* che decide se eseguire o meno la riconsiderazione.
Mi chiedo se in base alle mie intenzioni e i miei beliefs valga la pena di riconsiderare le mie intenzioni.
La funzione di *reconsider()* è stabilita su euristiche diverse, il suo costo è molto inferiore al processo deliberativo stesso.

Agent Control Loop Version 7

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

Diversi tipi di strategia di riconsiderazione delle intenzioni:

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

Kinny e Georgeff studiano l'efficacia di queste due strategie.

Il dinamismo nell'ambiente è rappresentato dal tasso di cambiamento del mondo γ.

Risultati che ottengono:

- Se γ è basso, ovvero se il mondo cambia poco, gli agenti audaci fanno meglio rispetto a quelli cauti
- Se γ è alto, cioè se il mondo cambia spesso, gli agenti cauti fanno meglio di quelli audaci

Pertanto per un corretto funzionamento della funzione di reconsider() bisogna intuire quanto l'ambiente sia soggetto a cambiamenti.

LOGICA PER GLI AGENTI

La logica fornisce strumenti formali per:

- **Rappresentazione della conoscenza** (linguaggio formale con semantica precisa, rappresentazione dichiarativa della conoscenza)
- **Ragionamento automatico** (regole d'inferenza)

RUOLO DELLA LOGICA PER GLI AGENTI

Può essere utilizzata da un agente intelligente per rappresentare la conoscenza e ragionare, dato che la conoscenza è espressa in un linguaggio formale, gli agenti possono usare metodi formali (inferenza) per derivare altra conoscenza.
Inoltre la logica può specificare il comportamento di un agente: ovvero viene usata per verificare che l'agente si comporti come specificato, anche se esso non fa uso della logica nel suo funzionamento.

LOGICA CLASSICA

Ossia logica proposizionale e del prim'ordine. Utilizzata maggiormente in ambito IA. Tuttavia la necessità di modellare concetti diversi e le esigenze di efficienza hanno portato l'IA ad usare logiche diverse da quelle classiche, ad esempio logiche non monotone.
Nell'ambito degli agenti si preferisce usare una logica non classica nota come logica modale.

LOGICA MODALE (INTRO)

Agenti spesso descritti come sistemi intenzionali, con stati mentali come credenze e desideri.
Supponiamo di voler formulare con la logica la frase:
John crede che Superman voli
In logica classica sarebbe:
Bel(John, vola(Superman))
Questo ha due problemi:

- Il primo è sintattico: le formule della logica classica hanno questa struttura *Predicato(Term, ..., Term)* tuttavia in *Bel(John, vola(Superman))* il secondo argomento di Bel è una formula e non un termine.
- Il secondo è semantico: gli operatori come Bel sono **referentially opaque**, ovvero creano dei contesti chiusi in cui non è possibile sostituire una formula con una equivalente come in logica classica. Per esempio assumendo vero *Bel(John, vola(Superman))* e che Superman e Clark denotino lo stesso individuo, da questo

non possiamo derivare *Bel(John, vola(Clark))* a meno che John non creda che Superman = Clark

Per risolvere queste inadeguatezze:

- **Logica modale**: fornisce operatori modali come Bel che possono avere anche formule come argomenti
- Oppure rimanere nella logica classica ma usare un **meta-linguaggio**, ossia un linguaggio del prim'ordine contenente termini che denotano formule (approccio non presentato in questo corso)

La semantica della logica modale (dovuta principalmente a Hintikka e Kripke) è basata sulla nozione di mondi possibili. Ogni mondo rappresenta una situazione considerata possibile dall'agente. Ciò che è vero in tutti i mondi si può dire che sia creduto dall'agente.

Ad esempio, un agente che sta giocando a carte conosce tutte le carte che ha in mano ma non quelle degli atri giocatori. Ogni configurazione delle carte di tutti i giocatori, consistente con quanto l'agente conosce, è un mondo possibile. Se l'agente possiede l'asso di picche, si può assumere che lui creda di possedere l'asso di picche, e quindi in ogni mondo possibile sarà vero che lui possiede l'asso di picche.

La proprietà fondamentale della logica modale è che tutto ciò che è vero in tutti i mondi è **necessariamente vero**. Ciò che è vero in qualche mondo è **possibile**.

Possiamo passare dalla logica classica alla logica modale inserendo gli operatori modali.

LOGICA PROPOSIZIONALE

La semantica della logica preposizionale (classica) definisce la verità delle formule rispetto ad ogni modello. Un modello assegna un valore di verità ad ogni simbolo proposizionale.

Dato un insieme P che contiene tutti i simboli preposizionali di un certo problema, se $|P|=n$, ci sono 2^n modelli. Un modello può essere rappresentato come un insieme $M \subseteq P$ che contiene tutte le proposizioni atomiche che sono vere nel modello. Quelle che non appartengono ad M sono false.

Una formula è soddisfacibile se e solo se c'è qualche modello che la soddisfa.

Una formula è valida (tautologia) se e solo se è soddisfatta da ogni modello.

LOGICA MODALE

Sviluppata da filosofi interessati nella distinzione di verità necessaria e verità contingente:

- Verità contingente: è vero adesso ma non è detto lo sia per sempre (ad esempio il fatto che un partito abbia la maggioranza adesso, ma non si sa tra un po')
- Verità necessaria: la radice quadrata di 2 non è un numero razionale, sempre vero.

LOGICA MODALE PROPOSIZIONALE

Estende la logica proposizionale classica con gli operatori □ and ◊ (applicabili a formule della logica).

◊ piove = è possibile che piova

□ piove = necessariamente piove

Sono uno il duale dell'altro:

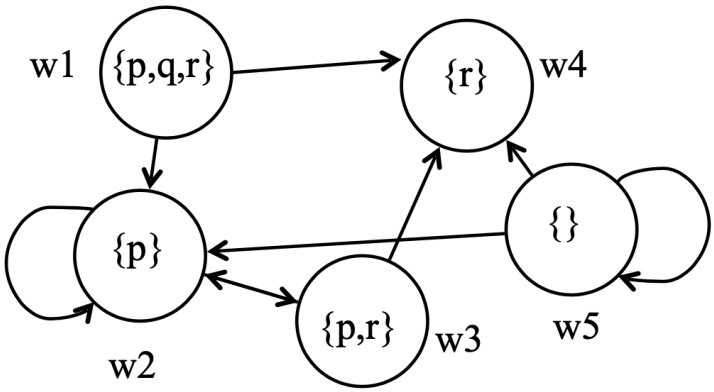
□ piove ≡ ¬◊ ¬piove.

Ovvero necessariamente piove è equivalente a non è possibile che non piova.

Un modello nella logica modale è dato come un insieme di mondi possibili (mentre nella logica classica c'è un solo mondo). Ogni mondo è costituito da un insieme di proposizioni che sono considerate vere in quel mondo. I mondi possibili sono collegati tra loro attraverso una relazione di accessibilità, relazioni binarie che collegano coppie di mondi.

Un modello è una tripla **<W, R, L>** con:

- **W** è un insieme di mondi
- **R** ⊆ W x W è una relazione di accessibilità fra mondi
- **L**: W —> 2^p dà l'insieme di proposizioni vere in ogni mondo ∈ W



Un modello può essere rappresentato con un grafo: i nodi sono i vari mondi, gli archi sono le relazioni (anche loop), lo stato dei nodi rappresenta le proposizioni vere in quel mondo.

La relazione di accessibilità R è utilizzata per definire la semantica degli operatori modali:
La formula $\Box\phi$ è vera in un mondo $w \in W$ se ϕ è vera in tutti i mondi accessibili da w.
La formula $\Diamond\phi$ è vera in un mondo $w \in W$ se ϕ è vera in qualche mondo accessibile da w.
La soddisfacibilità di una formula ϕ è definita rispetto a un modello M e ad un mondo w di questo modello con la notazione $M \models_w \phi$.

- $M \models_w p$ iff $p \in L(w)$
- $M \models_w \phi \vee \psi$ iff $M \models_w \phi$ or $M \models_w \psi$
- $M \models_w \neg\phi$ iff $M \not\models_w \phi$
- $M \models_w \Box\phi$ iff $(\forall w': R(w,w') \Rightarrow M \models_{w'} \phi)$
- $M \models_w \Diamond\phi$ iff $(\exists w': R(w,w') \Rightarrow M \models_{w'} \phi)$

Una formula ϕ è **valida** se è vera in tutte le coppie modello/mondo (tautologia).
I modelli della logica modale sono spesso chiamati **strutture di Kripke**.
La coppia <W, R> è chiamata **frame di Kripke**, è un normale grafo, senza nessuna informazione inserita nei nodi.

PROPRIETÀ DELLA LOGICA MODALE

Dalla semantica della logica modale seguono due proprietà fondamentali, che valgono su qualunque logica modale:

- **Assioma K**: $\Box(\phi \Rightarrow \psi) \Rightarrow (\Box\phi \Rightarrow \Box\psi)$
- **Necessitation**: regola di inferenza. Se ϕ è valida, allora $\Box\phi$ è valida
- "Box" si distribuisce sull'and ma non sull'or

LOGICA MODALE PER GLI AGENTI

Per quanto riguarda le problematiche degli agenti si possono attribuire agli operatori modali significati diversi (rispetto a necessario e possibile). Ad esempio potremmo attribuire all'operatore $\Box$ il significato di credenza (belief):
$\Box\phi$ —> l'agente crede che ϕ sia vero
Usiamo un nome diverso per $\Box$, ad esempio **B** per le credenze.
L'operatore duale possiamo non definirlo dato che possiamo fare uso dell'equivalenza: $\Box\phi \equiv \neg\Diamond\neg\phi$

PRINCIPALI LOGICHE MODALI PER AGENTI

Logica epidemica (conoscenza e credenze)

$K_a\phi$ l'agente a sa che ϕ è vero
$B_a\phi$ l'agente a crede che ϕ sia vero

Logiche Belief-Desire-Intention

$B_a\phi$ 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 (obblighi e permessi)

$O\phi$ è obbligatorio che ϕ

$P\phi$ è permesso che ϕ

Logica temporale (lineare)

Non proprio considerata una logica modale, ma gli somiglia parecchio.

$X\phi$ ϕ sarà vero al prossimo istante

$G\phi$ ϕ sarà sempre vero

$F\phi$ ϕ prima o poi diventerà vero

$\psi \mathbf{U} \phi$ ψ è vero fino a quando ϕ diventa vero

Logica dinamica (delle azioni)

$[\pi]\phi$ dopo l'esecuzione del programma π, ϕ è vero

(dove π è un'azione complessa ottenuta combinando azioni elementari)

Pertanto la logica modale deve essere rimodellata per ottenere le proprietà che vogliamo per gli agenti. Questo può essere fatto aggiungendo assiomi.

Ad esempio se $\Box$ rappresenta la conoscenza, vorremmo che la logica avesse la proprietà che tutto ciò che è conosciuto è vero.

Per farlo aggiungiamo l'assioma: $\Box\phi \Rightarrow \phi$ (ovvero conoscere ϕ implica che ϕ è vero)

Non è detto però che questo valga in tutti i modelli.

Oppure potremmo voler esprimere che le conoscenze dell'agente non devono essere contraddittorie aggiungendo l'assioma: $\Box\phi \Rightarrow \neg\Box\neg\phi$ (ovvero se conosco ϕ non conosco $\neg\phi$).

Tutto questo per dire che per modellare un problema spesso è necessario aggiungere qualcosa (tramite assiomi).

PROPRIETÀ DEI FRAME

Non è obbligatorio dover aggiungere assiomi, in alternativa è possibile restringere la classe dei modelli formulando delle condizioni per i **frame**.

Ad esempio potremmo richiedere che valga la **riflessività**, ossia che ogni mondo di un frame acceda a sé stesso (che è un vincolo che mettiamo sulla struttura del frame). Principali proprietà dei frame:

- **Reflexive**: $\forall w \in W.\ (w,w) \in R$
- **Serial**: $\forall w \in W.\ \exists\ w' \in W.\ (w,w') \in R$ (ovvero non si arriva mai ad un punto in cui non si esce da un mondo, c'è sempre un mondo raggiungibile da w)
- **Transitive**: $\forall w,w',w'' \in W.\ ((w,w') \in R \wedge (w',w'') \in R) \Rightarrow (w,w'') \in R$
- **Euclidean**: $\forall w,w',w'' \in W.\ ((w,w') \in R \wedge (w,w'') \in R) \Rightarrow (w',w'') \in R$

CORRISPONDENZA

In molti casi la formulazione con assiomi corrisponde ad attribuire condizioni ai frame: gli assiomi sono veri in tutti e soli i modelli il cui frame soddisfa le condizioni. Per esempio si può dimostrare che l'assioma $\Box\phi \Rightarrow \phi$ corrisponde alla riflessività dei frame.

PRINCIPALI CORRISPONDENZE

Axiom schema		**Condition on frames**
T	$\Box\phi \Rightarrow \phi$	**reflexive** $\forall w \in W.\ (w,w) \in R$
D	$\Box\phi \Rightarrow \Diamond\phi$	**serial** $\forall w \in W.\ \exists\ w' \in W.\ (w,w') \in R$
4	$\Box\phi \Rightarrow \Box\Box\phi$	**transitive** $\forall w,w',w''' \in W.\ ((w,w') \in R \wedge (w',w'') \in R) \Rightarrow (w,w'') \in R$
5	$\Diamond\phi \Rightarrow \Box\Diamond\phi$	**Euclidean** $\forall w,w',w'' \in W.\ ((w,w') \in R \wedge (w,w''') \in R) \Rightarrow (w',w'') \in R$

SISTEMI MODALI

Combinando le proprietà T, D, 4, 5 si ottengono 11 sistemi modali (sarebbero 16 ma alcune combinazioni sono equivalenti).
Sistemi più noti (si ricorda che l'assioma K è sempre valido):

- KT noto come **T**
- KT4 noto come **S4**
- KD45 noto come **weak-S5**
- KTD45 noto come **S5**

PROOF THEORY

A differenza della logica classica in cui ci sono vari metodi e strumenti per ragionare (come la regola di "Risoluzione" e altri theorem provers), nella logica modale il problema di definire una proof theory è complesso. Il motivo principale è che esistono diversi sistemi modali, ciascuno con proprietà (assiomi) diverse, ovvero la logica cambia, quindi non esistono metodi generali.

