Università degli Studi di Genova

Facoltà di Ingegneria

TESI DI LAUREA

Un sistema multi-agente per lo scheduling on-line in ambito manifatturiero

Relatori: Chiar.mo Prof. Antonio Boccalatte
Chiar.mo Prof. Massimo Paolucci

Correlatore: Dott. Ing. Andrea Gozzi

Candidato: Paolo Copello

LIDO – Laboratorio Informatica DIST Siemens-Orsi

Capitolo 1: Introduzione

1.1 I sistemi multi-agente come risposta alle nuove esigenze dei sistemi manifatturieri

In tempi recenti si è assistito a cambiamenti radicali nel campo dei sistemi manifatturieri: dalla produzione di massa si è passati ad una produzione sempre più personalizzata per venire incontro ai frequenti mutamenti della domanda. Le dimensioni degli ordini vanno via via riducendosi, come si sono ridotti notevolmente i cicli di vita dei prodotti.

Le imprese vengono condotte con mentalità sempre meno chiusa e più collaborativa, con l'esigenza di avere un'efficiente cooperazione tra le parti. Diventa inoltre sempre più difficile, se non quasi impossibile, gestire un sistema manifatturiero (composto di numerose parti, spesso eterogenee, ciascuna caratterizzata da diverse informazioni, obiettivi, esperienze, autorità decisionali, ecc.) in modo centralizzato [Gou98].

Appare chiaro quindi che i sistemi manifatturieri stanno subendo un significativo "cambio di paradigma". La tecnologia da sola non è sufficiente per supportare questo cambiamento, ma deve essere integrata con nuove strutture organizzative in grado di sfruttarne pienamente i benefici e di portare ad un aumento delle prestazioni. La struttura tradizionale, caratterizzata da una gerarchia fissa e verticale, deve diventare più agile, con un minor numero di livelli e con una maggiore distribuzione delle decisioni.

Alcuni dei requisiti fondamentali dei nuovi sistemi manifatturieri sono dunque una localizzazione dell'informazione e dei processi decisionali, una maggiore cooperazione, ed una migliore integrazione tra le componenti concrete (entità fisiche) ed astratte (informazioni).

Per questi e altri problemi si cercano soluzioni nell'introduzione di nuove tecnologie flessibili e scalabili, tra le quali spiccano quelle basate sugli *agenti*.

L'interesse per la ricerca nel campo degli agenti sta aumentando continuamente. I concetti di "agente autonomo" e "sistema multi-agente" (*MAS, Multi-Agent System*), introdotti per la prima volta nel campo dell'intelligenza artificiale distribuita (*DAI, Distributed Artificial Intelligence*), possono essere applicati a numerosi contesti per distribuire i controlli e i processi decisionali tra le diverse componenti di un sistema.

Una definizione abbastanza generica di agente (tra le numerose altre esistenti) è quella di Wooldridge e Jennings: un agente è "*un sistema informatico situato in un certo ambiente, capace di azioni autonome in tale ambiente allo scopo di raggiungere i propri obiettivi*". [Wool95]

Gli agenti sono entità software caratterizzate da autonomia, reattività (capacità di percepire i cambiamenti dell'ambiente e di comportarsi di conseguenza), proattività (capacità di prendere iniziative per soddisfare degli obiettivi) e abilità sociale (capacità di interagire con altri agenti o con esseri umani). I MAS o sistemi multi-agente sono comunità sociali di agenti che, pur conservando la propria autonomia e individualità, sono legati da rapporti di interdipendenza.

L'interesse per i MAS da parte dei ricercatori è testimoniato dalla grande quantità di studi sull'applicazione delle tecnologie ad agenti nei più disparati campi scientifici e industriali (si veda [Oliveira], [Jenn98]).

In particolare, nell'ambito dei sistemi manifatturieri, i MAS sono considerati uno strumento alternativo di supporto ad uno dei problemi decisionali più importanti e difficili al tempo stesso: lo *scheduling*.

1.2 Scheduling e sistemi ad agenti

La *schedulazione* o *scheduling* è in generale un problema decisionale di allocazione ottima di risorse nel tempo: tali risorse (limitate) devono essere assegnate nel tempo ad un insieme di attività, rispettando determinati vincoli, in modo da minimizzare una certa funzione di costo (definita in base al problema specifico). L'output di questo processo, ossia la soluzione del problema decisionale, è costituito da uno *schedule*.

Per quanto riguarda lo scheduling in campo manifatturiero, le risorse da assegnare sono in generale rappresentate da macchine, e le attività produttive sono i lavori (*job*) da svolgere.

Lo scheduling è di primaria importanza in questo ambito, in quanto è direttamente collegato alle prestazioni del sistema (consegna dei prodotti, livelli di scorte) nonché all'utilizzo efficiente delle macchine. Tuttavia, ottenere uno schedule ottimo è spesso impossibile: nella maggior parte dei casi, infatti, i problemi di scheduling risultano essere problemi combinatori computazionalmente intrattabili (*NP-hard*) per cui una soluzione ottima può essere trovata solo per istanze di dimensioni estremamente ridotte.

Per affrontare problemi reali quindi è in genere necessario sviluppare tecniche per ottenere uno scheduling sub-ottimo in tempi ragionevoli.

Sempre più spesso le decisioni di scheduling devono essere prese in tempo reale per reagire a variazioni dinamiche, dal momento che le informazioni sui job diventano disponibili solo dopo il loro arrivo. In più, molte industrie hanno

recentemente adottato la strategia di produzione *Just-In-Time* (JIT), la quale, pur riducendo notevolmente i costi operativi, perde molti dei suoi vantaggi in caso di cambiamenti improvvisi nelle attività della catena logistica. Per questo motivo la pianificazione e le decisioni operative tendono ad essere sempre più distribuite tra le varie entità, in modo da fornire a livello locale risposte più dirette e flessibili ai problemi decisionali.

L'applicazione dei sistemi ad agenti all'ambiente manifatturiero è un argomento ampiamente studiato (alcuni esempi sono riportati in [Shen99]). Per soddisfare la sempre crescente necessità di una produzione industriale che sia più flessibile e adattabile ai cambiamenti del mercato sono stati introdotti molti nuovi paradigmi, quali *Intelligent Manufacturing*, *Holonic Manufacturing Systems* (HMS) e *Agile Manufacturing*.

I sistemi informativi integrati, i sistemi ERP (*Enterprise Resource Planning*), le reti di calcolatori e, in generale, le nuove tecnologie della comunicazione rappresentano il miglior ambiente in cui sperimentare nuove metodologie e modelli.

1.3 Il problema affrontato e l'approccio sviluppato

Lo scopo della tesi è definire un approccio basato su un'architettura MAS per affrontare problemi di scheduling dinamici che emergono nel contesto dei sistemi manifatturieri flessibili (*FMS, Flexible Manufacturing Systems*), solitamente considerati i problemi più difficili nel campo della gestione della produzione. In particolare, è stata presa in esame una specifica classe di problemi: lo scheduling di job, caratterizzati singolarmente da tempo di rilascio (*ready time*) e data di consegna (*due date*), su un insieme di macchine identiche parallele, con l'obiettivo di minimizzare una funzione di costo che includa sia il ritardo rispetto alla due date (*tardiness*) sia l'anticipo (*earliness*). Si tratta di un obiettivo non-regolare (ossia tale che il costo non sempre cresce al crescere del tempo di

completamento di un job) tipico della produzione Just-In-Time. In più, si assume che la lista dei job da schedulare non sia nota a priori: in questo modo occorre affrontare un problema di scheduling dinamico; inoltre, poiché le caratteristiche dei job sono conosciute solo dopo il loro arrivo, il problema si dice anche di *on-line scheduling*. La versione off-line di questo problema (con tutti i job noti a priori), anche se concettualmente semplice, appartiene alla categoria NP-hard, cioè non è possibile trovarne la soluzione ottima in tempi brevi.

Questa tesi ha due obiettivi principali:

1. sviluppare un'architettura multi-agente che possa supportare decisioni di scheduling on-line in sistemi manifatturieri: da questo punto in avanti tale architettura sarà chiamata MASS, Multi-Agent Scheduling System. Ciò implica identificare i tipi e i ruoli degli agenti coinvolti nel sistema e i loro protocolli di interazione in modo da definire un sistema che possa essere implementato anche in un ambiente di produzione distribuito; in un simile sistema, gli agenti sono associati con entità sia fisiche che logiche, e sono responsabili delle decisioni legate a tali entità.

2. studiare nuove euristiche evolutive per il supporto alle decisioni, derivate dal comportamento collettivo di entità decisionali in collaborazione o in competizione tra loro. Da questo punto di vista, l'interesse per i MAS è motivato dalla necessità di modellare una società con particolari regole di interazione, il cui comportamento globale corrisponda ad una complessa euristica decisionale.

Il sistema sviluppato, che sarà descritto in dettaglio nel Capitolo 5, è composto da diverse classi di agenti, ma i più importanti sono quelli associati ai job da schedulare (*JA, job agent*) e alle macchine (*MA, machine agent*). L'obiettivo dei JA è di ottenere il miglior servizio possibile da parte dei MA: in caso di produzione JIT, ciò significa che il completamento dei job deve avvenire

esattamente alla data di consegna (*due date*). Dato che questo non è sempre possibile, lo schedule effettivo si ottiene attraverso un protocollo di negoziazione che coinvolge JA e MA. In particolare, i MA possono fare promesse di servizio ai JA, salvo poi annullarle quando si presentino alternative migliori.

Gli altri agenti presenti nel sistema sono i due "generatori" (JGA e MGA, il cui compito consiste nel creare rispettivamente agenti job e agenti macchine) e il *Real-time Coordinator Agent* (RCA), necessario per gestire l'evoluzione del tempo nell'ambiente ad agenti.

1.4 Descrizione del contenuto della tesi

La tesi è suddivisa in due parti principali: nella prima vengono presentate e descritte le problematiche di riferimento e gli strumenti e le metodologie disponibili. In particolare:

- Il Capitolo 2 è una panoramica sui problemi di scheduling e sui possibili approcci, in particolare per quanto riguarda i problemi dinamici e Just-In-Time.

- Nel Capitolo 3 è descritto in modo più approfondito il concetto di agente e di sistema multi-agente; inoltre sono analizzati lo standard FIPA (*Foundation for Intelligent Physical Agents*) e i diagrammi AUML.

- Nel Capitolo 4 vengono studiati e confrontati sei popolari strumenti software per lo sviluppo di sistemi ad agenti, con l'obiettivo di scegliere il più adatto agli scopi di questa tesi.

Nella seconda parte è definito il particolare problema di scheduling affrontato, e viene illustrato l'approccio adottato ed i risultati ottenuti. In particolare:

- Nel Capitolo 5 è formalizzato lo specifico problema di scheduling, ed è presentata l'implementazione del sistema di scheduling ad agenti.
- Nel Capitolo 6 è descritta la procedura per l'analisi delle prestazioni del sistema di scheduling e sono esposti i risultati delle prove sperimentali.
- Infine, alcune conclusioni e i possibili sviluppi futuri sono riportati nel Capitolo 7.