LOGICA EPISTEMICA

Logica della conoscenza e delle credenze, molto studiata e utilizzata nel contesto degli agenti e dell'IA.
Utilizza gli operatori modali: **K** che rappresenta la conoscenza (Knowledge) e **B** la credenza (Belief) di un agente.
Kϕ = l'agente sa che ϕ è vero
Bϕ = l'agente crede che ϕ sia vero
Gli operatori **K** e **B** hanno la stessa semantica del □.
Sapendo che necessario e possibile sono duali per esprimere ◊ si può usare la notazione ¬**K**¬ ϕ e ¬**B**¬ ϕ.

ONNISCIENZA LOGICA

Se ϕ è valida, allora **K**ϕ è valida (**necessitation**). Supponiamo che ψ sia conseguenza logica di ϕ. Allora ϕ ⇒ ψ deve essere valida. Secondo la **necessitation**, questa formula deve essere conosciuta dall'agente **K**(ϕ ⇒ ψ). Questo significa che l'agente deve conoscere tutte le tautologie, che sono infinite. Inoltre l'agente, per l'**assioma K**, deve conoscere ψ. Questo significa che l'agente conosce tutte le conseguenze che può derivare da quello che sa (chiuso rispetto alla conseguenza logica).
Questo è il problema della **onniscienza logica**, ed ha fatto pensare a molti che la logica modale non fosse un giusto approccio per modellare agenti reali.

ASSIOMI PER KNOWLEDGE BELIEF

Assioma D (serialità)

Dice che la conoscenza di un agente è non-contraddittoria.
Kϕ ⇒ ¬**K**¬ϕ se l'agente conosce ϕ allora non conosce ¬ϕ.

(È come dire box(ϕ) implica diamond(ϕ)). Assioma ragionevole sia per **K** che per **B**.

Assioma T (riflessività)

Kϕ ⇒ ϕ ciò che l'agente conosce è vero.

Accettabile per la conoscenza ma non per le credenze: non vogliamo che un agente

conosca qualcosa che è falso, ma accettiamo che creda vero qualcosa che è falso.

Assioma 4 (transitività)

Kϕ ⇒ **KK**ϕ introspezione positiva

Se l'agente sa ϕ, sa di sapere ϕ.

Assioma 5 (Euclidean)

¬**K**ϕ ⇒ **K**(¬**K**ϕ) introspezione negativa

Se l'agente non sa ϕ, allora sa di non saperlo.

Questi ultimi due assiomi implicano che l'agente ha conoscenza perfetta su quello che sa e quello che non sa.

Pertanto il sistema modale **S5** (KTD45) viene scelto come logica della conoscenza, mentre **weak-S5** (KD45) come logica delle credenze.

RAGIONARE SU TEMPO E AZIONI

Le principali caratteristiche di un agente sono di essere situato in un ambiente e di agire su di esso nel corso del tempo e sulla base delle percezioni e dei propri obiettivi. Per fare questo deve ragionare sul tempo e sulle azioni.

LOGICA TEMPORALE

È la logica modale del tempo. Noi tratteremo solo il caso in cui il tempo:

- È discreto
- Ha un istante iniziale
- È infinito nel futuro

Delle molte varianti della logica temporale noi consideriamo solo la Linear-time logic e la branching-time logic.

LINEAR TEMPORAL LOGIC (LTL)

Semantica

Struttura del tempo lineare, struttura come un insieme totalmente ordinato di istanti di tempo. Un modello M è una **struttura lineare** <S, x, L> dove:

- S è un insieme di stati
- X: N –> S è una sequenza infinita di stati (N: numeri naturali)
- L: S –> 2^p dà l'insieme delle proposizioni vere in ogni stato

Sintassi

Le formule della **Propositional Linear Temporal Logic (PLTL)** sono quelle della logica preposizionale classica più:

- **X**α ("nexttime α") ovvero α sarà vero nel prossimo istante di tempo
- α**U**ß ("α until ß") ovvero questa formula è vera all'istante t sse ß è vera in un futuro istante t' e α è vera in tutti gli istanti tra t e t'.

Da questi operatori è possibile derivare altri due operatori modali:

- **F**α ≡ *true* **U** α (“prima o poi α”)
- **G**α ≡ ¬**F**¬α (“sempre α”, è il duale di **F**)

COMPUTATION TREE LOGIC (CTL*)

La struttura del tempo è fatta ad albero (branching-time) dove ogni istante può avere più istanti successori (alberi infiniti).
In CTL* si possono formulare:

- **Path formulas**: che riguardano i cammini infiniti della struttura temporale ad albero (simili alle formule di LTL)
- **State formulas**: che riguardano tutti i cammini infiniti uscenti da uno stato

SINTASSI

State formulas

Oltre alle solite della logica preposizionale classica abbiamo:

- **A**π significa “per tutti i cammini uscenti da uno stato vale π”
- **E**π significa “esiste un cammino uscente da uno stato per cui vale π”

Dove π è una path formula. **A** e **E** sono duali.

Path formulas

Sostanzialmente come la LTL: aggiunte le formule **X**π e π**U**ρ, con π e ρ path formulas.

SEMANTICA

Un modello è una tripla <S, T, L> dice S e L sono come in LTL, mentre T informalmente è un albero infinito i cui nodi sono stati.
Dato un modello M, la semantica di una state formula si riferisce a uno stato (M, s⊨α) e la semantica di una path formula si riferisce a un cammino (M, x⊨π).
Ad esempio: M, $s_0 \models$ **A**π iff ∀path x = (s_0,s_1,s_2,...). M, x ⊨ π.
Tuttavia la CTL* è troppo complessa per essere utilizzata nel model checking, quindi si utilizza una logica chiamata CTL che restringe la sintassi, riduce le proprietà.
CTL non permette la combinazione di operatori linear-time, quindi le path formulas diventano: **X**α e α**U**ß, con α e ß state formula.

In pratica LTL e CTL sono utilizzare a seconda del tipo di problema che si intende risolvere perché hanno caratteristiche diverse, alcune formule espresse in CTL non sono esprimibili in LTL e viceversa.

RAGIONARE SULLE AZIONI

Prima trattazione nel 1969, McCarthy introduce il **calcolo delle situazioni** (*situation calculus*). Aspetti principali del calcolo delle situazioni:

- **Situazioni**: stato del mondo a qualche istante di tempo. S_0 situazione iniziale, *do(a, s)* denota una situazione risultante dall'esecuzione dell'azione *a* nello stato *s*
- **Fluenti**: proposizioni il cui valore varia da una situazione all'altra
- **Azioni**: causano un cambiamento nello stato del mondo

Il ragionamento sulle azioni si basa sulla logica classica.

AZIONI: PRECONDIZIONI ED EFFETTO

Ogni azione è descritta da due assiomi:

- **Assioma di possibilità**: dice quando è possibile eseguire l'azione
- **Assioma di effetto**: specifica quello che accade quando un'azione possibile è eseguita

Esempi sulla possibilità

Posizione(Agente,x,s) $\wedge$ Adiacente (x,y) $\Rightarrow$ Poss(Vai(x,y),s).

Oro(g) $\wedge$ Posizione(Agente,x,s) $\wedge$ Posizione(g,x,s) $\Rightarrow$ Poss(Afferra(g),s).

Portando(g,s) $\Rightarrow$ Poss(Lascia(g),s).

Esempi sull'effetto

Poss(Vai(x,y),s) $\Rightarrow$ Posizione(Agente,y, do(Vai(x,y),s)).

Poss(Afferra(g),s) $\Rightarrow$ Portando(g,do(Afferra(g),s)).

Poss(Lascia(g),s) $\Rightarrow$ ¬Portando(g,do(Lascia(g),s)).

FRAME PROBLEM

Un'azione influenza solo un numero limitato di fluenti, quindi come esprimere in modo parsimonioso (senza troppi sprechi) che tutto il resto non cambia?
Frame problem: chiamato così perché corrisponde a ciò che succede in un film quando si passa da un fotogramma ad un altro.
Un approccio è quello di scrivere assiomi di frame che affermino esplicitamente ciò che non cambia. Ma questo approccio è troppo costoso.

SUCCESSOR STATE AXIOMS

Soluzione del frame problem (proposta da Reiter) introducendo i **Successor State Axioms**, ovvero degli assiomi che definiscono lo stato successivo, facendo riferimento ad un fluente per volta

Ad esempio la formula fra parentesi quadre qui sotto mostra come una azione può cambiare il valore del fluente *broken*.

$Poss(a,s) \Rightarrow [broken(x,do(a,s)) \equiv \exists r.\ \{a=drop(r,x) \wedge fragile(x,s)\} \vee broken(x,s) \wedge \neg\exists r.\ a=repair(r,x))]$

GOLOG

Linguaggio di programmazione basato sul situation calculus. Usato per programmare robot di alto livello e agenti intelligenti. Le azioni primitive in GOLOG sono specificate dandone le precondizioni e effetti (rappresentati come *successor state axioms*).

PERCHÉ SISTEMI DISTRIBUITI DI AGENTI?

Le soluzioni centralizzate sono spesso impraticabili perché i sistemi e i dati coinvolti appartengono a organizzazioni indipendenti.

Le informazioni coinvolte sono distribuite e risiedono in sistemi informativi di grosse dimensioni e complessi sotto diversi punti di vista:

- Sono distribuiti geograficamente
- Sono composte da molte parti indipendenti
- I contenuti sono di grosse dimensioni, sia per numero di concetti coinvolti che per dati associati ad ogni concetto, cioè ogni concetto ha associato una moltitudine di dati
- Hanno scopi più ampi, coprono una porzione maggiore del dominio considerato

DISTRIBUTED ARTIFICIAL INTELLIGENCE (DAI)

Ci sono quattro tecniche principali per affrontare la dimensione e la complessità di un sistema informativo:

- Modularità
- Distribuzione
- Astrazione
- Intelligenza, ossia mostrare intelligenza nella ricerca e nella modifica delle informazioni

L'uso di moduli distribuiti intelligenti combinati a queste quattro tecniche produce un approccio di intelligenza artificiale distribuito (**DAI**)

Gli agenti possono essere considerati facenti parte di questo approccio.

SISTEMI AUTONOMI DI AGENTI

Per lo sviluppo di soluzioni globali e distribuite, gli agenti necessitano di essere eseguiti in maniera autonoma e sviluppati indipendentemente.
Gli agenti si coordinano e cooperano per la soluzione di problemi, condividendo capacità e lavorando in parallelo, affrontando possibili errori attraverso la ridondanza e rappresentando molteplici punti di vista ed esperienze.
Sistemi autonomi di agenti rappresentano soluzioni modulari e riutilizzabili.
I sistemi multiagente sono il modo migliore per caratterizzare o progettare sistemi distribuiti di computazione.

COMUNICAZIONE

Gli agenti operano in ambienti che contengono altri agenti.
Un sistema multiagente è un sistema che contiene agenti che interagiscono tra di loro attraverso la **comunicazione**.
Gli ambienti multiagente, per garantire la comunicazione, hanno le seguenti caratteristiche:

- Forniscono un'**infrastruttura** specifica per la comunicazione e l'utilizzo di protocolli di interazione
- Sono progettati per essere **aperti** (non sottoposti a vincoli di entrata) e **distribuiti**
- Gli agenti ospitati sono **autonomi**, **self-interested** o **cooperativi**

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

- **protocolli di comunicazione**: per permettere agli agenti di comprendere i messaggi scambiati
- **protocolli di interazione**: per permettere agli agenti di svolgere conversazioni complesse, ossia scambi strutturati di messaggi

Un protocollo di comunicazione potrebbe includere i seguenti tipi di messaggio:

- Proposta di esecuzione di un'azione
- Accettare/Rifiutare/Ritirare una proposta di esecuzione di un'azione
- Esprimere disaccordo rispetto una proposta di esecuzione di un'azione
- Proporre una differente esecuzione di azione

La comunicazione permette agli agenti di coordinare le loro azioni e i loro comportamenti. Questa abilità è parte della **percezione** (ricevere messaggi) e parte delle **azioni** (inviare messaggi).
La **coordinazione** è una proprietà di un sistema di agenti che eseguono attività in un ambiente condiviso.
La **cooperazione** è coordinazione tra agenti non antagonisti.

La **negoziazione** è coordinazione tra agenti antagonisti, competitivi o semplicemente self-interested (es. sistemi di aste).

AGENT COMMUNICATION LANGUAGE (ACL)

Un ACL fornisce agli agenti i mezzi per scambiarsi informazioni e conoscenza. È solitamente ad un livello più alto rispetto gli strumenti di comunicazione per i sistemi distribuiti, come remote procedure call o method invocation.
Un ACL tratta con proposizioni, regole, azioni anziché semplici oggetti.
Un messaggio in ACL descrive uno stato desiderato attraverso un linguaggio dichiarativo, spesso in termini di stati mentali (mental attitudes).
KQML, **FIPA** sono due esempi di ACL, JASON utilizza KQML e JADE utilizza FIPA.

ATTI COMUNICATIVI

I filosofi Austin e Searle introducono la teoria nota come **speech act theory** (teoria degli atti comunicativi) che tratta la **comunicazione come azioni**, come requesting, informing, replying...
Gli atti comunicativi sono spiegati in termini di intenzioni degli agenti, con riferimento a beliefs, desires, intentions e altre modalità.
Un atto comunicativo ha tre aspetti (si consideri l'esempio *Shut the door*):

- **Locuzione** (locution), l'atto fisico compiuto da chi parla, la produzione di enunciati grammaticali
- **Illocuzione** (illocution), il significato inteso dell'enunciato da parte di chi parla, quello che il parlante desidera esprimere (es. il parlante desidera che l'ascoltatore chiuda la porta)
- **Perlocuzione** (perlocution), l'azione che risulta dalla locuzione (es. l'ascoltatore chiude la porta (forse))

La speech act theory usa il termine **performativa** (performative) per identificare la forza illocutoria dell'enunciato, es. promise, tell, request, convince sono performative.
Esempi (sempre considerando *Shut the door*):
performative = request
content = "the door is closed"
speech act = "please close the door"

performative = inform
content = "the door is closed"
speech act = "the door is closed!"

performative = inquire
content = “the door is closed”
speech act = “is the door closed?”