Capitolo 2: Problemi di scheduling – concetti di base e terminologia

2.1 Introduzione

2.1.1 Definizione di scheduling

Come introdotto nel Capitolo 1, lo *scheduling* può essere definito come segue:

"l'allocazione, soggetta a vincoli, di alcune *risorse* ad un insieme di *attività* nello spazio-tempo, in modo da minimizzare un dato costo". [Petrovic02]

In [Agnetis00] è sottolineata l'importanza del tempo nei problemi di scheduling, definiti come

"problemi decisionali in cui riveste importanza il fattore *tempo*, visto come *risorsa (scarsa)* da allocare in modo ottimo a determinate *attività*"

Dal momento che i problemi di scheduling sono nati e si sono sviluppati in un contesto manifatturiero, si usano quasi sempre i termini "*job*" e "*macchine*" per indicare rispettivamente le attività e le risorse, anche quando le entità reali trattate

hanno poco a che vedere con la produzione industriale. Le macchine sono chiamate anche "*processori*" (soprattutto nei problemi applicati all'informatica e alla schedulazione nei sistemi operativi).

In alcuni problemi è necessario distinguere tra job e task: ogni *job* (lotto, ordine) è composto da più *task* (singole attività), ovvero un job è un insieme di task tecnologicamente legati tra loro. In tal caso, le sequenze di task che compongono ogni job definiscono dei cosiddetti *vincoli tecnologici* (in un problema di scheduling possono esserci anche altre tipologie di vincoli, discusse nel seguito); ogni passaggio di un task (o job) attraverso una macchina è chiamato *operazione*.

Strettamente legati allo scheduling sono i concetti di sequenziazione e timetabling, definiti come segue in [Petrovic02]:

"La *sequenziazione* (*sequencing*) è la costruzione, soggetta a vincoli, di un ordine secondo il quale devono essere svolte delle attività."

"Il *timetabling* è l'allocazione, soggetta a vincoli, di alcune risorse ad un insieme di attività nello spazio-tempo, in modo che un dato insieme di obiettivi sia il più possibile soddisfatto".

Si noti che la definizione di timetabling è identica a quella di scheduling, solo che si parla più genericamente di "insieme di obiettivi" (il che potrebbe comprendere anche una funzione di costo).

2.1.2 Descrizione di un generico problema di scheduling

Con il termine "problema" si intende solitamente una generica classe di problemi, mentre un caso particolare (un problema concreto) è chiamato *istanza* di problema. Nel seguito saranno indicati con **m** e **n** rispettivamente il numero di macchine e di job; inoltre con l'indice **j** verrà indicato un generico job, mentre con l'indice **i** una generica macchina.

Varie informazioni possono essere associate ad un job **j**:

- *Tempo di processamento* o *durata* $\mathbf{p_{ij}}$ (*processing time*[1]). È il tempo che il job j richiede alla macchina i per essere eseguito: in genere esso corrisponde ad uno dei task (l'i-esimo) che compongono il job j. Nel caso in cui ogni job sia composto da un singolo task, la durata è indicata semplicemente come $\mathbf{p_j}$. Comprende solitamente anche il tempo di trasporto del job presso la macchina[2].

- *Tempo di consegna* $\mathbf{d_j}$ (*due date*). Indica l'istante di tempo (rispetto a un tempo iniziale 0) entro il quale l'esecuzione del job j dovrebbe essere terminata. In genere, la violazione di una *due date* comporta dei costi (penale, perdita di fiducia da parte del cliente, ecc.).

- *Peso* $\mathbf{w_j}$. Non sempre è presente: rappresenta l'importanza relativa del job j rispetto agli altri. Il peso può rappresentare il costo di tenere nel sistema il job (ad es., un costo di immagazzinamento) oppure indicare una priorità dovuta ad un cliente più importante.

- Un'altra informazione che può essere associata ai job è il tempo di rilascio, che però viene spesso classificato tra i vincoli (vedi §2.2.2).

L'istante di inizio elaborazione di un job (*starting time*), ovvero la variabile che deve essere determinata per risolvere un problema di scheduling, è generalmente indicato come $\mathbf{s_j}$, anche se alcuni testi usano la notazione S(j), dove S è uno schedule.

Dallo starting time dipendono i valori delle altre variabili associate ai job:

---

[1] Si noti che il termine "tempo" (time) viene usato con due significati diversi a seconda del contesto: può indicare un istante di tempo oppure un intervallo di tempo.

[2] Per "trasporto del job" si intende l'insieme di materie prime o semilavorati che devono raggiungere fisicamente la macchina.

- *Tempo di completamento* $\mathbf{C_j}$. È l'istante in cui l'ultimo task del job j (e quindi l'intero job j) termina. Se non sono ammesse interruzioni, C_j è dato dalla somma dell'istante di inizio dell'ultimo task di quel job e del tempo di processamento dell'ultimo task.

- *Lateness* $\mathbf{L_j}$. È differenza tra il tempo di completamento e la data di consegna del job j. Si noti che se è positiva, la lateness indica un ritardo, se negativa un anticipo rispetto alla *due date*. Si ha dunque: $L_j = C_j – d_j$.

- *Tardiness* $\mathbf{T_j}$. Ritardo: coincide con la lateness quando questa è positiva, ed è zero altrimenti, ossia $Tj = \max [0, L_j]$.

- *Earliness* $\mathbf{E_j}$. Anticipo (opposto della tardiness): coincide con la lateness quando questa è negativa, ed è zero altrimenti, ossia $E_j = \max [0, - L_j]$.

2.1.3 Soluzione di un problema di scheduling: lo schedule

Una soluzione di un problema di scheduling prende il nome di *schedule*. In termini generali, uno schedule è una descrizione completa dell'utilizzo temporale delle macchine da parte dei job che devono essere eseguiti. Nel caso in cui i task che compongono i job non possano essere interrotti, uno schedule è completamente specificato da un assegnamento di istanti di inizio a tutti i task che devono essere eseguiti, altrimenti occorre specificare l'istante di inizio di ciascuna delle parti in cui viene suddivisa l'esecuzione di un task. Spesso si distingue il concetto di *sequenza* da quello di schedule. La sequenza specifica solo l'ordine in cui i job devono essere eseguiti da ciascuna macchina, lo schedule ne specifica anche gli istanti d'inizio. Il timetabling può anche essere definito come il processo che da una sequenza ricava uno schedule (molto spesso, però, data la sequenza è immediato risalire allo schedule).

Uno schedule si dice:

- *ammissibile* (*feasible*): rispetta tutti i vincoli del problema. Ad esempio: una stessa macchina non può eseguire due job contemporaneamente, uno stesso job non può essere eseguito da due macchine contemporaneamente, un job non può essere interrotto (tranne che nei problemi *preemptive*), eventuali precedenze devono essere rispettate.

- *semi-attivo*: la sequenza di elaborazione dei job è tale che, se una qualsiasi operazione fosse anticipata, andrebbe ad alterare la sequenza delle operazioni o a violare i vincoli.

- *attivo*: la sequenza di elaborazione dei job è tale che, se una qualsiasi operazione fosse anticipata, andrebbe a ritardare una qualche altra operazione o a violare i vincoli. Uno schedule attivo è anche semi-attivo.

- *non-delay*: nessuna macchina è lasciata inattiva quando potrebbe iniziare una qualche operazione. Uno schedule non-delay è anche attivo e semi-attivo.

2.1.4 Diagrammi di Gantt

Quando si studiano problemi di scheduling è spesso utile visualizzare la sequenza dei job assegnati alle macchine con un diagramma di Gantt: in tale diagramma vengono tracciati tanti assi orizzontali quante sono le macchine. Gli assi rappresentano il tempo (che scorre da destra a sinistra) e i rettangoli disegnati lungo gli assi indicano quali job vengono processati da una macchina, quando e per quanto tempo. Si veda ad esempio la figura seguente:

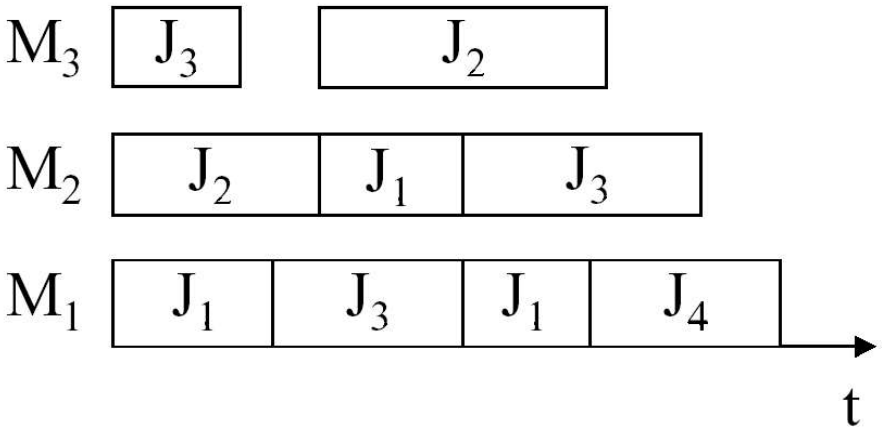


Figura 2-2.1 Esempio di diagramma di Gantt con tre macchine

Quello in figura 2-1 è un diagramma di Gantt orientato alle macchine; è anche possibile tracciare un diagramma orientato ai job (ovvero in cui vengono tracciati tanti assi quanti sono i job, e in cui nei rettangoli sono indicate le macchine che processano ogni job in un dato intervallo di tempo), anche se è molto meno frequente.

2.2 Classificazione e caratteristiche dei problemi di scheduling

Per classificare i problemi di scheduling è spesso usata una *notazione a tre valori*: qui di seguito viene descritta quella utilizzata in [Petrovic02], che è una semplificazione della *notazione di Graham* [Graham79]:

a | b | c

dove a, b e c sono simboli indicanti:

- a = caratteristiche del sistema (numero di macchine, ecc.)
- b = caratteristiche dei job (vincoli)
- c = criterio di ottimalità (funzione obiettivo)

Nel seguito viene indicato il significato dei simboli più utilizzati.

2.2.1 Caratteristiche del sistema (a)

- Singola macchina – "1"
- Macchine parallele identiche – "P" – Ogni job necessita di essere elaborato una sola volta da una qualsiasi delle macchine
- Macchine parallele con diverse velocità – "Q"
- Macchine parallele con velocità dipendente dai job – "R"
- Macchine in serie (*Flow-shop*) – "F" – Tutti i job devono attraversare tutte le macchine in un certo ordine (uguale per tutti).
- *Open-shop* – "O" – Tutti i job devono attraversare tutte le macchine, lo schedulatore decide l'ordine per ogni job.
- *Job-shop* – "J" – Tutti i job devono attraversare tutte le macchine, e ogni job deve farlo in un ordine diverso dagli altri (vincoli tecnologici). Questo problema è una generalizzazione del flow-shop.

Se si desidera si può aggiungere ad ognuno di questi valori il numero delle macchine (m) presenti nel sistema, ad esempio F_4 o F4 invece che F per indicare un problema flow-shop con quattro macchine.

2.2.2 Caratteristiche dei job e vincoli (b)

- *Tempo di rilascio* (*release time* o *ready time*) – "r_j" – Un job non può essere elaborato prima di r_j.

- *Tempi di set-up dipendenti dalla sequenza* – "s_{jk}" – Se si vuole fare seguire il job k al job j, allora tra il completamento del job j e l'inizio di k è necessario riconfigurare la macchina, e questo richiede un tempo s_{jk} (solitamente si assume che il tempo di set-up tra j e k sia comunque indipendente dai job precedenti j e seguenti k). Spesso i tempi di set-up sono trascurati, o supposti indipendenti dalla sequenza: in questo caso sono inglobati nel p_j.