MODELLAZIONE DI ATTI COMUNICATIVI

Cohen e Perrault forniscono una semantica agli atti comunicativi utilizzando le tecniche sviluppate nell’ambito dell’intelligenza artificiale e della pianificazione. In particolare, Cohen e Perrault fanno uso di una notazione simile a STRIPS, utilizzando precondizioni ed effetti delle azioni in termini di stati mentali.
Ad esempio *Request(S, H, α)* potrebbe essere rappresentata da:

- **Preconditions**: (S Believe(H CanDo α))$\wedge$ (S Believe (H Believe (S Want α)))
- **Effects**: (H Believe (S Believe (S Want α)))

Il completamento con successo della *Request* assicura che l’ascoltatore è conscio dei desideri del parlante ma non garantisce che l’azione α sarà eseguita.

ORIGINE DI KQML

É nato all’interno del **Knowledge Sharing Effort**, progetto avviato dal DARPA (ente difesa americano) nei primi anni ’90.
Obiettivo: sviluppare tecniche, metodologie e strumenti software per la condivisione della conoscenza e il riutilizzo della conoscenza.
La condivisione della conoscenza richiede la comunicazione, che a sua volta richiede un linguaggio. Ed ecco che vennero sviluppati:

- **KQML**: un linguaggio di interazione di alto livello
- **KIF** (*Knowledge Interchange Format*): un linguaggio logico, basato sulla logica del prim’ordine, per esprimere proprietà di oggetti in una base di conoscenza
- **Ontolingua**: un linguaggio per definire ontologie condivise, permettendo di dare significati uniformi su applicazioni diverse agli stessi concetti. Un’ontologia è una concettualizzazione di un insieme di oggetti, concetti e altre entità su cui si esprime la conoscenza, e dei rapporti tra esse

KIF

Utilizzato per dichiarare:

- Proprietà di cose in un dominio (ad esempio, “Noam is chairman”)
- Relazioni tra cose in un dominio (ad esempio, “Amnon is Yael’s boss”)
- Proprietà generali di un dominio (ad esempio, “All students are registered for at least one course ”)

Ispirato a Lisp (List Programming).

KNOWLEDGE QUERY AND MANIPULATION LANGUAGE (KQML)

KQML è un linguaggio di comunicazione di alto livello e indipendente da:

- Il meccanismo di trasporto (tcp/ip, RMI, . . .)
- Il linguaggio in cui è espresso il contenuto (KIF, Prolog, . . .)
- L'ontologia utilizzata

Un messaggio KQML specifica il tipo di messaggio (performativa), quindi esprime un atto comunicativo.

KQML ignora la porzione di messaggio che fa riferimento al contenuto.

La sintassi di un messaggio KQML si basa su una notazione a lista simile a Lisp.

Consiste in una performativa seguita da un numero di coppie "parola chiave / valore" (la cui sintassi, del valore, non è rilevante).

Esempio di una query da Joe a proposito del prezzo di una quota di IBM:

```
(ask-one                              ← performativa
      :sender joe
      :receiver stock-server
      :reply-with ibm-stock
      :language LPROLOG               ← il linguaggio di rappresentazione
                                            del contenuto
      :content (PRICE IBM ?price)
      :ontology NYSE-TICKS
)
```

Parole chiave riservate di KQML:

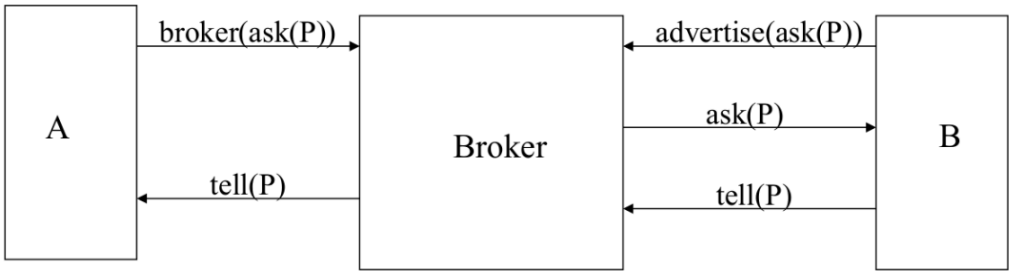
- :sender, il mittente della performativa
- :receiver, il destinatario della performativa
- :reply-with, se il mittente si aspetta una risposta, rappresenta l'identificatore della risposta
- :in-reply-to, l'identificatore della risposta specificata da :reply-with
- :language, il linguaggio di rappresentazione del :content
- :ontology, l'ontologia assunta dal parametro :content
- :content, il contenuto del messaggio

Alcune permorfative di KQML:

- achieve, il sender vuole che il receiver esegua qualcosa
- advertise, il sender afferma di essere adatto a gestire una certa performativa
- ask-one, il sender vuole una risposta tra le risposte del receiver alla domanda espressa dal content
- ask-all, il sender vuole tutte le risposte del receiver
- reply, si comunica una risposta attesa
- sorry, il sender non può fornire una risposta più specifica
- tell, il sender informa il receiver che conosce il content

KQML FACILITATORS

KQML introduce una classe speciale di agenti chiamati "communication facilitators" che dispongono di un insieme di performative per inoltrare messaggi, trovare servizi, ovvero funzioni di brokeraggio. In generale per facilitare la comunicazione tra agenti.



SEMANTICA DI KQML

Inizialmente non disponeva di una semantica. Si preoccupava di costruire il contenitore dove mettere il messaggio ma non di esprimerne il contenuto.
Labrou e Finin introducono una semantica per KQML in termini di **precondizioni** (preconditions), **postcondizioni** (postconditions) e **condizioni di completamento** (completions conditions):

- Dato un mittente A e un destinatario B, le precondizioni **Pre(A)** descrivono lo stato necessario per A al fine di inviare la performativa, e le precondizioni **Pre(B)** descrivono per B per accettarla e elaborarla con successo
- Dato un mittente A e un destinatario B, le postcondizioni **Post(A)** descrivono lo stato di A dopo aver espresso la performativa e le postcondizioni **Post(B)** descrivono lo stato di B dopo aver ricevuto il messaggio
- Le condizioni di completamento di una performativa descrivono lo stato finale dopo che una conversazione sia stata completata

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

Semantica di **tell(A,B,X)**

Pre(A) BEL(A,X) $\wedge$ KNOW(A,WANT(B,KNOW(B,S)))

Pre(B) INT(B,KNOW(B,S))

dove S può essere BEL(B,X) oppure ¬BEL(B,X)

Post(A) KNOW(A,KNOW(B,BEL(A,X)))

Post(B) KNOW(B,BEL(A,X))

Completion KNOW(B,BEL(A,X))

Un agente non può fornire informazioni non richieste. Una proactive tell potrebbe avere come come precondizioni:
Pre(A) BEL(A,X)
Pre(B) *vuoto*
Il problema principale di KQML era che (inizialmente) non aveva una semantica.

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS (FIPA)

Fondata con il fine di produrre degli standard per il software per agenti interagenti ed eterogenei e sistemi basati su agenti.
FIPA opera attraverso una collaborazione aperta internazionale di organizzazioni associate, aziende, centri di ricerca e università.
Tra le altre specifiche, la FIPA ha definito un linguaggio per comunicazione per agenti (**FIPA ACL**) simile a KQML. Prevede 22 atti comunicativi, come *inform, request, agree, query-if, ...*

CONFRONTO KQML E FIPA ACL

I due linguaggio sono molto simili rispetto ai loro concetti di base. Entrambi i linguaggi non sono vincolati ad un linguaggio per esprimere il contenuto.
La differenza principale è la semantica. Tuttavia, poiché la semantica si basa su stati mentali, le differenze potrebbero essere di secondaria importanza per chi programma agenti, in particolare se gli agenti non sono di tipo BDI.
Un'altra differenza è la mancanza di primitive di tipo facilitator in FIPA ACL. Introduce invece una standardizzazione del middleware molto più raffinata.

SINTASSI DI FIPA ACL

La sintassi è molto simile a quella di KQML. Esempio:

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

Le semantiche invece sono piuttosto differenti.

SEMANTICA DI FIPA ACL

Fondata sul Semantic Language (SL). SL è una logica multimodale quantificata con operatori modali:

- $B_{i\phi}$ i crede ϕ
- $U_{i\phi}$ i è incerto su ϕ ma pensa che ϕ è più probabile di $\neg\phi$

- $C_{i\phi}$ i desidera (choice, goal) che ϕ valga correntemente

Per permettere il ragionamento su azioni, l'universo del discorso coinvolge sequenze di eventi (actions). Operatori sono introdotti per ragionare sulle azioni:

- **Feasible**(a, ϕ) a può occorrere e, se occorre, ϕ sarà vera dopo tale occorrenza
- **Done**(a, ϕ) a è appena occorsa e ϕ è vera dopo tale occorrenza
- **Agent**(i,a) i è il solo agent che esegue l'azione a

Sulla base di belief, choice ed eventi viene definito il concetto di goal persistente (persistent goal) $PG_{i\phi}$. L'intenzione $I_{i\phi}$ è definita come un goal persistente che impone all'agente di agire.

La semantica degli atti comunicativi è specificata come un insieme di formule SL che descrivono le precondizioni di fattibilità (feasibility preconditions) e gli effetti razionali (rational effects).

Feasibility preconditions (**FP**): condizioni che devono valere per il mittente per eseguire in modo appropriato gli atti comunicativi.

Rational effects (**RE**): gli effetti che un agente può aspettarsi che occorrano come risultato dell'esecuzione di una azione (la ragione per la quale l'azione è selezionata). All'agente destinatario non è richiesto di garantire che occorra l'effetto atteso.

Per conformità di FIPA ACL si intende che quando un agente A esegue un atto comunicativo c, le FP(c) per A devono valere. I RE(c) non sono garantiti e sono irrilevanti al fine della conformità.

JADE

È conforme con le specifiche FIPA.

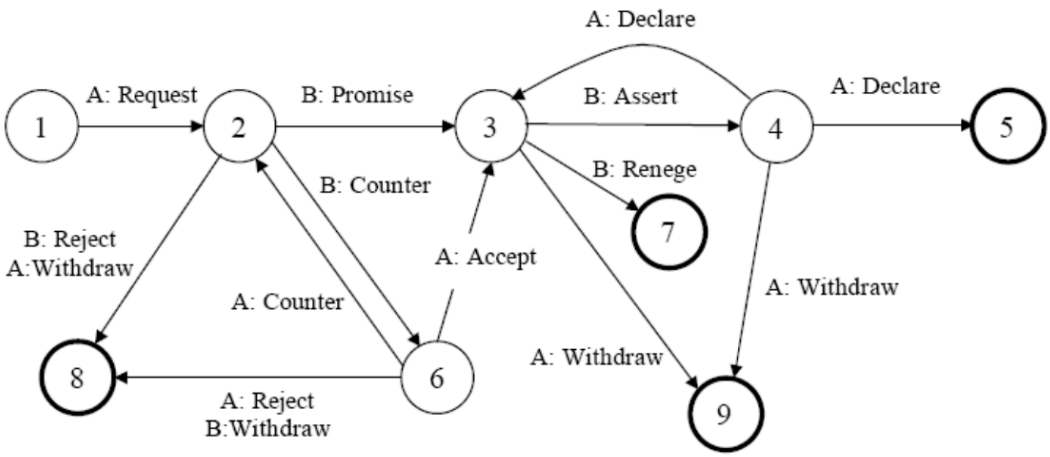
JADE è un middleware sviluppato da Telecom Italia per lo sviluppo di applicazioni distribuite multiagente basato su una architettura di comunicazione punto-a-punto (peer-to-peer).

PROTOCOLLI DI INTERAZIONE

Gli agenti non possono prendere parte a un dialogo semplicemente scambiandosi dei messaggi ACL. L'analisi di numerose conversazioni umane mostra che sono presenti degli schemi (pattern) ricorrenti di conversazioni. Un pattern ci garantisce un determinato risultato, pertanto lo adotto come protocollo di interazione quando ho intenzione di ottenere quel determinato risultato.

Un agente deve realizzare procedure di decisione che siano trattabili che gli permettano di selezionare e produrre i messaggi ACL che sono appropriati alle sue intenzioni: conversation policies o protocolli di interazioni.

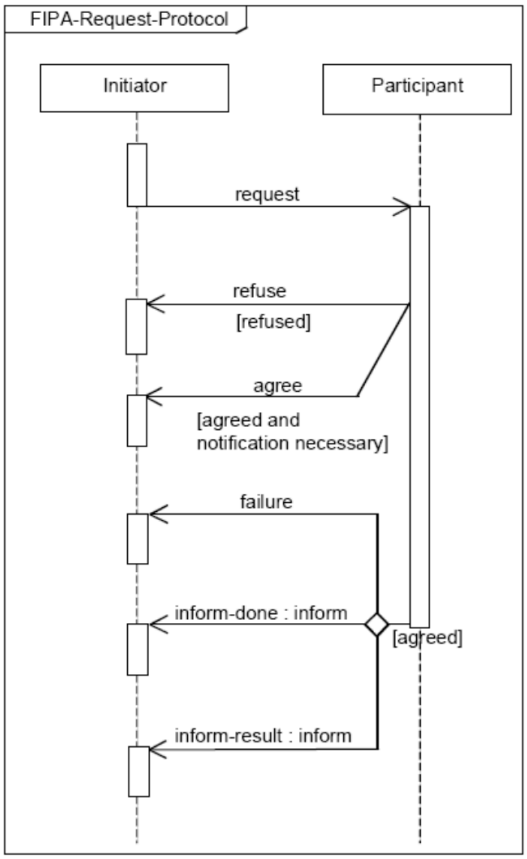
I protocolli sono solitamente modellati come macchine a stati finiti.