- *Preemption* – "prmp" – I job possono essere interrotti per consentire di elaborare job più urgenti. In questo caso il problema si dice *preemptive* (si tratta di un caso abbastanza raro in un contesto manifatturiero, ma frequente se si considera la schedulazione di processi in un computer).

- *Vincoli di precedenza* – "prec" – Per eseguire un job è necessario che prima siano stati eseguiti certi altri job (sono diversi dai vincoli tecnologici, che riguardano ciascuno un singolo job).

- *Possibili guasti alle macchine* (*breakdown*) – "brkdwn"

- *Restrizioni sulle assegnazioni dei job alle macchine* – "M_j" – Certi job possono essere eseguiti solo su certe macchine: si parla di "*machine eligibility restrictions*".

- *Permutazioni* – "prmu" – Si ha nei problemi flow-shop: indica che le code di accesso ad ogni macchina operano con una logica FIFO. In tal caso il problema viene detto "*permutation flow-shop*": in poche parole, i job non possono "sorpassarsi".

- *Blocking* – "block" – In un flow-shop, se a un dato istante il buffer della macchina i – 1 è pieno, un job terminato sulla macchina i – 1 non può trovare posto nel buffer della macchina i, ed è quindi costretto a tenere bloccata la macchina i – 1, che non può quindi iniziare un nuovo task fino a che non verrà liberata dal job.

- *No-wait* – "no-wait" – Ai job non è permesso di attendere su una macchina (cosa che potrebbe verificarsi nel caso "block") e occorre garantire che all'istante di completamento di un job su una macchina, la macchina successiva sia già libera per processare il job.

- *Ricircolo* – "recrc" – In un problema job-shop, indica che un job può essere processato da una macchina anche più di una volta.

Evidentemente tale notazione non è sufficiente ad esprimere tutti i possibili vincoli: ad esempio, i vincoli di precedenza sono solitamente rappresentati con grafi orientati aciclici, come quello in figura 2-2 [Petrovic02] (altri tipi di grafi sono descritti in [Agnetis00]).

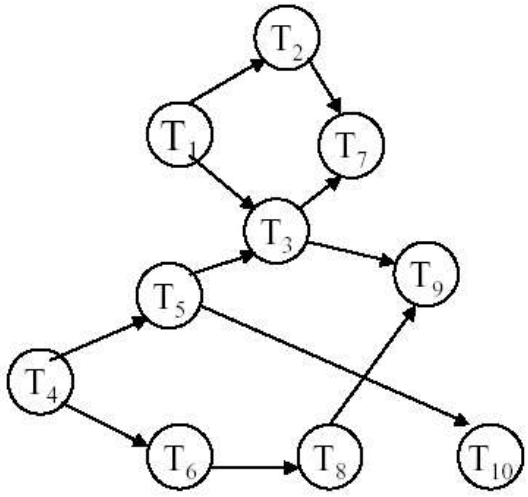


Figura 2-2.2 Esempio di grafo di precedenza

2.2.3 Criterio di ottimalità (c)

Le possibili *funzioni obiettivo* (dette anche *funzioni di costo* o *misure di prestazioni*) possono essere numerose e molto diverse tra loro. Nel seguito sono riportati alcuni esempi:

- *Massimo tempo di completamento* o *makespan* $C_{max} = \max[C_1,\ldots,C_n]$
  È il tempo di completamento del job che termina per ultimo. Esso rappresenta la misura (rispetto al tempo 0) del tempo necessario a completare tutte le attività.

- *Massima lateness* $L_{max} = \max[L_1,\ldots,L_n]$, ossia è il ritardo del job che termina in maggior ritardo rispetto alla propria data di consegna (si noti che potrebbe anche essere negativo, e in tal caso rappresenta l'anticipo – earliness – del job che termina con minore anticipo rispetto alla propria data di consegna).

- *Massima tardiness* $T_{max} = \max[0, L_{max}]$.

- *Somma pesata dei tempi di completamento*, definita come $\sum_{j=1,\ldots n}(w_j \cdot C_j)$.
  Nel caso in cui i pesi siano tutti uguali tra loro, è il tempo complessivamente trascorso nel sistema dai job (dall'istante 0 al loro completamento), ovvero una misura del livello di servizio offerto dal sistema. I pesi introducono inoltre un elemento di priorità.

- *Numero totale dei job in ritardo* (*tardy*): $n_T = \sum_{j=1,\ldots N}(w_j \cdot U_j)$
  U_j è definito come "penalità unitaria" del job j, pari a 1 se $C_j > d_j$, zero altrimenti.

- *Somma pesata di earliness e tardiness*: $\sum_{j=1,\ldots n}(we_j \cdot E_j + wt_j \cdot T_j)$
  É caratterizzata da due diversi pesi, uno per l'earliness e uno per la tardiness, ed è tipica dei problemi Just-In-Time. Questa funzione obiettivo, detta anche "E/T" o "ET", è quella che verrà utilizzata in questa tesi (vedi §2.4 e §5.3).

Una funzione obiettivo si dice *regolare* quando è sempre crescente all'aumentare dei vari C_j. In questo senso, le funzioni indicate nell'elenco

precedente sono tutte regolari tranne l'ultima (che considera anche l'earliness). È possibile definire molte altre funzioni di costo regolari, ad esempio il tempo di completamento medio $\overline{C}$ o la lateness media $\overline{L}$. La maggior parte delle funzioni di costo utilizzate nel problemi reali sono regolari, il che comporta un vantaggio: per certe classi di problemi si può dimostrare [French82] che se la funzione obiettivo è regolare, allora lo schedule ottimo è *attivo* (§2.1.3).

Due funzioni obiettivo si dicono *equivalenti* quando uno schedule ottimo rispetto alla prima è ottimo anche rispetto alla seconda e viceversa: ad esempio $\overline{C}$ e $\overline{L}$ sono equivalenti. In alcuni casi la relazione vale solo in un senso: ad esempio, minimizzare T_{max} minimizza L_{max}, ma il viceversa non è sempre vero (per le dimostrazioni vedi [French82]).

2.2.4 Singolo obiettivo e multi-obiettivo

Stabilire quali devono essere gli obiettivi in un problema non è semplice: sono numerosi, complessi e spesso in conflitto tra di loro. Quelli elencati in precedenza sono solo alcuni tra quelli possibili: altre misure di performance possono essere basate sulla minimizzazione dei tempi di attesa dei job, sulla massimizzazione del *throughput*[3] o sulla riduzione dei periodi di inattività delle macchine.

Spesso, per semplificare la trattazione matematica dei problemi di scheduling, si assume che vi sia un solo obiettivo da minimizzare. In realtà vi possono essere diverse esigenze, talora contrastanti, tra le quali si vuole trovare il miglior compromesso: nel momento in cui si considera più di un obiettivo, è necessario riconsiderare cosa sia da intendersi con soluzione ottima.

Dato uno schedule S, e dette $f_1(S)$ e $f_2(S)$ le due funzioni obiettivo (da minimizzare), S si dice *non-dominato* se non esiste nessun altro schedule S_0 tale che $f_1(S_0) \leq f_1(S)$, $f_2(S_0) \leq f_2(S)$, con almeno una delle due disuguaglianze valida in

[3] La quantità di job elaborati nell'unità di tempo.

senso stretto. In altre parole, se S è non-dominato, è possibile trovare uno schedule migliore per la prima funzione obiettivo solo a patto di peggiorare rispetto alla seconda (e viceversa).

Nei problemi multi-obiettivo è possibile allora adottare diversi approcci.

- Ci si riconduce a un problema con singolo obiettivo, ottimizzando rispetto a $f_1(\ldots)$ ponendo un vincolo sul massimo valore che può assumere $f_2(\ldots)$ (o viceversa).

- Si determinano tutte le soluzioni non-dominate. Peraltro, per determinare l'insieme di tutte le soluzioni non-dominate, può essere necessario risolvere varie istanze del problema con singolo obiettivo.

In questa tesi verranno trattati soltanto problemi a singolo obiettivo; per ulteriori informazioni sui problemi multi-obiettivo si faccia riferimento ad [Agnetis00].

2.2.5 Notazioni alternative per la rappresentazione dei problemi di scheduling

Una notazione più complessa e dettagliata è quella di Graham, trattata approfonditamente in [Graham79] ed adottata da [Blazew96].

In [French82] invece viene utilizza la seguente notazione:

n / m / A / B

dove n e m sono i numeri di job e macchine, A indica il tipo di problema e B la funzione obiettivo. In [French82] sono studiati soltanto problemi singola macchina (m=1 e A omesso), permutation flow-shop (A = P), flow-shop (A = F), e job-shop generico (A = G).

La notazione usata in [Agnetis00] è simile a quella di Graham, con la differenza che indica 1 = macchina singola, P = flow-shop e J = job-shop.

I problemi possono essere inoltre classificati come **statici/dinamici** o **deterministici/stocastici**:

Problemi	**I dati sui job sono:**
statici	noti a priori
dinamici	noti solo dopo un certo istante temporale
deterministici	certi
stocastici	incerti

Tabella 2-2.I

I problemi più semplici da risolvere sono quelli statici e deterministici, ma quasi tutte le situazioni reali presentano caratteristiche dinamiche e stocastiche: gli ordini possono arrivare in qualsiasi momento, ed esistono sempre incertezze sui dati relativi ai job (ad esempio, il processing time non è sempre costante perché potrebbero verificarsi guasti o altri inconvenienti).

Le due classificazioni non sono mutuamente esclusive: ad esempio, il problema affrontato in questa tesi (vedi Capitolo 5) è dinamico e deterministico.

2.3 Approcci ai problemi di scheduling

Negli ultimi trent'anni sono stati sviluppato numerosi algoritmi per risolvere problemi di scheduling: dal momento che è impossibile elencarli tutti e studiarli approfonditamente, in questa sezione sarà presentata una classificazione ad alto livello dei diversi approcci adottati.

2.3.1 Complessità degli algoritmi

Prima di affrontare la classificazione degli algoritmi, è necessario introdurre il concetto di *complessità* (il testo fondamentale su questo argomento è [Garey79]).

L'*efficienza di un algoritmo* nella risoluzione di un determinato problema è solitamente misurata valutando la *funzione di complessità temporale* **f(v)**, definita come il massimo numero di passi computazionali necessari per ottenere una soluzione ottima in funzione delle dimensioni **v** dell'istanza del problema. Poiché si studia il valore massimo, tale valutazione è chiamata anche *analisi del caso peggiore* (*worst case analysis*).

Il valore v, che indica le dimensioni dell'istanza, dipende ovviamente dal tipo di problema considerato, ma solitamente è direttamente proporzionale al numero *n* di job coinvolti. Tale valore viene definito più in dettaglio in [French82] e [Blazew96].

Valutare f(v) può essere molto difficile: per questo motivo, in genere ci si limita a studiarne l'ordine di grandezza. Si dice che **f(v) è O(g(v))** se le due funzioni f(v) e g(v) hanno lo stesso ordine di grandezza, ovvero il loro rapporto tende ad una costante quando v tende all'infinito.