Sono stati proposti numerosi altri formalismi, alcuni:

- Reti di Petri (Petri nets)
- Definite Clause Grammars (DCG): utilizzate per la specifica di protocolli KQML. Le DCG sono estensioni delle grammatiche context free dove i non terminali possono essere composti da termini e il corpo di una regola può contenere procedure
- AUML: Le specifiche di FIPA definiscono i protocolli attraverso AUML, una estensione di UML per gli agenti

FIPA REQUEST INTERACTION PROTOCOL



CONTRACT NET PROTOCOL

Molti protocolli sono stati definiti per realizzare sistemi di agenti cooperanti, il più conosciuto e utilizzato per la soluzione e realizzazione di sistemi cooperanti è il Contract Net Protocol.

È modellato sul meccanismo di contrattazione utilizzato per governare gli scambi di merci e servizi. Un agente che desidera che un certo task sia risolto viene chiamato manager (o initiator), gli agenti che potrebbero risolvere il task si chiamano contractors (o participants).

Manager:

- Annuncia un task che necessita di essere risolto
- Riceve e valuta le offerte dei possibili contractor
- Sceglie un contractor e lo informa
- Riceve il risultato

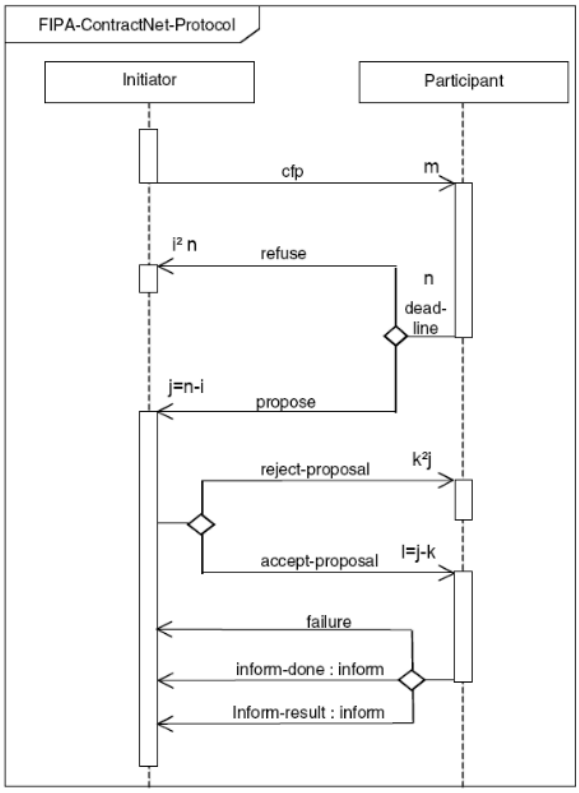
Contractor:

- Riceve gli annunci di task dal Manager
- Valuta il task
- Risponde declinando o facendo un'offerta
- Esegue il task se l'offerta è stata accettata
- Riporta il risultato

Un contractor per uno specifico task può agire a sua volta da manager sollecitando l'aiuto di altri agenti per risolvere parte di quel task.

Una deadline può essere fissata per effettuare offerte da parte dei contractor.

FIPA CONTRACT NET PROTOCOL



MODELLI DI COORDINAZIONE

Esistono modelli di coordinazione alternativi allo scambio di messaggi.
Un blackboard system è un approccio basato sull'architettura a lavagna dove una base di conoscenza comune, la "blackboard" è iterativamente aggiornata da sorgenti di conoscenza, partendo da una specifica di un problema alla soluzione del problema. La coordinazione quindi avviene attraverso la lavagna, quindi un agente aspetta ad eseguire un'operazione finché qualcuno non scrive le informazioni che lui necessita per l'operazione sulla lavagna.
Quindi l'ambiente per gli agenti diventa il piano su cui si coordinano: un agente fa un'azione perché reagisce all'ambiente.

SEMANTICA SOCIALE PER LA COMUNCIAZIONE

La semantica basata su stati mentali è adeguata per sistemi di agenti cooperanti ma presenta alcuni problemi quando il sistema è composto da agenti eterogenei e competitivi, infatti all'agente destinatario non è richiesto di garantire che occorra l'effetto atteso.
In pratica è impossibile fidarsi completamente di altri agenti o fare forti assunzioni a proposito dello stato interno degli agenti e sul loro modo di ragionare.
L'approccio sociale, invece, considera le conseguenze sociali di eseguire un atto comunicativo. Sostanzialmente riconosce che la comunicazione è inerentemente pubblica e che quindi dipende dal contesto sociale degli agenti. Ovvero la comunicazione è verificabile con fatti pubblici, oggettivi.

CARATTERISTICHE DI UN BUON ACL (secondo Singh)

Formal: chiarezza della specifica per guidare al meglio colui che realizza un sistema
Declarative: descrivere cosa (what) piuttosto che come (how)
Verifiable: determinare se un agente sta agendo in accordo con una data semantica
Meaningful: trattare la comunicazione secondo il suo significato e non come arbitrari token che devono essere ordinati in una qualche maniera
Esempi non "buoni":

- **English** is not good: non è formale
- **FSM** (macchina a stati finiti) non è good: non dichiarativo, non significativo
- **Mentalistic approach** is not good:non verificabile
- **Temporal logics** are not good: non significativo (se applicato ai token direttamente)

SEMANTICA PER ACL BASATA SUGLI STATI MENTALI E VERIFICA

Rispettare un protocollo: un protocollo è rispettato se e solo se l'interazione specificata/desiderata occorre durante l'esecuzione.

La semantica basata sugli stati mentali per un ACL è particolarmente adatta per ragionare in modo "statico" sulle proprietà del protocollo e sulle politiche adottate dagli agenti.

MA da un punto di vista di un agente che partecipa all'interazione, come essere sicuri che gli altri agenti rispettino la semantica specificata?

Non è possibile fare introspezione sugli agenti, solitamente lo stato mentale degli agenti non è accessibile: come possiamo dimostrare che un agente crede a ciò che dice se non è un agente BDI?

Quindi per verificare un protocollo: si necessita di una semantica basata su fatti "osservabili", l'agente che interagisce deve poter comprendere se il protocollo viene rispettato osservando l'interazione (osservando fatti oggettivi) e non basandosi sugli stati mentali.

Una semantica che sia verificabile sia da un partecipante sia da una terza parte che deve monitorare l'interazione. Una semantica che sia adatta ad un sistema aperto. Una semantica che renda osservabile la violazione di una specifica da parte di un partecipante all'interazione, senza che si debba prendere in esame la natura dell'agente o eseguire introspezione dello stato mentale dell'agente.

SEMANTICA SOCIALE

L'approccio è basato sui **practical commitments** (impegni) tra agenti: un agente (il debtor) si impegna verso un altro agente (il creditor) a rendere vero un qualche fatto o ad eseguire una qualche azione.

Secondo l'approccio sociale, i vari atti illocutori possono essere visti in termini di commitment (impegni) sociali che i partecipanti creano (questo è intuitivo per atti comunicativi come promise, meno per altri tipi di comunicazioni).

Singh propone una semantica sociale basata sul punto di vista del filosofo Habermas. Il filosofo Habermas propone tre livelli di semantica per ogni atto comunicativo: per esempio, informando j che p, i si impegna verso j che p è vero (objective claim), che egli crede p (subjective claim) e che i ha ragione di credere p (practical claim).

COMMITMENTS

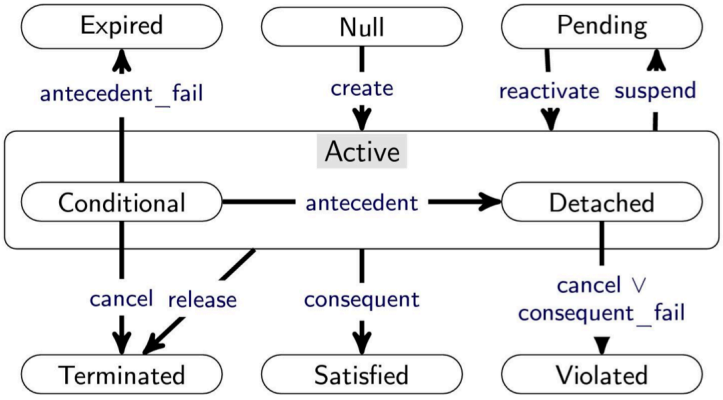
Cohen e Levesque: "Intention is choice with commitment (individuali)" (1990), le intenzioni sono definite come goal persistenti, le intenzioni sono alla base dei commitment.

Cohen e Levesque: le intenzioni condivise sono la base per un commitment sociale.
Castelfranchi: è d'accordo con Cohen e Levesque sui commitment individuali MA in disaccordo sui commitment sociali (1997, 1998).
Castelfranchi nel definire i commitment sociali enfatizza la loro natura distribuita e normativa:

- Un commitment sociale ha natura normativa e genera diritti
- A differenza di Cohen e Levesque, cancellare o ritirare un commitment sociale non implica una informazione verso la controparte
- Cancellare o ritirare un commitment viola un'obbligazione, frustra delle aspettative e dei diritti che l'agente ha creato

SINGH'S COMMITMENTS

Include una nozione di contesto sociale nella definizione di commitment (il team in cui l'agente partecipa e in cui comunica). Permette metacommitment (ovvero commitment di commimtment) per catturare un'ampia varietà di relazioni legali e sociali (come gli impegni, le richieste, i diritti, i privilegi, i poteri) in un unico framework. Il comportamento degli agenti è influenzato dai commitment perché l'assunzione è che gli agenti rispettino gli impegni presi. I commitment sono manipolabili dagli agenti.
C(debtor, creditor, antecedent, consequence): il debtor è socialmente legato al creditor a rendere vera la condizione conseguente se la condizione antecedente è vera. Ad esempio C(Bookie, Alice, $12, BeatingtheOdds): significa che Bookie si impegna nei confronti di Alice che se lei paga $12, allora Bookie le spedirà il libro Beating the Odds.
Detach: quando l'antecedente del commitment diventa vero, ovvero quando si verifica la condizione per cui il commitment diventa un obbligo. Se Alice effettua il pagamento, allora il commitment è detached: C(Bokkie, Alice, T, BeatingtheOdds).
Discharge: se Bookie spedisce il libro, il commitment è discharged, è compiuto e quindi decade.
Ciclo di vita di un commitment:



OPERAZIONI SUI COMMITMENT

create(x,y,r,u) è eseguita da x e causa C(x,y,r,u)
cancel(x,y,r,u) è eseguita da x e rimuove C(x,y,r,u)
release(x,y,r,u) è eseguita da y e rimuove C(x,y,r,u)
delegate(x,y,z,r,u) è eseguita da x e causa C(z,y,r,u)
assign(x,y,z,r,u) è eseguita da y e causa C(x,z,r,u)

POSTULATI SUI COMMITMENT

Discharge: $u \rightarrow \neg C(r, u)$
Detach: $C(r \wedge s, u) \wedge r \rightarrow C(s, u)$
Augment: da "$C(r, u) \wedge r$ infers s" deriviamo $C(s, u)$
L-disjoin: $C(r, u) \wedge C(s, u) \rightarrow C(r \vee s,u)$
R-conjoin: $C(r, u) \wedge C(r, v) \rightarrow C(r, u \wedge v)$
Consistency: $\neg C(r, false)$
Nonvacuity: da "r infers u" deriviamo $\neg C(r, u)$
Weaken: $C(r, u \wedge v) \wedge \neg u \rightarrow C(r, u)$

PROTOCOLLI DI INTERAZIONE CON COMMITMENT

Secondo Singh, i protocolli possono essere specificati come un insieme di commitment piuttosto che come macchine a stati finiti.
Gli agenti giocano ruoli diversi nella società e i ruoli definiscono l'associazione con gli impegni sociali e le obbligazioni verso gli altri ruoli. Per esempio, A onorerà una quotazione se B risponderà entro la scadenza.
In generale, gli agenti possono gestire i propri impegni (commitment) manipolandoli o cancellandoli. Poiché i requisiti di un protocollo vengono espressi solamente attraverso commitment, gli agenti possono essere conformi sulla base della loro comunicazione (non è necessaria conoscere la loro implementazione).
Yolum e Singh introducono i protocolli basati su commitment, nel 2001, come insiemi di azioni il cui significato espresso in termini di effetti sullo stato sociale sono d'accordo tutti gli agenti interagenti. Quindi il protocollo è un accordo su un insieme di azioni il cui significato è espresso tramite i commitment. In particolare gli effetti sono espressi in termini di operazioni sui commitment (create, delete, release, delegate, assign, discharge). L'idea è di catturare la relazione "count-as" che descrive il significato istituzionale (nella società) dell'azione.
Il solo vincolo che un protocollo a commitment deve soddisfare perché una interazione sia di successo è che tutti i commitment siano alla fine discharged.
Il protocollo a commitment deve associare un significato alle azioni in termini di manipolazioni di commitment. Manipolazioni di commitment che portano impegni

su azioni, su fatti verificabili che possano appunto essere verificate dagli agenti interagenti.

RUN DEL PROTOCOLLO A COMMITMENT

Un run (di successo) del protocollo è una sequenza (finita) di azioni che portano in uno stato in cui tutti i commitment sono fulfilled, ossia non ci siano commitment pendenti (commitments C(x,y,$\top$,p) dove p non è stata resa vera).
Data la precedente specifica delle azioni, posso costruire delle sequenze che permettono di raggiungere uno stato in cui non ci siano commitment pendenti. Queste sono i run del protocollo.
Verifica: data una sequenza di azioni eseguite dagli agenti, posso verificare se essa sia o non sia una sequenza ammessa dal protocollo (un run del protocollo).