È utile, a questo proposito, dividere i problemi in due classi, **P** (*polinomiali*) ed **NP** (*non-deterministici polinomiali*): tale classificazione è in genere adottata nell'ambito dei problemi decisionali o di decidibilità, ovvero problemi in cui viene posta una domanda a cui bisogna dare risposta affermativa o negativa. In teoria i problemi di scheduling non dovrebbero ricadere in questa categoria – si tratta di problemi di ottimizzazione – tuttavia si può dimostrare che per ogni problema di ottimizzazione è possibile definire un problema decisionale corrispondente.

- Un problema è detto di "classe P" se esiste un algoritmo in grado di trovarne la soluzione ottima in tempi polinomiali, ovvero se **f(v) è $O(v^k)$** con k costante.

- Un problema è detto di "classe NP" se non esiste un algoritmo in grado di trovare la soluzione ottima in tempo polinomiale. Solitamente per i problemi NP si ha che **f(v) è $O(k^v)$**, ovvero la funzione di complessità temporale cresce esponenzialmente al crescere delle dimensioni del problema.

In particolare, un problema decisionale Π (di classe NP) si dice **NP-completo** se qualsiasi altro problema di classe NP è riconducibile a Π in tempo polinomiale. Un problema di ottimizzazione (quindi anche di scheduling) si dice **NP-hard** quando il problema decisionale corrispondente è NP-completo.

Purtroppo, moltissimi problemi di scheduling sono NP-hard: per questo motivo, a differenza di quanto accade per altre aree dell'ottimizzazione combinatoria, a tutt'oggi per i problemi di scheduling più difficili non è possibile indicare un unico approccio nettamente preferibile per la loro soluzione, ma di volta in volta può essere più appropriato utilizzare tecniche diverse. Nel seguito sono elencate le più utilizzate, divise tradizionalmente in tre categorie: tecniche *costruttive*, *enumerative*, *euristiche*.

2.3.2 Algoritmi costruttivi

Un algoritmo costruttivo genera la soluzione ottima a partire dai dati del problema, seguendo un insieme di regole che determinano esattamente la sequenza delle operazioni. Molti problemi a singola macchina (e un numero molto basso di problemi coinvolgenti più macchine) possono essere risolti con tali tecniche.

Esempi di algoritmi costruttivi per problemi a singola macchina sono SPT (Shortest Processing Time) e EDD (Earliest Due Date); mentre per problemi coinvolgenti più macchine è possibile citare l'algoritmo di Johnson o la tecnica grafica di Arker [French82].

Le tecniche costruttive hanno quasi sempre complessità polinomiale (i relativi problemi appartengono cioè alla classe P).

2.3.3 Algoritmi di enumerazione

Queste tecniche si basano su di un principio molto semplice: elencare (o *enumerare*) tutte le possibili soluzioni ammissibili, eliminare quelle non ottime e ottenere quella (o quelle) ottime.

In teoria questo approccio funziona in quanto il numero di possibili soluzioni ad un problema di scheduling è finito (si tratta di un problema combinatorio); tuttavia è proibitivo dal punto di vista computazionale, dato che il numero delle soluzioni cresce esponenzialmente in funzione delle dimensioni del problema. Per questo sono state create le tecniche di *enumerazione implicita*, che enumerano tutte le possibili sequenze di job, senza però farlo esplicitamente, ma sfruttando opportunamente le informazioni specifiche disponibili su un dato problema.

La più nota tecnica di enumerazione implicita è il *branch and bound*, che partiziona l'insieme delle possibili soluzioni in sottoinsiemi sempre più piccoli, costruendo un albero di enumerazione (*branching*), e quindi stimando in ciascun nodo un limite inferiore (*lower bound*) sul valore della migliore soluzione ottenibile da quel nodo in giù.

Altri algoritmi enumerativi sono quelli basati sulla *programmazione dinamica*, una tecnica ideata da Richard Bellman [Bellman57] e basata sulla ricorsione.

Gli approcci enumerativi sono molto utili per risolvere numerosi problemi di ottimizzazione, non solo di scheduling; tuttavia, si tratta quasi sempre di algoritmi con complessità esponenziale, e quindi adatti solo per problemi di dimensioni ridotte.

2.3.4 Algoritmi approssimati o euristici[4]

Le euristiche sono spesso l'unica tecnica possibile per risolvere – in tempi ragionevoli – problemi NP-hard di grandi dimensioni. Non vi è alcuna garanzia che la soluzione trovata da questa categoria di algoritmi sia ottima; tuttavia, essi richiedono un tempo di calcolo relativamente basso (polinomiale), e mediamente la qualità della soluzione prodotta è molto elevata (alcune tecniche di approssimazione garantiscono una qualche differenza minima tra la soluzione trovata e quella ottima).

Gli algoritmi euristici spesso mescolano metodi di approssimazione con classiche tecniche costruttive o enumerative (ad es. branch and bound); in molti casi cercano una soluzione tra gli schedule attivi o non-delay (gli schedule non-delay, generalmente non ottimi, sono spesso molto buoni).

Gli *approcci ad agenti*, come quello sviluppato in questa tesi, rientrano nella categoria delle tecniche euristiche.

2.4 Scheduling e produzione Just-In-Time

Il *Just-In-Time* (**JIT**) è una filosofia produttiva derivante dal modello aziendale giapponese che vede la costruzione oppure l'assemblaggio dei prodotti all'atto dell'ordine. Il Just-In-Time tende ad evitare la produzione anticipata ed a

[4] I due termini non sono sinonimi, ma nell'ambito dei problemi di scheduling vengono spesso considerati intercambiabili

ridurre al minimo l'immagazzinamento, secondo la logica "zero scorte, zero difetti". Principali caratteristiche di un sistema JIT sono [Moscardi]:

- vasta gamma di prodotti offerti
- lotti di dimensioni ridotte
- consegne frequenti e rapide

Per realizzare un sistema di produzione JIT efficiente è necessario sincronizzare gli impianti produttivi con le attività commerciali e l'intera *supply chain* (catena degli approvvigionamenti): in questo modo si ottengono numerosi vantaggi, tra cui:

- aumento del throughput produttivo
- migliore servizio al cliente
- riduzione delle giacenze
- minimizzazione dei tempi di trasporto delle merci dai centri di produzione ai clienti
- riduzione del *Work In Process*[5] (*WIP*) e quindi anche del costo di immobilizzo totale dei materiali.

In alcuni casi reali, minimizzare le scorte a magazzino non è solo utile, ma addirittura indispensabile: si pensi ad esempio ad un'industria alimentare (elevate scorte implicano grandi costi di mantenimento – soprattutto in caso di generi alimentari deperibili) o ad un'impresa che assembla computer (l'hardware diventa obsoleto molto rapidamente).

[5] Indica il "carico" dell'impianto, ovvero la quantità di input che l'impianto ha iniziato a processare ma che non è stata ancora completata. Spesso è indicato in valuta (valore del materiale in corso di lavorazione), anche se può essere espresso in qualsiasi altra unità che rappresenti il flusso delle operazioni.

L'adozione di tecniche Just-In-Time ha un'influenza diretta sullo scheduling, ed in particolare sulle funzioni obiettivo adottate. Nella maggior parte dei problemi di scheduling tradizionali si considerano soltanto obiettivi regolari – ad esempio minimizzazione del ritardo (tardiness) rispetto alla data di consegna – che consentono di semplificare la ricerca di una soluzione (ottima o approssimata): tuttavia, in un contesto Just-In-Time questo non è più possibile. Le esigenze del JIT (elevata efficienza, riduzione delle scorte a magazzino) obbligano a considerare anche penalità derivanti dall'*anticipo* di un job rispetto alla sua data di consegna. L'anticipo è l'earliness E_j definita in §2.1.2, e una tipica funzione obiettivo che tenga conto sia dell'earliness che della tardiness è quella già indicata in §2.2.3 e utilizzata in questa tesi (§5.3).

Il problema con questo approccio è che, similmente a quanto accade per molti problemi di scheduling, i problemi E/T (ovvero quelli che considerano sia earliness che tardiness) sono quasi sempre NP-hard, e per risolverli è necessario affidarsi a tecniche euristiche.

2.4.1 Scheduling E/T: breve analisi della letteratura

La maggior parte della letteratura si concentra su obiettivi regolari (si vedano [Shmoys95] e [Sgall98]), tuttavia i problemi di scheduling E/T hanno recentemente ricevuto una maggiore attenzione, in quanto associati con la produzione JIT.

Il problema di scheduling E/T a singola macchina è stato studiato per la prima volta da [Kanet81]; in seguito, molti modelli simili sono stati proposti e se ne può trovare un elenco in [Baker90].

I problemi studiati in letteratura si dividono generalmente in due categorie:

1 quelli che considerano le *due date* come variabili decisionali da determinare

2 quelli caratterizzati da *due date* fissate

Per il tipo (2), la maggior parte dei modelli presi in esame sono problemi a singola macchina; il modello viene esteso alle macchine parallele in [Hall86], ma, come in molti altri testi, il problema viene semplificando introducendo una *due date* comune a tutti i job. Quest'ultimo problema può essere risolto con una tecnica costruttiva chiamata "schedule a V": i job vengono ordinati in ordine crescente di durata (opportunamente pesata), quindi vengono assegnati alla macchina, partendo dal primo della lista, uno in modo che termini prima della *due date*, un altro che termini dopo, e così via. I job completati prima della *due date* vengono ordinati secondo la tecnica LPT (Longest Processing Time) in modo che l'ultimo termini esattamente alla *due date*, viceversa per i job che terminano dopo la *due date*.

Tale tecnica è applicabile se le *due date* (che in un caso reale saranno distinte per ogni job) sono grandi abbastanza da non influenzare l'assegnazione dei job che devono essere completati prima di esse. D'altra parte, i problemi con *due date* ristrette sono computazionalmente intrattabili anche nel caso si consideri una sola macchina [Hall91].

I problemi E/T con *due date* distinte non risultano essere molto studiati; è stato dimostrato che il problema a singola macchina con penalità simmetriche per earliness e tardiness è NP-hard [Garey98]. Recentemente è stato proposto un approccio genetico per un problema E/T caratterizzato da *ready time* e *due date* distinti, e tempi di set-up dipendenti dalla sequenza [Sivrikaya99].

2.5 Scheduling on-line e competitività

Come si è detto in §2.2.5, un problema di scheduling è dinamico quando i dati riguardanti i job (processing time, *due date*, ecc.) non sono noti a priori. I problemi dinamici sono generalmente risolti con algoritmi *on-line*: un algoritmo si dice on-line quando le decisioni prese dipendono soltanto dalla conoscenza attuale del sistema e dalla sua storia passata, altrimenti si dice *off-line*.

Un algoritmo in grado di risolvere un problema di scheduling dinamico è un algoritmo on-line: infatti, i job diventano noti a partire da un certo istante temporale a_j (*arrival time*) diverso per ogni job j; e le decisioni prese al tempo τ si basano soltanto sui job tali che $a_j < \tau$.

Studiare le prestazioni degli algoritmi on-line è più complesso rispetto ad altri tipi di scheduling: un approccio molto comune è rappresentato dall'*analisi di competitività* (*competitive analysis*), una tecnica definita per la prima volta da [Sleator85] basata sul confronto tra le prestazioni di un algoritmo on-line rispetto ad un algoritmo *ottimo* off-line.

Essenziale per questo tipo di analisi è la definizione di rapporto di competitività (*competitive ratio*): in letteratura è possibile trovare più definizioni, diverse tra loro a seconda del particolare problema considerato ([Deng00], [Yeh95]); nel seguito si tenterà di fornire una definizione generale.

Data una certa classe di problemi, il rapporto di competitività *c* si definisce come il rapporto tra il costo massimo (cioè il valore massimo della funzione obiettivo) dell'algoritmo on-line rispetto al costo ottimo di un algoritmo off-line che risolva lo stesso problema.

Un algoritmo on-line si dice *c-competitivo* (dove c è il rapporto di competitività), se per ogni istanza del problema il costo è superiormente (e asintoticamente) limitato dal costo ottimo di quell'istanza moltiplicato per c. [Achliop00].

Nel seguito vengono formalizzate le definizioni date in precedenza. Adottando la seguente notazione:

- I è un'istanza del problema
- S è uno schedule per I, calcolato dall'algoritmo on-line
- S_{opt} è lo schedule ottimo per I, calcolato off-line.

- C(a, b) è il valore della funzione obiettivo calcolata per lo schedule a dell'istanza b.

si può definire il rapporto di competitività come:

$$c = \max_{I} \frac{C(S, I)}{C(S_{opt}, I)}$$

Analogamente, un algoritmo si dice c-competitivo se:

$$\forall I \qquad C(S, I) \leq c \cdot C(S_{opt}, I) + \beta$$

con β costante.

L'analisi di competitività è vantaggiosa in quanto consente di studiare le prestazioni degli algoritmi on-line senza bisogno di introdurre ipotesi probabilistiche sull'input (cioè sulle istanze considerate). Uno svantaggio è dato dal fatto che tale analisi può risultare molto pessimistica rispetto ai casi reali, dal momento che considera sempre il caso peggiore (*worst case*) per ogni istanza possibile.

Identificare il valore del costo massimo – ovvero il caso peggiore – non è semplice, soprattutto perché il costo dell'algoritmo on-line è diverso a seconda della specifica istanza considerata: per questo si preferisce considerare il *valor medio* del rapporto competitivo (*Average Competitive Ratio* o *ACR*), solitamente valutato considerando un numero elevato ma finito di istanze I del problema.

Un esempio di analisi di competitività per un problema di scheduling on-line verrà presentato nel Capitolo 6.

Capitolo 3: Sistemi ad agenti

3.1 Gli agenti

3.1.1 Introduzione

I concetti dell'intelligenza artificiale, ovvero l'idea di riprodurre il comportamento umano all'interno di entità software, interessano da molti anni un grande numero di ricercatori; in questo ambito sono state proposte numerose nuove teorie, e una delle più interessanti è la *teoria degli agenti*.

Un agente (*agent*) è considerato come una particolare entità software in grado di agire in modo autonomo prelevando informazioni dall'ambiente in cui si trova e agendo secondo la propria base di conoscenze, di scambiare informazioni con altri agenti o con esseri umani, e di prendere l'iniziativa per soddisfare i propri obiettivi. Un agente è detto autonomo o intelligente (*autonomous agent* o *intelligent agent*) quando ha necessità di operare in un ambiente dinamico ed imprevedibile, nel quale le azioni hanno una certa probabilità di fallire e devono essere prese rapidamente [Wool99]. Oltre a quella qui presentata, in letteratura si trovano numerose definizioni di agente: in §3.1.2 sono riportate le più citate.

Strettamente correlato alla teoria degli agenti è il concetto di sistemi multi-agente, descritti in §3.2. Tali sistemi sono considerati un approccio interessante per costruire sistemi intelligenti distribuiti – l'interesse in questo campo è motivato dai vantaggi che in genere comporta l'adozione della tecnologia ad

agenti: tra le principali ricordiamo la fault tolerance, la flessibilità ed l'alto parallelismo.

3.1.2 Definizioni di agente

Quando si parla di sistemi ad agenti, la prima cosa da chiarire è che non esiste una definizione univoca e universalmente accettata di *agente*. In questa sezione vengono prese in considerazione alcune delle più diffuse definizioni, cominciando dalla più semplice possibile: quella linguistica.

> Agente
>
> 1. agg. Che provoca un effetto determinato mediante la propria azione.
>
> 2. s.m. Qualsiasi ente o individuo che si determini attraverso una data azione o operazione.
>
> 3. s.m. e f. Operatore incaricato di compiti o uffici determinati per conto o alle dipendenze di terzi.
>
> [DevOli]

I significati qui elencati si ritrovano quasi sempre nelle definizioni "informatiche" di agente che i vari ricercatori hanno elaborato nel corso del tempo.

Storicamente, il concetto di agente deriva dall' "actor model" (modello degli attori), nato negli anni '70 nell'ambito dell'Intelligenza Artificiale Distribuita (*DAI, Distributed Artificial Intelligence*). In [Hewitt77] un attore è definito come un'entità software che

> "possiede un indirizzo di posta e un comportamento. Gli attori comunicano scambiandosi messaggi e agiscono concorrentemente".

Secondo [Agha86], un attore è costituito da due componenti:

- *mailbox*: una casella di posta dove vengono mantenuti i messaggi in attesa di essere elaborati. La consegna dei messaggi è garantita mentre non è garantito un tempo massimo per tale consegna;

- *behaviour*: una lista di metodi per elaborare i messaggi contenuti nella mailbox. Ogni metodo è associato ad un particolare tipo di messaggio.

Durante l'esecuzione un attore può creare nuovi attori o cambiare il proprio behaviour, cioè la procedura con cui vengono elaborati i messaggi.

Il concetto di attore si è evoluto nel concetto di agente. Oggi, una delle più diffuse definizioni di agente è quella già vista nel Capitolo 1:

"Un sistema informatico situato in un certo ambiente, capace di azioni autonome in tale ambiente allo scopo di raggiungere i propri obiettivi". [Wool95]

La definizione data da Pattie Maes è più specifica:

"Sistema computazionale che possiede obiettivi, sensori ed attuatori, e che decide autonomamente quali azioni intraprendere" [Maes96]

Altre definizioni specificano il ruolo che un agente ha nei confronti degli utenti. La seguente è fornita in [Cybele] e riassume in sé i contenuti di altre definizioni ([Satapathy] e [Shoham], oltre a [Wool95]).

"Un'entità software persistente che riceve ed invia segnali e/o eventi, agendo in modo autonomo per conto dell'utente".

Le definizioni come le ultime due sono anche chiamate "*definizioni sensor-effector*" (il termine "persistente" distingue un agente da una semplice subroutine).

In [Franklin96] sono elencate altre otto diverse definizioni di agente, oltre a quelle sopra citate. Come si può vedere, è difficile definire esattamente che cosa sia un agente (un po' come spesso è difficile definire cosa sia l'intelligenza

artificiale): per questo motivo, molti testi rinunciano a dare una definizione formale e precisa di agente e preferiscono concentrarsi su altri aspetti, ovvero:

- le caratteristiche proprie degli agenti (o meglio, quelle caratteristiche che dovrebbero possedere).
- le diverse tipologie di agenti esistenti
- le caratteristiche dell'ambiente in cui gli agenti operano

3.1.3 Caratteristiche, tipologie, ambienti

Caratteristiche degli agenti

Poiché non c'è accordo circa la definizione di agente, è più semplice descrivere le caratteristiche che un agente dovrebbe avere per essere tale, e che differenziano gli agenti da altri tipi di applicazioni.

L'attributo principale che caratterizza gli agenti è l'*autonomia*: gli agenti devono avere la capacità di eseguire azioni legate a particolari processi e obiettivi, senza interventi da parte dell'utente; inoltre l'esistenza di un agente (o, come spesso si dice, la sua *persistenza*) non deve dipendere da altri agenti. Due esempi di autonomia sono i seguenti [Odell00]:

- *autonomia dinamica*, o capacità di dire "go": l'abilità di "prendere l'iniziativa" per soddisfare i propri obiettivi.
- *autonomia deterministica*, o capacità di dire "no": se un agente richiede ad un altro agente di effettuare un'azione, quest'ultimo può rifiutarsi.

L'autonomia dinamica è più spesso chiamata "*proattività*" (*proactiveness* o *proactivity*) in contrapposizione alla reattività, ovvero la capacità di agire in

risposta agli stimoli esterni. Si noti che l'autonomia non impedisce che un agente persegua obiettivi per conto dell'utente.

Oltre all'autonomia, le principali caratteristiche proprie degli agenti sono le seguenti ([Maes96], [Wool99]):

- *abilità a comunicare*: gli agenti devono accedere ad informazioni provenienti da altre sorgenti (non necessariamente altri agenti, ma anche basi di dati, pagine web, persone, directory service, ecc.) circa lo stato di ambienti esterni ed essere in grado di scambiarsele;

- *capacità di cooperazione*: per eseguire processi complessi e perseguire obiettivi comuni gli agenti devono lavorare insieme;

- *capacità di ragionamento*: un agente deve possedere l'abilità di decidere sulla base della sua conoscenza ed esperienza;

- *capacità di adattamento*: l'agente deve avere dei meccanismi per giudicare lo stato attuale del suo dominio esterno ed incorporare questo all'interno di decisioni circa azioni future (tale caratteristica è nota anche come apprendimento);

- *sincerità*: l'utente e gli altri agenti devono essere sicuri che un dato agente agirà e riporterà in maniera del tutto sincera l'informazione a sua disposizione. Inoltre un agente deve agire per il bene del suo proprietario.

Quelle indicate non sono ovviamente tutte le possibili caratteristiche. Ad esempio, [Maes96] afferma che un agente dovrebbe essere *long-lived*, ovvero "vivere a lungo": è un modo diverso per esprimere l'autonomia e per distinguere un agente da una subroutine (stesso significato del termine "persistente" visto nella sezione precedente).

Non sempre per un agente è obbligatorio possedere tutti questi attributi: un esempio è dato dalla capacità di apprendimento: in molte applicazioni l'apprendimento può non essere necessario o risultare addirittura nocivo.

Normalmente un agente avrà un repertorio di azioni disponibili: tale insieme di azioni rappresenta l'*effector capacity* dell'agente. Il problema chiave nella definizione di ogni agente è stabilire quale delle sue azioni debba compiere per raggiungere nel miglior modo possibile, rispetto a qualche predefinita funzione di costo o guadagno, l'obiettivo preposto.

Tipologie di agenti

Un documento interessante per comprendere lo stato dell'arte per quanto riguarda le ricerche nel campo degli agenti è [Nwana96], che definisce diverse tipologie di agenti e analizza nel dettaglio ipotesi, motivazioni, obiettivi e problemi che li caratterizzano.

Le sette tipologie di agenti esaminate in [Nwana96] sono le seguenti:

- *Agenti collaborativi* (collaborative agents): sono agenti autonomi che hanno la capacità di cooperare con altri agenti; possono possedere capacità di apprendimento oppure no. Sono strettamente legati ai sistemi multi-agente (vedi §3.2).

- *Agenti d'interfaccia* (Interface agents): agenti che hanno il compito di gestire il dialogo con l'utente. Sono autonomi ed hanno capacità di apprendimento, ma in genere non comunicano con altri agenti. Spesso sono chiamati *personal assistants*.