JADE

JADE implementa tutti i servizi base e l'infrastruttura di un'applicazione multi agente distribuita. È realizzato in Java. L'obiettivo di JADE è quello di nascondere ai programmatori tutti gli aspetti della standardizzazione di FIPA, renderlo trasparente al programmatore. JADE è un'implementazione di FIPA.
JADE offre una piattaforma che mette in contatto vari container anche localizzati su device diversi e che ospitano i vari agenti dell'applicazione.

MODELLAZIONE LOGICA DI AGENTI BDI

Secondo Wooldridge e Ciancarini i metodi formali giocano tre ruoli: **specificare** i sistemi, **programmare direttamente** i sistemi, **verificare** sistemi.
Due approach logici per **specificare** agenti: Cohen-Levesque's intention logic e Rao-Georgeff's BDI logics.

COHEN-LEVESQUE'S INTENTION LOGIC

La logica riguarda principalmente il "rational balance" relativo a beliefs, goals, plans, intentions, commitments, e azioni di agenti autonomi. Seguendo Bratman, Cohen e Levesque ritengono che il comportamento razionale dovrebbe essere analizzato in termini di beliefs, desires and intentions (BDI).
Le intenzioni dovrebbero soddisfare le seguenti proprietà:

- Le intenzioni pongono dei problemi per gli agenti, che devono determinare modi di soddisfarle.
- Le intenzioni forniscono un "filtro" per adottare altre intenzioni che non devono entrare in conflitto.

- Gli agenti "tracciano" il successo delle loro intenzioni, e sono disposti a tentare di nuovo se il loro tentativo fallisce.
- Gli agenti credono che le loro intenzioni siano possibili.
- Gli agenti non credono che non riusciranno a soddisfare le loro intenzioni.
- In certe circostanze, gli agenti credono che riusciranno a soddisfare le loro intenzioni.
- Gli agenti non si aspettano tutti i side-effects delle loro intenzioni (ciò che uno intende è, approssimativamente, un sottoinsieme di ciò che uno sceglie).

La logica ha quattro operatori modali:

- (BEL i φ) agent i believes φ
- (GOAL i φ) agent i has goal (desire) of φ
- (HAPPENS α) action α will happen next
- (DONE α) action α has just happened

La semantica è data mediante mondi possibili. I mondi sono connessi mediante relazioni BEL, GOAL, ... , mentre il ragionamento in un mondo è basato sulla logica temporale LTL. Le formule sono valutate rispetto a qualche mondo w_i, e un indice t_j in quel mondo, che rappresenta un punto nella sequenza degli eventi.
La semantica di BEL e GOAL è data assegnando ad ogni agente una belief accessibility relation, e una goal accessibility relation.
La belief accessibility relation è KD45 (weak S5). La goal accessibility relation è KD.
Si assume che la goal relation di ogni agente sia un sottoinsieme della sua belief relation.

Modalità derivate

Gli operatori standard di LTL nel tempo futuro **G** (always) and **F** (sometime), possono essere definiti come abbreviazioni

Fφ ≡ ∃α. (HAPPENS α;φ?) esiste una azione α tale che, dopo la sua esecuzione vale φ. ? è l'operatore di test.

Gφ ≡ ¬**F**¬φ

(LATERφ)≡ ¬φ ∧ **F**φ

E' possibile derivare anche un operatore di precedenza temporale (BEFORE φ ψ), ossia φ vale prima di ψ:

(BEFORE φ ψ) ≡ ∀α. (HAPPENS α;ψ?) ⇒ ∃β. (β ≤ α) ∧ (HAPPENS β;φ?)

Achievement goals

Achievement goals sono quelli che l'agente crede falsi, ma desidera che in futuro diventino veri.

(A-GOAL i φ) ≡ (GOAL i (LATER φ)) ∧ (BEL i ¬φ)

Ad esempio i crede che φ sia falso, ma desidera che in futuro diventi vero.

Goal persistenti

Un agente ha il persistent goal ϕ se:

- Ha un goal che ϕ prima o poi diventerà vero, e crede che ϕ attualmente non lo sia (achievement goal)
- Prima di abbandonare il goal, una delle seguenti condizioni deve essere vera:
  - l'agente crede che il goal sia stato soddisfatto;
  - l'agente crede che il goal non sarà mai soddisfatto.

$$(\text{P-GOAL}i\phi) \equiv (\text{GOAL}i(\text{LATER}\phi)) \wedge (\text{BEL}i\neg\phi) \wedge [\text{BEFORE}((\text{BEL } i\ \phi) \vee (\text{BEL } i\ G\ \neg\phi)) \neg(\text{GOAL } i\ (\text{LATER}\ \phi))]$$

Intenzione di eseguire un'azione

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

L'agente i intende eseguire l'azione α se ha un goal persistente di raggiungere uno stato in cui lui aveva appena creduto di stare per eseguire α, e subito dopo α viene eseguita. Il criterio che l'agente sia committed a credere di stare per eseguire l'azione α, evita che l'agente sia tenuto ad eseguire l'azione accidentalmente o senza pensarci.

RAO-GEORGEFF'S BDI LOGIC

Tre modalità: BEL, GOAL (Desire) e INTEND (**BDI**).

I mondi sono strutture temporali branching time invece di essere linear time.

Le loro formule temporali sono simili a quelle di CTL*.

Il linguaggio estende le formule di CTL* con:

- BEL(ϕ), GOAL(ϕ), INTEND(ϕ)
- succeeds(e), fails(e), does(e), succeeded(e), failed(e), done(e), dove e è un evento.

Inoltre ad esempio succeeds(e) può essere definita come **AX**succeeded(e) (l'evento è valido in tutti cammini uscenti dall'istante di tempo in cui la formula è valutata).

Commitments

Con questo linguaggio è possibile descrivere commitment diversi.

Ad esempio, un agente *blindly committed* (*fanatical*) mantiene le sue intenzioni finché non arriva a credere di averle soddisfatte.

$$\text{INTEND}(\text{AF}\phi) \Rightarrow \text{A}(\text{INTEND}(\text{AF}\phi)\ \text{U}\ \text{BEL}(\phi))$$

Con *single-minded* commitment, un agente mantiene le sue intenzioni finché crede che queste possano essere realizzabili.

$$\text{INTEND}(\text{AF}\phi) \Rightarrow \text{A}(\text{INTEND}(\text{AF}\phi)\ \text{U}\ (\text{BEL}(\phi) \vee \neg\text{BEL}(\text{EF}\phi)))$$

Questa formulazione può essere modificata definendo un agente *open-minded* come un agente che mantiene le sue intenzioni finché queste intenzioni sono ancora i suoi goal.
$INTEND(AF\phi) \Rightarrow A(INTEND(AF\phi)\ U\ (BEL(\phi) \vee \neg GOAL(EF\phi)))$

Varianti della logica

Rao e Georgeff hanno analizzato diverse relazioni di accessibilità fra belief, desire, e intention. Per esempio, considerano il caso in cui una relazione sia un sottoinsieme di un'altra, definendo anche una relazione di sub-world.

MODEL CHECKING

Metodo formale (molto usato nella pratica) per verificare le proprietà di sistemi di agenti o, più in generale di sistemi concorrenti.
Il model checking è basato su un approccio semantico: dato un modello M in una logica L, e una formula φ di L, determinare se φ è valida in M.

Le proprietà da verificare sono basate su logiche temporali.
Il model checking è usato prevalentemente per verificare le proprietà di sistemi non terminanti (concorrenti).
Il comportamento di un sistema π può essere formulato come un transition system (tipo un automa) consistente di un insieme di stati, un insieme di transizioni fra gli stati e una funzione che che associa ad ogni stato un insieme di proposizioni vere in quello stato.
Il transition system può essere visto come una Kripke structure M_π, ossia un modello della logica temporale (modale). Questo transition system descrive tutte le possibili computazioni di un sistema.

Model checking per LTL

Supponiamo di avere un sistema π formulato come un transition system (che spiega tutti i comportamenti del sistema), e una formula LTL φ che descrive una proprietà che vogliamo verificare. Per mostrare che φ vale per π, dobbiamo dimostrare che φ è vera per ogni esecuzione di π. Siccome sappiamo che un modello in LTL è una sequenza infinita di stati, dimostrare che φ è vera per ogni esecuzione di π, significa che ogni cammino infinito di π è un modello di φ.

La dimostrazione che una formula φ è valida, ossia è vera per tutti i cammini infiniti del modello, viene fatta per **refutazione**: dimostrare che $\neg\varphi$ è insoddisfacibile nel modello, ossia non esiste nessuna computazione che soddisfi $\neg\varphi$. In altre parole, se si trova un cammino che soddisfa $\neg\varphi$, questo costituisce un controesempio che contraddice la validità di φ.

Automi di Büchi e Model Checking

Automa di Büchi: automa che accetta stringhe infinite.

Ha esattamente la stessa struttura di un tradizionale automa a stati finiti, ma accetta stringhe infinite. L' automa $< \Sigma, S, \delta, S_0, F >$ consiste di un alfabeto Σ, un insieme di stati S, una funzione di transizione $\delta \subseteq S \times \Sigma \times S$, un insieme di stati iniziali S_0 e un insieme di stati di accettazione F.

Informalmente, una stringa infinita (run) w è accettata dall'automa se è compatibile con δ (è un cammino infinito nel grafo dell'automa) e almeno uno stato in F appare infinite volte in w.

Uno dei vantaggi principali di usare gli automi per il model checking è che il sistema modellato e le proprietà da verificare sono rappresentati allo stesso modo, ossia con automi di Büchi. Infatti il transition system di un sistema π, ossia il modello, può essere visto come un automa di Büchi $B(\pi)$ i cui stati sono tutti di accettazione.

Inoltre, esiste un algoritmo per tradurre una formula LTL φ in un automa di Büchi $B(\varphi)$ che accetta esattamente le sequenze infinite che sono modelli di φ.

Procedura per il Model Checking

- siano dati il transition system π e la formula LTL φ da verificare
- costruire i due automi di Büchi $B(\pi)$ e $B(\neg\varphi)$
- costruire l'automa di Büchi che accetta l'intersezione dei linguaggi accettati da $B(\pi)$ e $B(\neg\varphi)$ (il prodotto dei due automi di Büchi)
- ogni run dell'intersezione è sia un run infinito di π sia un modello di $\neg\varphi$
- se l'intersezione è vuota, allora φ vale per π, altrimenti un run dell'intersezione fornisce un controesempio

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

Planning

Oltre alla verifica di proprietà, il model checking può essere utilizzato per altri tipi di ragionamento, come, ad esempio, la **pianificazione** (con cammini infiniti).

Dato l'automa che rappresenta un sistema computazionale π e una formula φ che rappresenta il goal, tutti i run infiniti dell'automa prodotto di $B(\pi)$ e $B(\varphi)$ soddisfano φ e quindi rappresentano un piano sequenziale.

Se si vuole ottenere un piano di lunghezza finita con questo approccio, è sufficiente aggiungere nello stato finale un loop fittizio.

Spin

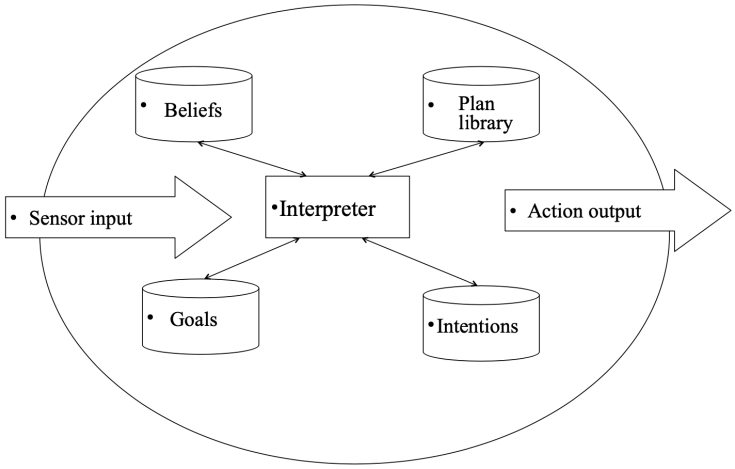
È un tool per la verifica di sistemi concorrenti basato su tecniche di model checking. La specifica del modello è data nel linguaggio Promela, che consente di definire sistemi distribuiti mediante un insieme di processi formulati in un codice pseudo C, che consente l'uso di primitive di sincronizzazione e scambio di messaggi.
La proprietà da verificare è una formula in linear time temporal logic (LTL).

Altri approcci

Sono stati sviluppati molti altri model checker basati su approcci diversi. In particolare l'approccio per verificare formule CTL, che si basa sul ragionare direttamente sul transition system, decomponendo la struttura di Kripke in componenti fortemente connessi. È un approccio più efficiente, ma in realtà le formule LTL e CTL non sono confrontabili e quindi nemmeno ha senso stare a confrontarli.

PROCEDURAL REASONING SYSTEM (PRS)

Una delle prime architetture per agenti con uso del paradigma BDI (sviluppato da Georgeff e colleghi). Usata in molte applicazioni multi-agenti come il controllo del traffico aereo, sistemi diagnostici, ecc.



Gli input del sistema sono eventi, ricevuti attraverso una coda di eventi. Gli eventi possono essere:

- Esterni, percepiti dall'ambiente
- Interni, come aggiunta o cancellazione di belief o goal

Gli output sono azioni esterne o interne, che sono eseguite dall'interprete.
I belief sono rappresentati come fatti del Prolog: in pratica come atomi della logica del prim'ordine.
In PRS gli agenti non pianificano, ma fanno riferimento ad una libreria di piani predefiniti dal programmatore. I componenti di un piano sono:

- nome

- condizione di invocazione: triggering event (goal)
- precondizioni: le condizioni che devono valere prima di avviare l'esecuzione del piano
- body: è una sequenza di simple plan expressions, ossia: una azione atomica o un sottogoal

Per raggiungere un dato goal, l'agente formula l'intenzione di raggiungere questo obiettivo, cioè sceglie un piano applicabile "triggered" dal goal. Questo piano diventa una intenzione, ed è aggiunto alla intention structure. A ogni passo del loop principale, l'interprete sceglie un piano (parzialmente eseguito) nella intention structure, e ne esegue un passo. Se ci sono molte opzioni disponibili, l'interprete può scegliere quella con massima utilità, o entrare in un ragionamento di metalivello usando "metalevel plans".

I passi principali del control loop del PRS sono i seguenti:

- aggiorna belief e goal secondo gli eventi nella event queue
- i cambiamenti dei goal e belief "trigger" diversi piani
- uno o più dei piani applicabili sono scelti e messi nella intention structure
- scegliere una intenzione (task) dalla intention structure
- eseguire un passo di quel task. Questo può risultare in:
  - esecuzione di una azione primitiva
  - scelta di un nuovo subgoal, che è "posted" nella event queue.

Dato che l'interprete a ogni iterazione sceglie una intenzione(task) e ne esegue un passo, l'esecuzione dei task può essere "interleaved", come in un sistema multithreaded.

Per tenere traccia di questo, ogni intenzione (task) è implementata, come al solito, come uno stack di frames, che descrive uno stato intermedio dell'esecuzione del task.

AGENTSPEAK(L)

Linguaggio di programmazione per agenti BDI proposto da Rao nel '96, scritto e interpretato in modo simile ai programmi logici basati su clausole di Horn. Ha dato origine al linguaggio JASON.

AGENT-ORIENTED PROGRAMMING (AOP)

Nuovo paradigma di programmazione che promuove una visione sociale della computazione, in cui più agenti interagiscono uno con l'altro. Proposto in un articolo di Shoham, che presenta:

- Un linguaggio formale per descrivere stati mentali
- Un linguaggio di programmazione AGENT-0 per definire agenti

Categorie mentali

Ci sono due categorie mentali principali: belief and obligation (o commitment). Decision (o choice) è trattata come una obligation a se stesso. Una ulteriore categoria, che non è un costrutto mentale, è capability.

Le formule fanno riferimento esplicitamente al tempo. Viene adottato un semplice linguaggio basato sugli istanti di tempo.

Belief

$B_a^t\phi$: agent a believes ϕ at time t.

For instance:

$B_a^3 B_b^{10}\text{like(a,b)}^7$ means that at time 3 agent a believes that at time 10 agent b will believe that at time 7 a liked b.

Obligation

$OBL^t_{a,b}\phi$

means that at time t agent a is obligated, or committed, to agent b about ϕ. ϕ can be a fact representing an action.

Decision (**choice**)

Decision is defined as commitment to oneself:

$DEC^t_a\phi =_{def} OBL^t_{a,a}\phi$

Capability

The fact that at time t agent a is capable of ϕ is represented by

$CAN^t_a\phi$

Example: $CAN^5_{robot}\text{open(door)}^8$

AGENT-0

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

Facts: (t atom) atom holds at time t

Private actions: (DO t p-action)

Communicative actions: (INFORM t a fact) at time t inform agent a that fact

(REQUEST t a action)

AGENT-0 è implementato come una estensione al LISP.

Ogni agente in AGENT-0 ha 4 componenti:

- A set of capabilities (things the agent can do)
- A set of initial beliefs
- A set of initial commitments (things the agent will do)
- A set of commitment rules

I commitment rules hanno la seguente struttura:

(COMMIT message-cond mental-cond (agent action)*)

A mental condition è una combinazione logica di mental patterns. A mental pattern is one of: (B fact) or ((CMT a) action). (CMT means “committed").
A message condition è una combinazione logica di message patterns: (From Type Content) dove From è il nome del mittente, Type è il tipo della azione comunicativa (INFORM, ...) e Content è un fatto o una azione che dipende dal tipo.
(agent action)* è una lista di coppie.
Beliefs, commitments, e capabilities di un agente sono rappresentati ciascuno da un database. Beliefs e commitments possono cambiare ad ogni passo dell’esecuzione di un programma, mentre le capabilities sono fisse.
L’interprete esegue il seguente ciclo:

- Leggi i messaggi correnti, e aggiorna i tuoi belief e commitment (valutando le commitment rules)
- Esegui i commitment per il tempo attuale, eventualmente risultando in un cambiamento di belief

I beliefs sono aggiornati o come risultato di essere informati, o come risultato di eseguire una azione privata.
Per ogni commitment rule:
(COMMIT messagecond mentalcond ($agent_i$ $action_i$)*),
se la message condition vale per il nuovo messaggio in arrivo;
se la mental condition vale per il nuovo stato mentale;
per tutti gli i, l’agente è capace di eseguire l’$azione_i$, allora per tutti gli i, prendere un commitment con $agent_i$ per eseguire $action_i$ al tempo specificato.

PROLOG

Esempio di linguaggio logico per programmare sistemi di agenti. I programmi in Prolog sono un insieme di clausole di Horn della logica classica.

GOLOG

Linguaggio di programmazione basato sul calcolo delle situazioni, usato per programmare robot di alto livello e agenti intelligenti. Le azioni primitive in GOLOG sono specificate dandone le precondizioni e effetti (rappresentati come successor state axioms). GOLOG permette di definire azioni complesse usando la abbreviazione Do(δ, s, s') dove δ è una espressione di azione complessa; intuitivamente Do(δ, s, s') è vera quando la situazione s' è una situazione terminante di una esecuzione di δ iniziata nella situazione s.
Primitive actions: $Do(a, s, s') =_{def} Poss(a, s) \wedge s' = do(a, s)$
Test actions: $Do(\phi?, s, s') =_{def} \phi \wedge s = s'$
Sequence: $Do([\delta 1; \delta 2], s, s') =_{def} \exists s''. Do(\delta 1, s, s'') \wedge Do(\delta 2, s'', s')$

Nondeterministic choice: $Do((\delta 1 \mid \delta 2), s, s') =_{def} Do(\delta 1, s, s') \vee Do(\delta 2, s, s')$

Nondeterministic choice of action: $Do((\pi x)\delta(x), s, s') =_{def} \exists x.\ Do(\delta(x), s, s')$

GOLOG (algol in logic) è stato progettato come un linguaggio di programmazione logica per domini logici, cercando di mescolare lo stile di programmazione dell'ALGOL con la logica. Prende in prestito dall'ALGOL molti costrutti standard della programmazione come sequenza, condizionale, procedure ricorsive e loop.

Per esempio: if ϕ then $\delta 1$ else $\delta 2 =_{def} [\phi?; \delta 1] \mid [\neg\phi?; \delta 2]$

Eseguire un programma significa stabilire la seguente derivazione:

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

dove S_0 è la situazione iniziale.

Un programma eseguito con successo restituisce un legame per s:

$$s = do(a_n, ..., do(a_2, do(a_1, S_0))....)$$

L'interprete del GOLOG è un theorem prover general-purpose (in generale per la logica del second'ordine).

Il GOLOG è stato descritto in un articolo del 1994, in cui viene presentata una implementazione prototipale in Prolog.

Da allora il GOLOG è stato esteso in vari modi. Ad esempio:

CONGOLOG: incorpora la concorrenza, trattando processi concorrenti.

IndiGolog: i programmi possono essere eseguiti incrementalmente per permettere azioni interleaved, planning, sensing.

CONCURRENT METATEM

È un linguaggio basato sulla esecuzione diretta di formule temporali.

Un sistema Concurrent METATEM contiene un insieme di agenti che possono comunicare fra di loro via asynchronous broadcast message passing.

Ogni agente è programmato dando una specifica del suo comportamento mediante logica temporale. Questa specifica può essere eseguita direttamente.

La logica usata dal Concurrent METATEM è la logica temporale linear time LTL estesa nel passato oltre che nel futuro.

Futuro:

- $\bigcirc\varphi$ — φ must be satisfied in the next state
- $\varphi\ U\ \psi$ — φ will be true until ψ
- $\Diamond\varphi$ — φ must be satisfied at *some* state in the future
- $\Box\varphi$ — φ must be satisfied at *all* states in the future

Passato:

- $\varphi\ \mathcal{S}\ \psi$ — φ has been true since ψ
- $\bullet\varphi$ — φ was satisfied in the previous state

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

□(past and present formula ⇒ present or future formula) dove la parte destra è vincolata a essere una disgiunzione o una formula sometime.

Il linguaggio fornisce due meccanismi ortogonali per rappresentare la scelta:

- indeterminatezza statica, con l'operatore classico ∨
- indeterminatezza temporale, con l'operatore sometime ◊ (però, dato ◊ϕ, l'esecuzione cerca di soddisfare ϕ appena possibile).

L'interprete esegue in continuazione i seguenti passi:

- Verifica quali regole hanno gli antecedenti soddisfatti
- Congiungi i conseguenti di queste regole
- Riscrivi questa congiunzione in forma disgiuntiva e scegli uno di questi disgiunti da eseguire
- Se si trova una contraddizione, può essere possibile fare backtrack a una scelta precedente

Il meccanismo base di comunicazione tra agenti è il broadcast message-passing, modellato con tre categorie di predicati:

- Predicati dell'ambiente: che rappresentano i messaggi in entrata
- Predicati component: che rappresentano quelli che escono
- Predicati interni

Ogni agente ha un'interfaccia che definisce come l'agente può interagire con l'ambiente. L'interfaccia consiste di un ID per l'agente, una lista di predicati dell'ambiente (i messaggi che l'agente riconosce) e una lista di predicati component (i messaggi che l'agente può mandare).

ARCHITETTURE REATTIVE

Ci sono molti problemi irrisolti associati all'**IA simbolica** (classica).

Questi problemi hanno portato alcuni ricercatori a discutere la realizzabilità dell'intero paradigma, ed a sviluppare architetture **reattive**.

Sebbene uniti dal credere che quelle assunzioni che sostengono l'IA classica siano sbagliate in qualche senso, i progettisti di agenti reattivi usano molte tecniche diverse tra di loro.

Approccio di Brooks

Brooks ha avanzato tre tesi:

- Un comportamento intelligente può essere generato senza rappresentazione esplicita del tipo di ciò che è proposto dall'IA simbolica
- Un comportamento intelligente può essere generato senza ragionamento esplicito del tipo di ciò che è proposto dall'IA simbolica
- L'intelligenza è una proprietà emergente di certi sistemi complessi

Brooks identifica due idee chiave:

- **Collocazione e personificazione**: l' intelligenza reale è situata nel mondo, non in sistemi senza sostanza come i "theorem provers" o "sistemi esperti"
- **Intelligenza e apparenza**: il comportamento 'Intelligente' è generato come un risultato di una interazione con l'ambiente. L'intelligenza sta "negli occhi di chi guarda", non è una proprietà innata e isolata

Per illustrare le sue idee, Brooks fa uso della **subsumption architecture**.
Una subsumption architecture è una gerarchia di task che eseguono behaviours, ogni behaviour è una struttura a regole molto semplici. Ogni behaviour 'compete' con altri per esercitare il controllo sugli agenti. Gli strati più bassi rappresentano tipi di behaviour più primitivi (come evitare ostacoli) e hanno la precedenza su strati più in alto nella gerarchia. I sistemi risultanti sono estremamente semplici in termini della quantità di computazione che fanno.
La scelta di una azione è realizzata attraverso un insieme di behaviours. Un behaviour è una coppia (c,a), dove c è un insieme di percezioni dette condizioni, e a è una azione. Un behaviour (c,a) scatta quando la funzione "*see*" restituisce una percezione p, tale che $p \in c$.
Associata a un insieme di behaviour rules c'è una relazione di inhibition <.
Leggiamo b1 < b2 come "b1 inibisce b2", cioè, b1 è più basso nella gerarchia di b2, e quindi ha la priorità su b2.

Vantaggi degli agenti reattivi:

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

Limiti degli agenti reattivi:

- Agenti senza modello dell'ambiente devono avere sufficienti informazioni disponibili localmente
- Se le decisioni sono basate sull'ambiente locale, come può l'agente tenere conto dell'informazione non-locale (ha una visione a breve termine)
- Difficile realizzare agenti reattivi che apprendono
- Visto che i behaviour emergono da interazioni fra componenti più ambiente, è difficile vedere come ingegnerizzare agenti specifici (non esistono metodologie generali)
- È difficile definire agenti con un gran numero di behaviour (le dinamiche delle interazioni diventano troppo complesse da capire)

ARCHITETTURE IBRIDE

I sistemi ibridi cercano di combinare approcci classici e reattivi.

Un approccio è quello di costruire un agente con due (o più) sottosistemi:

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

Spesso, ai componenti reattivi viene data precedenza rispetto a quelli deliberativi.

Questo tipo di struttura porta naturalmente all'idea di una architecture a strati (layered).

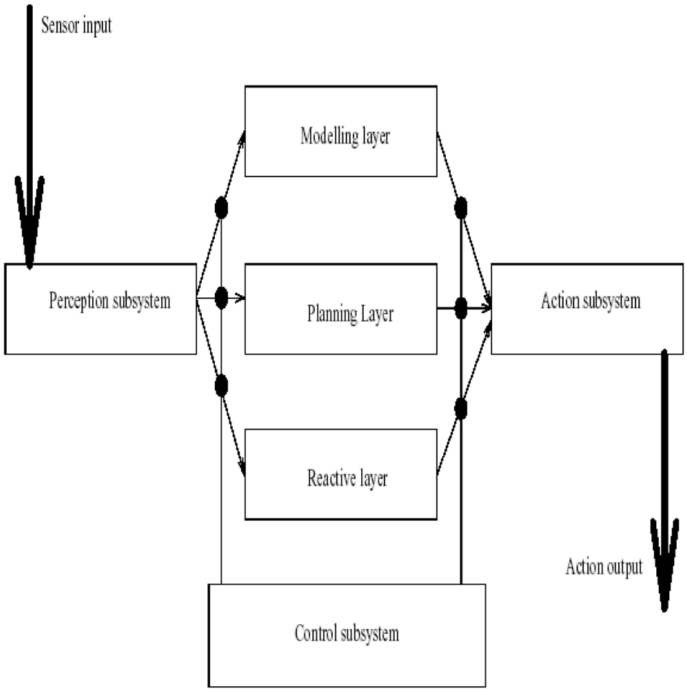
Un problema importante in queste architetture è in che tipo di schema di controllo inserire i sottosistemi dell'agente per trattare le interazioni fra i vari strati.