- *Agenti mobili* (Mobile agents): agenti in grado di spostarsi (migrare) da un computer all'altro (attraverso una rete) durante la loro stessa esecuzione. Gli agenti mobili costituiscono un argomento di ricerca interessante ma che rappresenta una percentuale piuttosto piccola dell'intera teoria degli agenti.

- *Information/Internet agents*: agenti per la gestione di informazioni provenienti da varie fonti, che possono essere numerose ed eterogenee, spesso prelevate da Internet interagendo con motori di ricerca.

- *Agenti reattivi* (Reactive agents): agenti dotati soltanto di reattività e non di proattività. Non possiedono una rappresentazione interna simbolica dell'ambiente esterno, ma agiscono solo secondo il principio richiesta-risposta.

- *Agenti ibridi* (Hybrid agents): agenti che combinano due o più delle tipologie sopra indicate.

- *Agenti intelligenti* (Smart agents): agenti dotati di autonomia, capacità di collaborazione e di apprendimento.

La definizione precisa del termine *intelligente* dipende dal contesto in cui ci si trova ad operare: solitamente indica che l'agente persegue i propri obiettivi ed esegue i propri compiti in modo da ottimizzare una data misura di prestazione. Il fatto che gli agenti siano intelligenti non significa che essi non commettono errori, indica piuttosto che essi operano in maniera flessibile e razionale in un certo numero di circostanze ambientali, date le informazioni possedute ed una certa capacità percettiva ed attuativa [Wool95] (è ovvio che non esistono agenti intelligenti *in assoluto*).

L'ambiente in cui operano gli agenti

Russell e Norvig [Russel95] hanno suggerito la seguente classificazione delle proprietà dell'ambiente in cui gli agenti operano:

- accessibilità o non accessibilità;

- determinismo o non determinismo;

- episodicità o non episodicità: avvenimenti di tipo episodico sono più semplici d'affrontare in quanto ogni agente può decidere quale azione compiere basandosi unicamente sull'episodio corrente;

- staticità o dinamicità;

- discreto o continuo: il movimento delle pedine su una scacchiera è un esempio di moto in ambiente discreto, mentre la guida di un veicolo è un esempio di moto in ambiente continuo.

Se un ambiente è sufficientemente complesso, il fatto che esso sia deterministico non è di molto aiuto. Le classi di ambienti generalmente più complesse (che corrispondono alla maggior parte dei casi reali) sono quelle inaccessibili, non deterministiche, non episodiche, dinamiche e continue.

3.1.4 Agenti e sistemi esperti

I sistemi esperti, detti anche *expert systems* o *knowledge-based systems*, hanno la caratteristica fondamentale di essere in grado di eseguire compiti, di solito affidati a una persona esperta in un particolare dominio. Poiché l'esperto è colui il quale possiede notevole conoscenza ed esperienza in un ambito ben definito ed è capace, in tale ambito, di dare risposte corrette, il sistema esperto deve essere in grado di emularne l'operato, in particolar modo compiendo le stesse azioni, fornendo gli stessi giudizi ed esibendo le stesse spiegazioni. L'introduzione del termine "sistema esperto" è datata 1977, e la sua precisa definizione è la seguente:

"Un sistema esperto è un programma di calcolatore che usa conoscenze e tecniche di ragionamento per risolvere problemi che normalmente richiederebbero l'aiuto di un esperto. Un sistema esperto deve avere la capacità di giustificare o spiegare il perché di una particolare soluzione per un dato problema." [Feigenbaum]

I sistemi esperti sono diversi dagli agenti (solitamente non sono autonomi né hanno capacità di collaborazione con altri sistemi esperti), tuttavia i due concetti sono strettamente correlati, tanto che è spesso utile visualizzare un agente come un'estensione del concetto di sistema esperto.

3.2 Sistemi multi-agente

Un sistema multi-agente, come si è già accennato nel Capitolo 1, è una comunità sociale di membri interdipendenti che agiscono individualmente. Una definizione più precisa è data da [Shen99]:

> "un sistema in cui alcuni agenti intelligenti interagiscono per soddisfare un certo insieme di obiettivi, allo scopo di portare a termine un certo insieme di compiti"

Gli agenti esistono ed operano in qualche ambiente che tipicamente è sia computazionale sia fisico. Tale ambiente può essere aperto o chiuso e può contenere altri agenti. In certe situazioni gli agenti possono operare singolarmente, ma generalmente essi interagiscono con altri agenti: se gli agenti sono molto numerosi è difficoltoso trattarli individualmente, è quindi più conveniente trattarli collettivamente, cioè come *società di agenti* o *agenzia* (*agency*).

L'ambiente fornisce l'infrastruttura computazionale per l'interazione fra gli agenti; tale infrastruttura comprende i protocolli per la comunicazione e per l'interazione; i *protocolli di comunicazione* permettono agli agenti di scambiare e comprendere messaggi, quelli di *interazione* rendono gli agenti in grado di avere conversazioni strutturate come scambio di messaggi.

Le caratteristiche principali degli ambienti multi-agente sono le seguenti:

- forniscono una infrastruttura che specifica i protocolli di comunicazione e interazione;
- sono aperti;
- contengono agenti autonomi che possono perseguire ognuno i propri interessi oppure essere cooperativi.
- ciascun agente ha informazioni e capacità limitate
- il sistema di controllo è distribuito

- i dati sono decentralizzati
- la computazione è asincrona.

I principali vantaggi offerti dai sistemi multi-agente rispetto a sistemi tradizionali sono:

- *velocità ed efficienza* (gli agenti possono operare in modo asincrono ed in parallelo)
- *robustezza e affidabilità* (il fallimento o la non-disponibilità di uno o più agenti non necessariamente rende inservibile l'intero sistema)
- *scalabilità e flessibilità* (il sistema può essere adottato in un problema di dimensione accresciuta aggiungendo agenti senza necessariamente alterare l'azione degli altri; può inoltre interagire con sistemi già esistenti in maniera non invasiva)
- *costi* (può essere composto di semplici sottosistemi di basso costo e, quindi, il costo è inferiore a quello di un sistema centralizzato)
- *sviluppo e riutilizzo* (ciascun agente può essere sviluppato separatamente e indipendentemente dal contesto in cui sarà utilizzato. Il sistema totale potrà essere testato e mantenuto più semplicemente ed anche riconfigurato e, inoltre, gli stessi agenti potranno essere utilizzati in uno scenario diverso).

3.2.1 Comunicazione e coordinazione tra agenti

Gli agenti comunicano (si scambiano e condividono informazioni) onde conseguire nel miglior modo possibile i propri scopi o quelli della società/sistema cui appartengono. Ad esempio, un agente può scomporre il proprio compito (*task*)

in task di dimensioni più piccole e delegarli ad altri agenti che non conoscono l'obiettivo (*goal*) del delegante.

La seguente figura schematizza i diversi tipi di coordinazione:

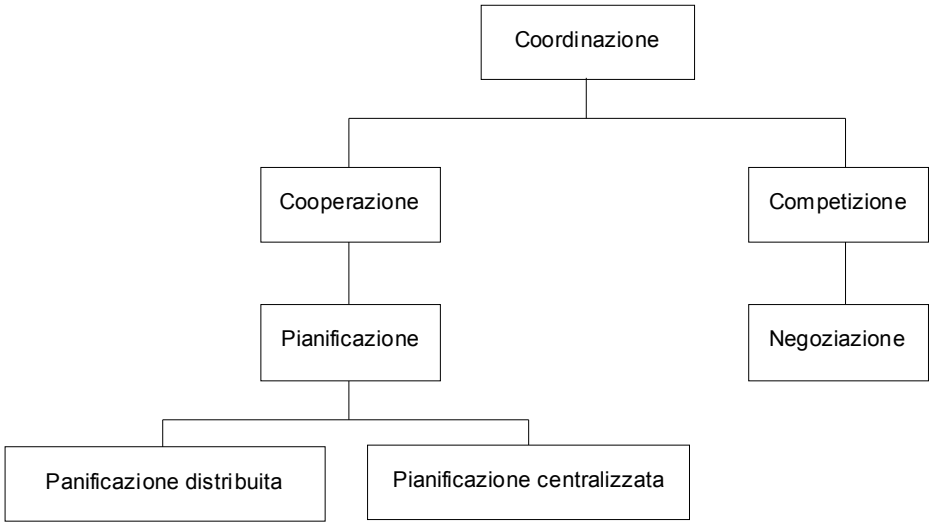


Figura 3-3.1 Coordinazione tra agenti

La **cooperazione** è la coordinazione fra agenti non antagonisti, mentre la **negoziazione** è la coordinazione fra agenti competitivi o semplicemente fra agenti che perseguono i propri interessi. Tipicamente, affinché la cooperazione abbia successo ogni agente deve mantenere un modello degli altri agenti e sviluppare un modello delle future interazioni; questo presuppone la socievolezza.

Un sistema multi-agente deve mantenere una coerenza globale senza un esplicito controllo centralizzato: gli agenti devono essere capaci di determinare per conto proprio gli obiettivi che condividono con altri agenti, i compiti comuni, evitare conflitti non necessari e mettere in comune la conoscenza (per questo motivo, nell'ambito dei sistemi multi-agente, solitamente non è prevista la "pianificazione centralizzata").

3.2.2 Messaggi

I messaggi, chiamati spesso "*atti comunicativi*" ("*communication acts"* o "*communicative acts*" o anche "*speech acts*"), si dividono principalmente in messaggi di affermazione e di domanda. Ogni agente, sia esso attivo o passivo dal punto di vista della comunicazione, deve essere in grado di ricevere informazioni che, nel caso più semplice, provengono da una sorgente esterna con significato di affermazione. Un agente che nel dialogo ha ruolo passivo deve inoltre avere la capacità di rispondere a domande (per es. deve riuscire ad eseguire i seguenti due passi: 1) accettare la domanda proveniente dalla sorgente esterna e 2) mandare una risposta a tale sorgente formulando un'affermazione).

Al fine di assumere un ruolo attivo nel dialogo, un agente deve emettere domande e formulare affermazioni, in modo tale da poter controllare un altro agente costringendolo a rispondere a una domanda o ad accettare una determinata informazione. Un agente che si trova allo stesso livello di un altro (*peer*) può assumere nel dialogo sia un ruolo passivo sia un ruolo attivo, deve quindi essere in grado di formulare e accogliere sia affermazioni sia domande.

I messaggi devono essere espressi in un ben determinato *linguaggio*, comprensibile a tutti gli agenti che compongono un sistema: tale linguaggio è spesso denominato **ACL** – Agent Communication Language. Come esistono moltissime definizioni di agente, esistono anche numerosi tipi di ACL, tutti molto diversi e incompatibili tra di loro.

Uno dei linguaggi più noti per la comunicazione tra agenti – ma anche tra generiche applicazioni – è il **KQML** (Knowledge Query and Manipulation Language), linguaggio derivato dal LISP (si veda a questo proposito [McCarthy]).

Il KQML è stato elaborato nel 1993 e, anche se è tuttora molto diffuso, sta cedendo il passo ad un nuovo linguaggio, molto più evoluto: **FIPA-ACL** (vedi §3.4.7).