Sono stati proposti:

- **Horizontal layering**: Tutti gli strati sono direttamente connessi all'input e all'output: ogni strato opera come un agente, producendo suggerimenti su quale azione eseguire. Problema: introduce collo di bottiglia nel sistema di controllo centrale
- **Vertical layering**: Input dei sensori e output delle azioni sono gestiti al massimo da uno strato ciascuno. Problema: se uno strato qualsiasi genera un errore ne risente tutta la struttura

Architettura TOURINGMACHINES

L'architettura di tipo orizzontale che consiste di sottosistemi di percezione e azione, che interfacciano direttamente con l'ambiente dell'agente, e tre control layers, inseriti in un control framework, che media fra gli strati.

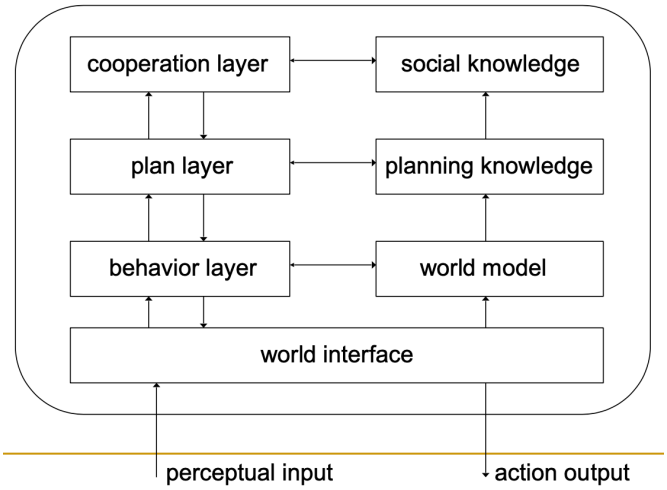


Lo strato **reactive** è implementato come un insieme di regole situation-action.

Il **planning layer** costruisce piani e sceglie azioni da eseguire, in modo da raggiungere i goal dell'agente. Il **modelling layer** contiene la rappresentazione simbolica degli 'stati cognitivi' di altre entità nell'ambiente dell'agente. I tre layers comunicano tra di loro e sono inseriti in un "control framework", che usa "control rules".

Architettura InteRRaP

Architettura verticale a due passate.



Sempre tre livelli, simili ai tre dell'architettura TOURINGMACHINES.

TEORIA DEI GIOCHI

Abbiamo un insieme di agenti egoisti ovvero che partecipano per conto proprio cercando di raggiungere i propri obiettivi. Uno dei problemi principali consiste nel trovare un meccanismo per prendere delle decisioni razionali.

Nel caso della teoria dei giochi le mosse dei giocatori sono simultanee. Per semplicità tratteremo solo giochi che consistono in una sola mossa.

In realtà la teoria dei giochi è usata in situazioni decisionali anche molto serie, come procedimenti per bancarotta, sviluppo di prodotti e relativo prezzo sul mercato, difesa nazionale ecc.

La teoria dei giochi può essere utilizzata in almeno due modi:

- Progettazione di agenti: la teoria dei giochi può analizzare le decisioni dell'agente e calcolare l'utilità attesa per ognuna di esse
- Progettazione di meccanismi: quando un ambiente è popolato da più agenti potrebbe essere possibile definirne le regole (ovvero il gioco a cui gli agenti devono partecipare) in modo da massimizzare il bene comune

Un gioco è definito dai seguenti componenti:

- **Giocatori** o agenti che prenderanno decisioni. Noi vedremo solo giochi a due giocatori

- **Azioni** (**strategie**) che i giocatori possono scegliere. Nel nostro caso la strategia consiste di una sola azione

Ogni giocatore deve adottare e poi eseguire una strategia pura, che specifica una particolare azione da intraprendere in ogni situazione (una singola azione nel nostro caso). Il risultato di un gioco è determinato dallo **strategy profile**, che specifica quale strategia è adottata da ogni giocatore.
Normalmente anziché scrivere tutti gli strategy profile, viene realizzata una matrice di payoff che dà l'utilità a ogni giocatore per ogni combinazione di azioni di tutti i giocatori.
Una soluzione a un gioco è uno strategy profile in cui ogni giocatore adotta una strategia razionale.
Diciamo che una strategia s per il giocatore p **domina fortemente** la strategia s' se il risultato di s è meglio per p di quello di s' per ogni scelta di strategie da parte degli altri giocatori.
Una **strategia dominante** è una che domina tutte le altre. E' irrazionale scegliere una strategia fortemente dominata, ed è irrazionale non scegliere una strategia dominante quando ne esiste una.
Dato un giocatore A e due strategie sA1 e sA2 di questo giocatore, diciamo:

- sA1 **domina strettamente** sA2 se, qualunque cosa faccia l'altro giocatore, sA1 dà al giocatore A un payoff maggiore di quello che sA2 può dare
- sA1 **domina debolmente** sA2 se, qualunque cosa faccia l'altro giocatore, sA1 dà al giocatore A un payoff maggiore o uguale a quello che sA2 può dare e, in almeno un caso sA1 dà un payoff maggiore di quello che sA2 dà.
- sA1 è una strategia **fortemente dominante** se sA1 domina fortemente ogni altra strategia del giocatore A. Analogamente per debolmente dominante.

Un risultato è **Pareto optimal** se non c'è un altro risultato preferibile da tutti i giocatori.
La definizione di Pareto optimal si basa sulle preferenze: abbiamo visto che uno strategy profile è una coppia di strategie, una per ciascun giocatore, con un valore numerico (utilità) associato a ciascun giocatore. Dati due risultati w1 e w2, un giocatore A preferisce w1 a w2 se l'utilità per A in w1 è maggiore di quella in w2.
Possiamo definire:

- w1 è strettamente Pareto superiore a w2 se ogni giocatore preferisce w1 a w2.
- w1 è debolmente Pareto superiore a w2 se ogni giocatore considera w1 migliore o uguale a w2 e almeno un giocatore preferisce w1 a w2.

Un risultato è Pareto optimal se non ce n'è nessuno superiore a lui.
La ragione per cui la soluzione Pareto optimal è improbabile, è che non è un punto di equilibrio, di solito si parla di equilibrio di Nash.

Definizione Pareto otpimal: quando non è possibile aumentare l'utilità di un giocatore senza peggiorare quella degli altri. In altre parole è una strategia utile per tutti i giocatori.

Equilibrio di Nash

Uno strategy profile S=(P1:s1,P2:s2) è un equilibrio di Nash se ogni giocatore P_i in S farebbe peggio se deviasse da S, assumendo che tutti gli altri giocatori si attengano a S. Precisamente:

- Assumendo che il primo agente giochi s1, il secondo agente non può fare niente di meglio che giocare s2, and
- Assumendo che il secondo agente giochi s2, il primo agente non può fare niente di meglio che giocare s1

L'equilibrio è una condizione necessaria per qualunque soluzione a un gioco.
Si può dimostrare che un equilibrio con strategie dominanti è un equilibrio di Nash, ma non tutti i giochi hanno strategie dominanti. Ci possono essere equilibri di Nash che non sono strategie dominanti.
Se ad esempio non c'è nessuna strategia dominante, ma ci sono due equilibri di Nash, a questo punto gli agenti hanno un problema: ci sono diverse soluzioni accettabili, ma se ogni agente sceglie una soluzione differente, allora lo strategy profile risultante non è una soluzione accettabile.

Definizione Equilibrio di Nash: la scelta di un giocatore è ottima per Nash se è la migliore possibile data la scelta dell'altro giocatore. In generale possono esistere più equilibri di Nash in un gioco; alcuni giochi non hanno un equilibrio di Nash; gli equilibri di Nash possono essere sub-ottimali ovvero non essere soluzioni ottimali.
Come si può convergere su una soluzione? Una possibilità è che entrambi scelgano la soluzione Pareto-optimal. Sfortunatamente, è facile trovare giochi dove non ci sia questa soluzione. In questo caso, gli agenti devono comunicare, sia stabilendo una convenzione che ordina le soluzioni prima dell'inizio del gioco, sia negoziando per raggiungere durante il gioco un accordo che soddisfi entrambi.

Gioco a somma zero: un gioco in cui la somma dei payoff in ogni cella della matrice dei payoff è zero. Nei giochi a somma zero a due giocatori le vincite devono essere uguali ed opposte. Quindi nessun profilo di strategie può essere un equilibrio e dovremo considerare strategie miste.
Se un giocatore ha k scelte possibili s_1, s_2, ..., s_k, una strategia mista su queste scelte ha la forma:

- esegui s_1 con probabilità p_1
- esegui s_2 con probabilità p_2
- ...

- esegui s_k con probabilità p_k
  con $p_1 + p_2 + ... + p_k = 1$

Nash ha dimostrato il seguente risultato: ogni gioco in cui ogni giocatore ha un insieme finito di strategie possibili ha un equilibrio con strategie miste (ovvero usando strategie miste c'è sempre una soluzione).
Il teorema di Nash dice che, per ogni gioco, esiste un equilibrio con strategie miste, ma come si fa a trovarlo e con quale complessità?
Esistono solo alcuni risultati per particolari classi di giochi.
Il dilemma di disertare (testify) oppure cooperare (refuse) è il problema fondamentale delle interazioni multi-agenti. Il dilemma del prigioniero è onnipresente. Sembra implicare che la cooperazione non si presenti in società di agenti self-interested (egoisti).

PROGETTAZIONE DI MECCANISMI

Indica la progettazione di protocolli per governare interazioni multiagenti in modo che questi protocolli abbiano certe proprietà desiderabili come ad esempio:

- Successo garantito: il protocollo garantisce che prima o poi verrà raggiunto un accordo
- Massimizzare il welfare sociale: un accordo massimizza la somma delle utilità dei partecipanti
- Pareto efficiency: il risultato della negoziazione deve essere Pareto optimal
- Individual rationality: i partecipanti hanno interesse nel seguire le regole del protocollo
- Stability: il protocollo dà un incentivo a comportarsi in un certo modo (es. equilibrio di Nash)
- Simplicity: la strategia del protocollo è "ovvia"

Esempi di progetti di meccanismi includono la messa all'asta di biglietti aerei economici, l'instradamento di pacchetti TCP tra computer, ecc.
Formalmente un meccanismo consiste in un linguaggio per la descrizione delle possibili strategie adottabili dagli agenti.
A prima vista, il problema di progettare un meccanismo potrebbe sembrare banale. Se ogni agente massimizza la propria utilità, questo porterà automaticamente alla massimizzazione dell'utilità globale. Sfortunatamente questa soluzione non funziona, perché le azioni di ogni agente possono influenzare il benessere degli altri in modo tale da ridurre l'utilità globale.
Un esempio è la tragedy of commons, una situazioni in cui tutti i contadini portano le loro bestie a brucare gratis sui prati comunali, con il risultato di distruggerli e giungere così a un'utilità negativa per tutti. Ogni contadino ha agito singolarmente

in modo razionale, ragionando che l'uso del prato comune era libero. Un approccio standard per gestire simili problemi è "far pagare" a ogni agente l'uso delle risorse collettive.

ASTE

Un meccanismo importante è quello delle aste. Assumiamo che ci sia un singolo bene in vendita, che ha un valore pubblico per tutti i partecipanti e un valore privato che può essere diverso da un partecipante ad un altro.
Il tipo di asta più comune è l'**asta inglese**, in cui il banditore incrementa via via il prezzo dell'articolo, controllando che ci siano ancora acquirenti interessati, finché non ne rimane solo uno. L'asta inglese ha anche la caratteristica che i partecipanti hanno una semplice strategia dominante: continuare a puntare finché il costo corrente è inferiore al valore personale.
Un tipo di asta meno nota è l'**asta olandese** in cui il banditore parte offrendo il bene a un prezzo molto alto e continua abbassando il prezzo finché un qualche partecipante accetta l'offerta del banditore.
Una proprietà indesiderabile di queste aste è il loro alto costo di comunicazione: i partecipanti devono essere tutti nella stessa stanza o devono disporre di linee sicure ad alta velocità.
Un meccanismo alternativo è quello dell'**asta in busta chiusa**, in cui ogni partecipante comunica al venditore segretamente una singola offerta, e quella più alta vince. Con questo meccanismo, la strategia di offrire il valore reale non è più dominante.
Una variante di questo meccanismo è l'**asta Vickrey**, detta anche asta in busta chiusa al secondo prezzo, in cui, come nel caso precedente vince chi fa l'offerta più alta, ma paga un prezzo corrispondente alla seconda offerta più alta, e non alla sua propria. In questo caso si può verificare che esiste una strategia dominante molto semplice: offrire il proprio valore privato.
In generale, se il tuo valore privato è v_i e hai offerto b_i, è semplice verificare che non si ottiene un risultato migliore se b_i è diverso da v_i :

- $b_i > v_i$: puoi risultare primo, ma rischi di pagare una cifra maggiore di v_i
- $b_i < v_i$: hai meno probabilità di vincere rispetto a offrire v_i. Ma, anche se vinci, la cifra da pagare sarà sempre quella che pagheresti offrendo v_i, ossia quella proposta dal secondo vincitore.

AGENTSPEAK(L) E JASON

Linguaggio di programmazione per agenti BDI introdotto da Rao nel '96, si propone di colmare il gap tra specifica teorica ed implementazione di un agente BDI.

Jason è la prima significativa implementazione di AgentSpeak(L), realizzata in Java.
Gli agenti BDI vengono da sempre trattati da due punti di vista:

- Specifica teorica
- Implementazione

Le implementazioni esistenti utilizzano strutture dati per rappresentare belief, desires e intentions, invece che gli operatori modali. Il gap tra teoria e pratica resta eccessivo. Causa principale: complessità del theorem proving e del model checking delle logiche usate per formalizzare i BDI agents.
Rao quindi introduce una formalizzazione alternativa degli agenti BDI: AgentSpeak(L), come architettura per agenti e linguaggio di programmazione.
Visto anche come estensione della programmazione logica per supportare l'architettura BDI.
Agenti BDI:

- Sistemi collocati in un ambiente dinamico, che muta nel tempo
- Sistemi in grado di percepire informazioni provenienti dall'ambiente
- Sistemi in grado di compiere delle azioni (ed apportare modifiche all'ambiente) sulla base delle proprie attitudini mentali: beliefs, desires e intentions sono le principali attitudini

Belief: cattura la componente di informazione
Desire: cattura la componente motivazionale
Intention: cattura la componente decisionale
Per formalizzare queste nozioni sono state impiegate logiche multi-modali, temporali, dinamiche e action logics. Tuttavia, la complessità del theorem proving e del model-checking di tali logiche non è ancora chiara.

AgentSpeak(L)

Si sviluppa da delle precedenti proposte: PRS (Procedural Reasoning System) e la sua evoluzione dMARS (Distributed Multi-Agent Reasoning System).
Parte da queste basi, ne assume le caratteristiche principali, ne semplifica gli aspetti secondari e quindi AgentSpeak(L) ne emerge come una proposta concreta che mette insieme il buono delle precedenti proposte senza eccedere nelle complicazioni.
Linguaggio di programmazione basato su un linguaggio del prim'ordine semplificato, con eventi e azioni.
Il comportamento di un agente (ossia le sue interazioni con l'ambiente) è dettato dai programmi scritti in AgentSpeak(L). Beliefs, desires ed intentions non sono rappresentati esplicitamente con formule modali, bensì il progettista attribuisce tali nozioni all'agente usando il linguaggio AgentSpeak(L).

Current belief state dell'agente: modello di se stesso, dell'ambiente e degli altri agenti.

Desires: stati che l'agente vuole raggiungere (sulla base di stimoli interni o esterni); per la precisione, AgentSpeak(L) fa riferimento ai goals, che si possono intendere come desires adottati/scelti.

Intentions: adozione di programmi per soddisfare certi stimoli.

AgentSpeak(L) è un linguaggio testuale per scrivere programmi per agenti.

Nozioni di base beliefs (analoghi ai fatti della programmazione logica), piani, azioni, intentions, eventi, goals.

I piani codificano il modo in cui deve essere raggiunto un certo goal.

I piani sono:

- Context sensitive
- Possono essere invocati dall'utente
- Consentono una decomposizione gerarchica dei goal
- Sintatticamente simili alle clausole della programmazione logica (anche se con un diverso comportamento)

Alfabeto del linguaggio formale

- Variabili
- Costanti
- Simboli funzionali
- Simboli predicativi
- Simboli di azione
- Connettivi
- Quantificatori
- Simboli di punteggiatura

Connettivi

Connettivi della logica del primo ordine:

- & congiunzione "and"
- not negazione
- <- implicazione

! achievement goal

? test goal

; sequenza

Dalla logica del prim'ordine AgentSpeak(L) prende le definizioni di termini e formule.

Le variabili del linguaggio sono caratterizzate dall'iniziale maiuscola.

Belief atom

Sia b un simbolo predicativo e siano $t_1, \ldots, t_n$ termini.

Allora: $b(t_1,\ldots,t_n)$ è un belief atom e si scrive anche b(**t**).

Un belief atom e la sua negazione sono detti belief literal.
Un belief atom ground (cioè senza variabili con occorrenze libere, con tutte variabili ground) è detto base belief.
Siano b(**t**) e c(**s**) belief atoms; allora:

b(**t**) & c(**s**)

not b(**t**)

sono beliefs.
Istanze ground di beliefs si dicono base beliefs.

TRIGGERING EVENT

Quando un agente acquisisce un nuovo goal oppure nota una modifica nell'ambiente, esso può far scattare aggiunte o cancellazioni di goals o beliefs.
Possibili triggering events:

- Aggiunta/Rimozione di un belief
- Aggiunta/Rimozione di un goal

L'aggiunta di un belief/goal è rappresentata dall'operatore +
La rimozione di un belief/goal è rappresentata dall'operatore –

AZIONE

Scopo di un agente: osservare l'ambiente e, sulla base di tali osservazioni e dei propri goal, eseguire determinate azioni. Le azioni compiute dall'agente possono modificare lo stato dell'ambiente.
Sia a un simbolo di azione e siano $t_1, \ldots, t_n$ termini. Allora: $a(t_1,\ldots,t_n)$ è un'azione.

PIANO

Un piano specifica il modo in cui un agente potrebbe raggiungere un determinato obiettivo:
⟨piano⟩ := ⟨head⟩ <– ⟨body⟩
⟨head⟩ := ⟨triggering event⟩ : ⟨context⟩
triggering event specifica perché il piano è stato attivato, ossia l'aggiunta/rimozione di un belief o di un goal che tale piano si propone di gestire.
context specifica quali beliefs dovrebbero essere soddisfatti nel set delle credenze dell'agente quando il piano è attivato (scatta).
body è una sequenza di goal o azioni e specifica:

- I goals che l'agente vuole raggiungere (achievement goals) o testare (test goals)
- Le azioni che l'agente deve eseguire

true rappresenta un componente vuoto (context o body). Un body vuoto viene riscritto per convenzione con true.

Siano:

- e un triggering event
- $b_1,...,b_m$ belief literals
- $h_1,...,h_n$ goal o azioni

allora e : $b_1\&...\&b_m$ <– $h_1,...,h_n$ è un piano.

PROGETTO DI UN AGENTE

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

- Un insieme di beliefs B
- Un insieme di piani P
- Un insieme di intentions I
- Un insieme di eventi E
- Un insieme di azioni A
- Un insieme di funzioni di selezione: S_ε, S_O, S_I

Il progettista specifica un agente scrivendo un insieme di base beliefs e un insieme di piani. Il progetto di un agente è, pertanto, del tutto simile alla scrittura di un programma logico, con la definizione di un insieme di fatti e un insieme di regole (ma con differenze).

AGENTSPEAK(L) REASONING CYCLE

Viene generato un triggering event quando un agente nota una modifica nell'ambiente circostante oppure quando un utente esterno chiede all'agente di raggiungere un goal.

Gli eventi possono essere:

- Esterni (modifica dell'ambiente)
- Interni (modifica dello stato dell'agente)

Tutti gli eventi vengono aggiunti al set E.

Belief revision function: un evento, come un'aggiunta di un nuovo belief, causa anche una revisione dei belief interni all'agente.

S_ε seleziona in E un evento E_0 da processare. E_0 viene rimosso da E e viene usato per unificare con i triggering events dei piani del set P. Per unificazione si intende l'algoritmo che permette di definire un'instanziazione delle variabili libere di due termini affinché diventino equivalenti.

Quando uno dei due termini è già ground la sostituzione si chiama **matching**.

Quando invece la sostituzione avviene in tutti e due i termini allora si chiama **unificazione**.

I piani i cui triggering events unificano con E_0 sono detti **piani rilevanti**; l'unificatore è detto **unificatore rilevante**. L'unificatore rilevante è applicato al contesto dei piani rilevanti.
Una correct answer substitution è ottenuta dal contesto.
Per alcuni piani le condizioni dei contesti risultano essere conseguenze logiche del set dei base beliefs B: questi piani sono detti **piani applicabili** mediante un **unificatore applicabile**. L'unificatore applicabile risulta dalla composizione della correct answer substitution con l'unificatore rilevante.
Dato un evento E_0, diversi piani risultano applicabili. La funzione di selezione S_O sceglie uno di questi piani che chiamiamo P_o. Applicando l'unificatore applicabile a P_O si ottiene l'intended means in risposta al triggering event.
Ogni intention è uno stack di piani parzialmente istanziati o intention frames.
Evento esterno: l'intended means è usato per creare una nuova intention, che viene aggiunta al set I.
Evento interno: l'intended means è inserito in cima all'intention esistente che ha "fatto scattare" (triggered) l'evento interno (per raggiungere un goal).
A questo punto la funzione di selezione S_I sceglie una intention da eseguire.
Quando l'agente esegue una intention, esegue il primo goal o azione del body del top dell'intention:

- Eseguire un achievement goal equivale a generare un evento interno per aggiungere il goal alla corrente intention
- Eseguire un test goal equivale a trovare una sostituzione per il goal che lo renda una conseguenza logica dei base beliefs; se viene trovata una sostituzione, il test goal è rimosso dal corpo del top dell'intention
- Eseguire un'azione equivale ad aggiungerla al set di azioni A e rimuoverla dal corpo del top dell'intention

A questo punto, l'agente torna a valutare il set degli eventi E, il ciclo ricomincia, fino a quando:

- Il set degli eventi E è vuoto
- Non è possibile eseguire altre intentions

INTENTION

I è l'insieme delle intentions, ogni intention è uno stack di piani parzialmente istanziati, ossia piani dove alcune variabili sono state istanziate. Un'intention è denotata con uno stack: [p1 |p2 | ... |pz]
Una particolare intention è la true intention: [+!true: true <– true] che denoteremo con **T**.

EVENTO

L'insieme E è l'insieme degli eventi. Ogni evento è una coppia: ⟨e,i⟩. Dove:

- *e* è un triggering event
- *i* è una intention

Se *i* è la true intention **T**, l'evento è un evento esterno, altrimenti si dice evento interno.

Scelto un evento ⟨d,i⟩ dal set E, il triggering event d viene unificato con i triggering event di tutti i piani contenuti nel set P. Il **most general unifier** (**mgu**) che unifica i due eventi è detto relevant unifier.

L'intention i può essere: la true intention T o un'intention esistente che ha generato l'evento.

PIANO APPLICABILE

Un piano rilevante è anche applicabile se esiste una sostituzione che, composta con l'unificatore rilevante ed applicata al contesto, fa sì che quest'ultimo risulti una conseguenza logica dei base beliefs B. In altre parole, la condizione del contesto di un piano rilevante deve essere conseguenza logica di B affinché il piano sia applicabile.

INTENDED MEANS

Come detto, la funzione S_ε seleziona un evento ⟨d,i⟩. La natura di i determina il tipo di evento. A seconda del tipo di evento (interno o esterno), l'agente esegue un opportuno intended means.

JASON

È un interprete AgentSpeak (la prima implementazione significativa di AgentSpeak(L)) basato su Java e usato con la piattaforma SACI per la distribuzione di un sistema multi-agent sulla rete.

Interpreta il linguaggio AgentSpeak(L) originale ma aggiunge alcune importanti caratteristiche:

- Gestione del fallimento dei piani
- Supporto per lo sviluppo di ambienti da programmare in Java
- Possibilità di eseguire un ambiente multi-agente utilizzando SACI
- Possibilità di personalizzare (in Java) funzioni di selezione, di belief-revision, di azione, di comunicazione fra agenti
- Libreria di "azioni interne" di base
- Possibilità di aggiungere "azioni interne" definite in Java dall'utente
- Aggiunta della negazione forte

Costruire un agente AgentSpeak(L) con Jason è molto semplice: si deve inserire il nome dell'agente nel file di configurazione e creare un file .asl contenente i piani che descrivono il comportamento dell'agente.

Differenze con AgentSpeak(L)

Possibile uso della negazione forte:

- La negazione debole è usata nel contesto dei piani come in AgentSpeak(L), introdotta dal not: not location(car, Y) (anche negazione per fallimento, ovvero non riesco a dimostrare location(car, Y))
- La negazione forte è usata per negare un predicato/fatto: ~location(waste, b) (questo vuol dire proprio che la spazzatura non è in b)

I termini possono essere:

- Atomi
- Strutture
- Variabili
- Liste (tipo Prolog)
- Numeri (interi o floating point)
- Stringhe (tra "")

Introdotta la possibilità di annotare i predicati atomici della belief base, utile per conservare l'origine di tale informazione. **Annotazione**: lista di termini tra parentesi quadre associata ad un predicato, es.: $ps(t_1, ..., t_n)[a_1, ..., a_n]$.

Due annotazioni particolari:

- [source(percept)]: l'informazione è stata percepita dall'ambiente
- [source(self)]: l'informazione è stata aggiunta dall'agente stesso durante l'esecuzione di un piano

Tutti i predicati nella belief base hanno una annotazione speciale che riporta la sorgente dell'informazione:

```
1. blue(box1)[source(ag1)]
2. red(box1)[source(percept)]
3. colourblind(ag1)[source(self), doc(0.7)]
4. lier(ag1)[source(self), doc(0.2)]
```

È possibile associare una label a ciascun piano. La label può essere un qualsiasi predicato (consigliata arietà zero), quindi anche un predicato con annotazione. Utili per personalizzare le funzioni di selezione.

Sono previsti eventi per la gestione del fallimento dei piani. Tale evento è generato se un'azione fallisce o non vi sono piani applicabili per un evento con aggiunta di un goal +!g. In tali situazioni viene generato un evento interno per -!g associato alla stessa intention. Se il programmatore ha previsto un piano per -!g e questo risulta

applicabile, verrà eseguito. Altrimenti, viene eliminata l'intera intention e segnalato un warning.
Azioni interne: possono essere usate sia nel contesto che nel body di un piano. Introdotte dal punto: .send(...). Sono distinte dalle azioni che l'agente compie sull'ambiente mediante gli attuatori. Azioni interne standard sono quelle contenute nella directory src/stdlib e quelle definite dall'utente in Java.
Esempi di azioni interne per la gestione degli aspetti BDI:

- .desire(literal)
- .intend(literal)
- .drop_desires(literal)
- .drop_intentions(literal)

Esempi di azioni interne standard:

- .send(receiver, illocutionary force, propositional content): usata nella comunicazione tra agenti. Receiver è il nome del destinatario del messaggio; illocutionary force descrive il tipo di messaggio (es. tell, achieve); propositional content è un predicato che rappresenta l'informazione trasmessa
- .print(...): scrive messaggi sulla console su cui l'agente o SACI è in esecuzione. Ha un numero qualsiasi di parametri, che possono essere stringhe così come termini AgentSpeak(L)

SISTEMI MULTI-AGENTE CON JASON

Un sistema multi-agente prevede:

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

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