3.2.3 Protocolli di interazione

I protocolli di interazione (IP, Interaction Protocols) governano lo scambio di *serie di messaggi* fra agenti. Sono costituiti da un insieme di regole alle quali gli interlocutori devono attenersi, affinché le loro conversazioni assumano un preciso significato; si tratta di specifiche ad alto livello, che consentono allo sviluppatore di definire un sistema ad agenti senza preoccuparsi degli effettivi protocolli di trasporto utilizzati.

Sono stati proposti svariati protocolli di interazione per i sistemi di agenti: se questi ultimi hanno obiettivi conflittuali, o semplicemente perseguono ognuno i propri interessi, lo scopo del protocollo è massimizzare i profitti degli agenti; nel caso in cui gli agenti abbiano obiettivi simili o problemi comuni, lo scopo del protocollo è mantenere una performance globalmente coerente, cioè senza violazioni di autonomia, per esempio, o senza un esplicito controllo globale [Durfee91].

Protocolli di coordinazione

Nell'ambito dell'intelligenza artificiale distribuita, la ricerca volta allo sviluppo di sistemi coordinati si è concentrata principalmente sulle tecniche per la distribuzione sia del controllo sia dei dati.

Controllo distribuito significa che gli agenti hanno un certo grado di autonomia riguardo alla generazione di nuove azioni e alla scelta dei prossimi obiettivi da conseguire. Lo svantaggio di questo approccio è nel fatto che la conoscenza sullo stato del sistema è dispersa nel sistema stesso e ogni agente ha una visione solo parziale che potrebbe anche essere imprecisa; tale aumento di incertezza implica una maggiore difficoltà ad ottenere un comportamento globale coerente.

Protocolli di cooperazione

La strategia base condivisa da molti protocolli di cooperazione consiste nella scomposizione e successiva distribuzione dei task: questo approccio può ridurre la complessità di un compito, inoltre sotto-operazioni più piccole richiedono minore "abilità" da parte degli agenti e necessitano di meno risorse.

Se sono applicabili più tecniche di scomposizione, il sistema deve decidere quale tipo di scomposizione operare tenendo conto del fatto che il processo prescelto deve essere compatibile con le capacità degli agenti e con le risorse a disposizione. Inoltre possono esserci interazioni fra i sottotask e conflitti fra gli agenti. La scomposizione dei task può essere effettuata da chi progetta il sistema e quindi programmata durante l'implementazione, oppure può essere effettuata dagli stessi agenti secondo una pianificazione gerarchica, può essere di tipo spaziale o funzionale.

Esistono poi vari criteri per la distribuzione dei compiti, per esempio può essere proprio un agente ad assegnare agli altri le finalità, le quali possono sovrapporsi in modo da avere coerenza; inoltre quelle assegnate ad agenti prossimi a livello spaziale o semantico possono essere strettamente interdipendenti (questo massimizza i costi di comunicazione e sincronizzazione).

Il più noto e diffuso meccanismo di distribuzione dei task è il *Contract net* [Huhns99], un protocollo per l'interazione di agenti cooperanti. Tale protocollo è modellato sulla contrattazione che governa lo scambio di beni e servizi e fornisce una soluzione al cosiddetto "problema di connessione", la determinazione cioè dell'agente più adatto a svolgere un certo compito. Una descrizione del Contract Net verrà data in §3.6.3, dove si parlerà anche delle tecniche che consentono di specificare nel dettaglio un qualsiasi protocollo di interazione.

3.2.4 Ontologie

L'*ontologia* è la descrizione degli oggetti, dei concetti e delle relazioni in una certa area di interesse. Una definizione precisa è data da [FIPA01]:

“Un insieme di simboli associati ad un’interpretazione che può essere condiviso da una comunità di agenti. Un’ontologia include un vocabolario di simboli che si riferiscono ad oggetti in un dato dominio, insieme ad altri simboli associati alle relazioni che caratterizzano tale dominio.”

I concetti possono essere rappresentati nella logica del primo ordine come predicati monadici, mentre predicati più complessi rappresentano le relazioni.

Ogni agente deve rappresentare la propria conoscenza nel vocabolario di una specifica ontologia; in questo modo gli agenti che condividono la stessa ontologia per la rappresentazione della conoscenza comprendono le “parole” del linguaggio di comunicazione utilizzato.

3.3 Applicazioni dei sistemi ad agenti

La crescente necessità di gestire sistemi distribuiti (basti pensare all’affermazione e alla diffusione di Internet come a uno degli esempi più lampanti) ha fatto sorgere l’esigenza di applicare un nuovo approccio che permetta di gestire ed utilizzare tutte le informazioni acquisibili, ovunque esse si trovino, in modo dinamico e, soprattutto, efficace. Sono così nate e si sono sviluppate le tecniche di programmazione distribuita, favorite anche dall’affermazione dei linguaggi orientati agli oggetti che promuovono la programmazione *multi-thread*. Gli agenti software rappresentano una tecnica innovativa che permette, tra le altre cose, di automatizzare e rendere più efficace la fornitura di un servizio. Questo avviene perché è il sistema ad agenti che si fa carico di tutte quelle operazioni necessarie allo svolgimento di un determinato compito ed è potenzialmente in grado di fornire servizi più evoluti rispetto a quelli realizzati con tecnologie software più tradizionali.

Rispetto ai sistemi tradizionali nei quali l’“intelligenza” è centralizzata, un’architettura distribuita, nella quale le capacità elaborativa e decisionale sono condivise e distribuite tra più entità (agenti) può offrire molti vantaggi, come ad esempio interfacce di comunicazione semplici e maggiore affidabilità del sistema.

La struttura dei meccanismi decisionali e di comunicazione facilita l'implementazione in sistemi real time. Per un simile sistema si possono ipotizzare molti possibili campi applicativi: potenzialmente tutti quelli in cui si debbano affrontare problemi che coinvolgano strutture variamente complesse o criteri decisionali molto articolati.

In letteratura sono descritte numerose applicazioni industriali e commerciali, sia esistenti sia potenziali, riguardanti l'utilizzo di sistemi multi-agente, ad esempio:

- commercio elettronico e negozi virtuali
- gestione e monitoraggio in tempo reale di reti di telecomunicazioni
- immagazzinamento dell'informazione in ambienti informatici (ad esempio Internet)
- gestione del traffico urbano ed aereo
- investigazione degli aspetti sociali dell'intelligenza e simulazione di fenomeni sociali complessi.

Tra le tante applicazioni possibili, questa tesi si concentra su di una in particolare: la gestione dei sistemi manifatturieri e produttivi.

3.3.1 Applicazioni in ambito manifatturiero

Nel campo dei sistemi produttivi molti ricercatori hanno applicato la teoria degli agenti e le tecnologie multi-agente ad una grande varietà di problemi, ad esempio:

- integrazione d'impresa (enterprise integration)

- gestione degli approvvigionamenti
- pianificazione della produzione
- schedulazione e controllo
- gestione dei materiali
- Holonic Manufacturing Systems
- gestione del workflow.

È in particolare quest'ultimo aspetto che si desidera approfondire nel corso della tesi. La sessione seguente introduce i possibili usi degli agenti per la gestione del flusso di lavoro in ambito industriale.

3.3.2 Agenti e gestione del workflow

L'attività aziendale, sia vista in maniera globale sia analizzata gerarchicamente, può essere modellata come un flusso di lavoro (*workflow*), ossia un diagramma in cui sono organizzati processi che possono essere sequenziali, concorrenti o alternativi. Tali processi, in relazione al dettaglio con cui si considerano, possono corrispondere ad attività molto aggregate o al contrario molto dettagliate.

Dal punto di vista operativo il sistema si compone di un insieme relativamente numeroso di processi elementari. Il workflow management integrato per un'azienda manifatturiera corrisponde quindi al controllo del flusso di esecuzione di tali processi e delle informazioni da essi gestite.

In genere una fotografia che fornisca lo stato del workflow di un'azienda in un qualsiasi istante presenterebbe un quadro piuttosto complesso di attività in esecuzione, sospese, ritardate i cui legami reciproci non sarebbero certo evidenti. Molte attività sarebbero ad esempio istanze diverse di una stessa attività generica applicata al controllo della produzione di lotti o ordini diversi.

In questo quadro è evidente che un sistema per il supporto di ciò che genericamente si potrebbe indicare come la gestione di un sistema manifatturiero, ovvero del workflow ad esse connesso, non può risultare un qualcosa di monolitico ma più facilmente un'entità trasversale che consente, eventualmente per mezzo di strumenti preesistenti ai diversi livelli della gerarchia del modello prima citato, di interoperare. Questa soluzione consente tra l'altro una crescita graduale del sistema gestionale e in genere non richiede scelte che in tempi troppo anticipati potrebbero precludere l'adozione di strumenti specifici da integrare in tempi più maturi per l'azienda.

Il collante in un tale scenario, ovvero il sistema nervoso del sistema di gestione aziendale, può essere costituito da una serie di piccole ma ben definite entità software che, senza appesantire il sistema gestionale in termini di requisiti per rendere possibile la loro esistenza, controllano e coordinano il sistema produttivo riferendo puntualmente agli operatori umani. É evidente che tale premessa fa riferimento ad un sistema basato su agenti.

In effetti l'uso di agenti in un contesto manifatturiero può avere applicazioni diverse, in particolare a ciascuno dei livelli prima elencati, e comportare vantaggi di natura diversa.

L'adozione di architetture ad agenti (o, se si vuole, di una filosofia ad agenti) può essere percepita come una concessione ad una moda ora in voga e non risultare come una reale esigenza. Questo può forse ed in parte essere vero: le aziende manifatturiere esistono ed operano già e potrebbero certo continuare a farlo senza mai adottare tecniche ad agenti. Tuttavia la sensazione è che ci si trovi di fronte ad una possibile rivoluzione dal punto di vista degli strumenti software; un qualcosa di simile a ciò che ha caratterizzato l'adozione della filosofia dei sistemi object oriented. Gli agenti, infatti, possono giocare un ruolo simile agli oggetti non tanto rispetto ai componenti di un sistema software (dati e metodi o procedure), ma a componenti di un sistema, nel caso considerato un sistema manifatturiero. Un agente può essere associato ad un'attività dell'azienda manifatturiera (sia essa logica a livello più alto, o fisica a livello più basso),

monitorarne e controllarne l'esecuzione ed interfacciarla con altre attività, ovvero con altri agenti allo stesso livello o ad uno superiore. Agenti di natura diversa (con un più o meno elevato livello di autonomia, proattività o in genere di intelligenza) saranno responsabili di una decomposizione sia verticale (gerarchica) che trasversale (funzionale) della complessità: ogni agente avrà un compito ben definito, una conoscenza ed una strategia per eseguirlo, ed un meccanismo per comunicare e per operare in presenza di altri agenti (socialità). Similmente ad un oggetto, un agente può essere associato ad un elemento fisico e/o ad un'attività ed incapsula i metodi per svolgerla. L'utente avrà come interfaccia verso il sistema manifatturiero un insieme di agenti il cui compito sarà quello di esporre verso l'utente stesso quelle funzioni di cui necessita per operare (agire sul processo, decidere, ecc.) nascondendogli i dettagli del funzionamento istante per istante del sistema ad agenti che regola sui vari livelli il sistema produttivo.

In relazione al loro ruolo gli agenti potranno essere cooperanti o in competizione.

Evidentemente in questo scenario è facile intuire l'immediata estensione ed apertura del sistema aziendale verso la rete, ossia la possibilità introducendo nuovi elementi rappresentati da nuovi agenti di operare in stretta connessione con i propri clienti e fornitori (realizzando quindi processi di *e-commerce* e di *e-business*).

Inoltre l'azienda può estendersi geograficamente o assumere dimensioni multiple (una nuova azienda costituita da una rete di aziende che operano cooperativamente, come ad esempio avviene nei casi di outsourcing): la presenza di un'infrastruttura di rete consente l'adozione di un sistema di agenti distribuiti e, se necessario, di agenti mobili.

3.4 Standard FIPA

La Fondazione per gli Agenti Fisici Intelligenti (FIPA, "*Foundation for Intelligent Physical Agents*") è un'associazione internazionale no-profit nata a

Ginevra nel 1996 e composta da varie società ed organizzazioni con l'obiettivo di definire uno standard riguardante le tecnologie ad agenti. Lo scopo finale non è tanto quello di creare una tecnologia adatta a particolari applicazioni ma, principalmente, quello di generare dei modelli di riferimento che possano essere utilizzati dagli sviluppatori per l'implementazione di sistemi complessi, nelle aree e nei settori più disparati.

All'inizio della sua esistenza, l'obiettivo principale di FIPA erano le applicazioni robotiche (da cui il termine "physical" nella sigla FIPA), poi gradualmente l'interesse si è spostato nel campo degli agenti software.

Uno degli aspetti che più preme ai ricercatori di FIPA è il raggiungimento, utilizzando tali modelli, di un elevato livello di interoperabilità tra tecnologie differenti: l'architettura FIPA si concentra sulla definizione del "comportamento esterno" (cioè sociale) degli agenti piuttosto che su quello "interno" (il modo in cui gli agenti "ragionano" ed elaborano le informazioni ricevute dipende dalla particolare implementazione). Per questo motivo si è proposto di dare alla sigla FIPA il nuovo significato di "*Foundation for InteroPerable Agents*".

Sebbene si parli comunemente di "standard FIPA", tale denominazione non è corretta, in quanto tutte le specifiche prodotte da FIPA sono ancora ad un livello sperimentale; tuttavia esse sono ampiamente adottate da buona parte della comunità degli sviluppatori di agenti, costituendo dunque uno standard *de facto*.

3.4.1 Architettura FIPA

La prima documentazione rilasciata, denominata FIPA97 Specifications, risale al 1997 e delinea le regole di base per la gestione di una comunità di agenti, sottolineando gli aspetti relativi alla loro collaborazione. Nell'ottobre 2000 è stato approvato il nuovo insieme di specifiche cui è stato dato il nome FIPA2000. Tali specifiche sono solitamente classificate per argomenti:

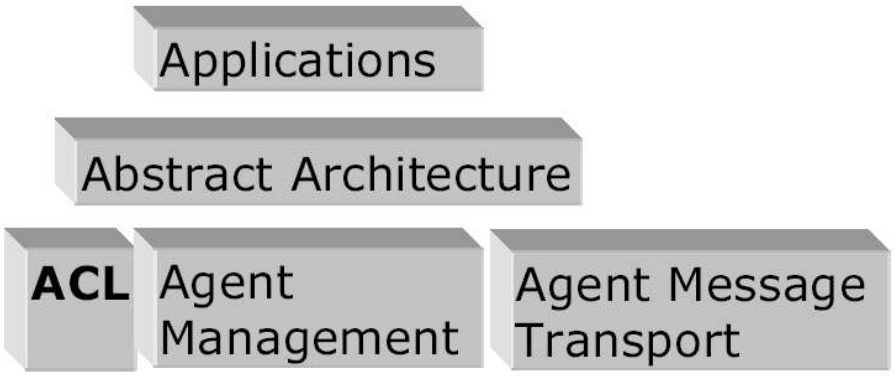


Figura 3-3.2 Categorie delle specifiche FIPA

- **Abstract Architecture**: definisce in modo astratto le entità che compongono un sistema ad agenti, anche attraverso l'uso di diagrammi UML. Si concentra soprattutto sull'interoperabilità tra gli agenti, evitando di scendere nei dettagli implementativi (ad es. non specifica la struttura interna degli agenti).

- **Agent Communication**: definisce la struttura generica del linguaggio di comunicazione tra agenti (FIPA-ACL); inoltre specifica alcuni tra i più popolari protocolli di interazione, atti comunicativi e linguaggi per la rappresentazione dei contenuti (*content languages*). I protocolli sono formalizzati con diagrammi AUML (Agent UML, descritto più avanti in questo capitolo).

- **Agent Management**: definisce un'infrastruttura generica in cui gli agenti FIPA possono esistere, operare ed interagire; inoltre specifica quali sono le entità necessarie alla comunicazione ed alla gestione degli agenti stessi. Comprende anche una descrizione del supporto per gli agenti mobili.

- **Agent Message Transport**: definisce come deve avvenire la trasmissione dei messaggi scambiati attraverso diversi protocolli di trasporto; inoltre specifica le possibili rappresentazioni concrete dei messaggi FIPA-ACL e degli involucri (envelope) in cui essi sono trasportati.

- **Applications**: si tratta di esempi di applicazioni concrete dei sistemi ad agenti, tra cui i personal assistants, la gestione della rete, il multimediale e l'integrazione degli agenti con i sistemi legacy (es. database).

Molte delle specifiche FIPA sono normative (obbligatorie per ogni implementazione) mentre altre sono informative (linee guida che non è obbligatorio seguire).

3.4.2 Caratteristiche di un agente FIPA

FIPA introduce una definizione di agente molto generica: "*Un processo computazionale che implementa le funzionalità di autonomia e comunicazione di un'applicazione*" [FIPA01].

Ogni agente è caratterizzato da un identificatore univoco chiamato *AID* (*Agent IDentifier*), una collezione ampliabile di coppie parametro/valore che comprende, oltre al nome, anche altri parametri, tra cui:

- tutti gli indirizzi che indicano dove è situato fisicamente l'agente, su quale host, quale piattaforma e quale contenitore (sono necessari per le comunicazioni e la migrazione).

- i resolver, cioè tutti quegli agenti dove l'agente in questione è registrato (vedi §3.4.4).

altri parametri, a discrezione del progettista del sistema applicativo.

Solo il nome di un agente è invariabile mentre gli altri parametri possono essere modificati durante il corso della sua "vita". Inoltre un agente supporta differenti tecniche di comunicazione ed è in grado di memorizzare indirizzi multipli (indispensabili per gli spostamenti tra contenitori o piattaforme) nel parametro *addresses* di un AID.

Ogni agente attraversa un ciclo di vita riassunto nel seguente schema:

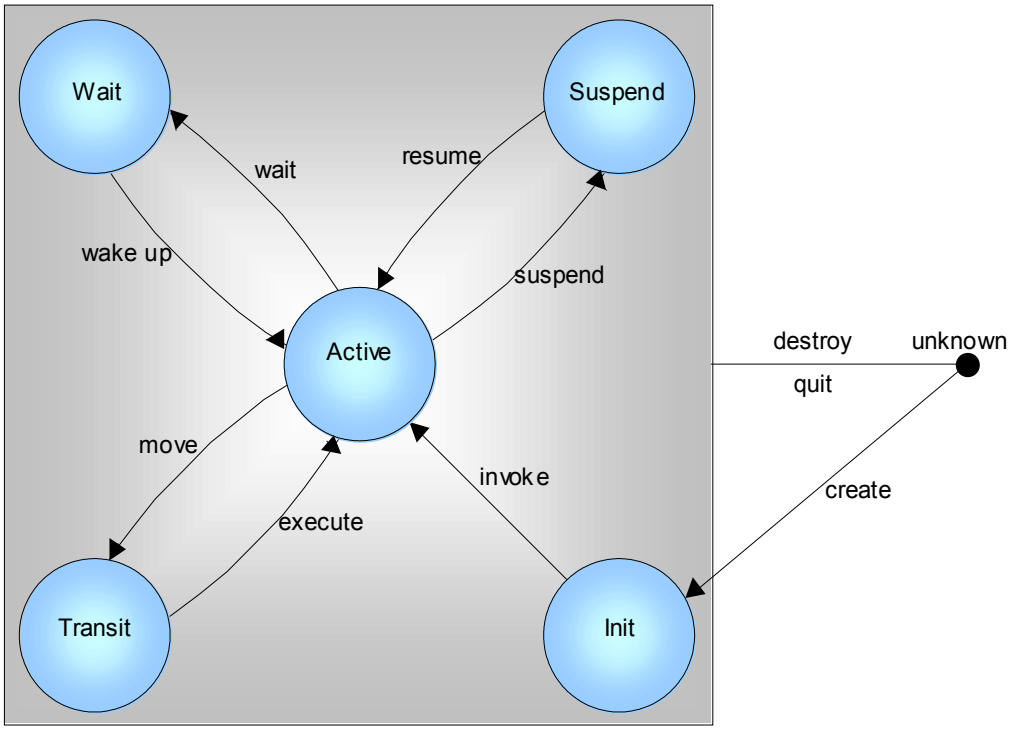


Figura 3-3.3 Ciclo di vita di un agente FIPA

3.4.3 La piattaforma ad agenti (AP)

Un'agente FIPA esiste solo in un *contenitore* di agenti presso un'apposita *piattaforma* in esecuzione: senza tutto ciò non ha senso parlare di agente.

FIPA definisce la "piattaforma ad agenti" (*agent platform* o AP) come l'ambiente nel quale gli agenti possono esistere ed interoperare. L'AP può essere vista come una realizzazione concreta dell'Abstract Architecture: è un sistema software composto da uno o più *contenitori di agenti* i quali devono appartenere, in ogni istante, ad una ed una sola AP. Gli agenti posseggono la facoltà, quando necessario, di spostarsi da una piattaforma ad un'altra (agenti mobili).

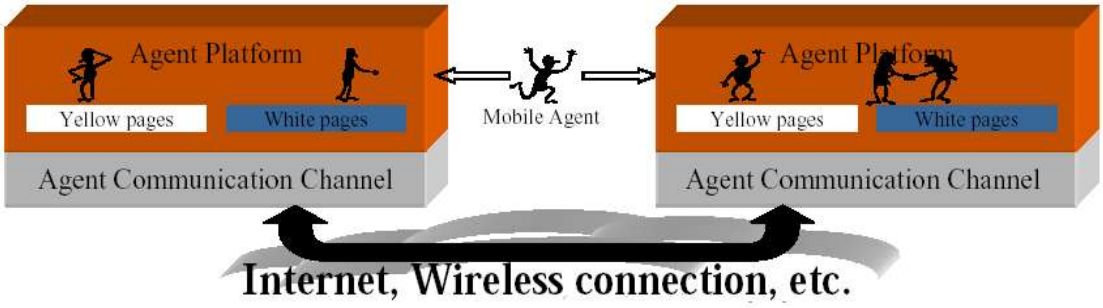


Figura 3-3.4 Piattaforme ad agenti FIPA

Una piattaforma è costituita da diverse componenti (oltre che, ovviamente, dall'hardware e dal sistema operativo):

- Software di supporto agli agenti (ad esempio i framework descritti nel Capitolo 4).
- I "FIPA agent management components", descritti nel seguito.
- Gli agenti.

Le entità fondamentali per il corretto funzionamento della piattaforma sono le seguenti tre (agent management components): AMS, DF, MTS.

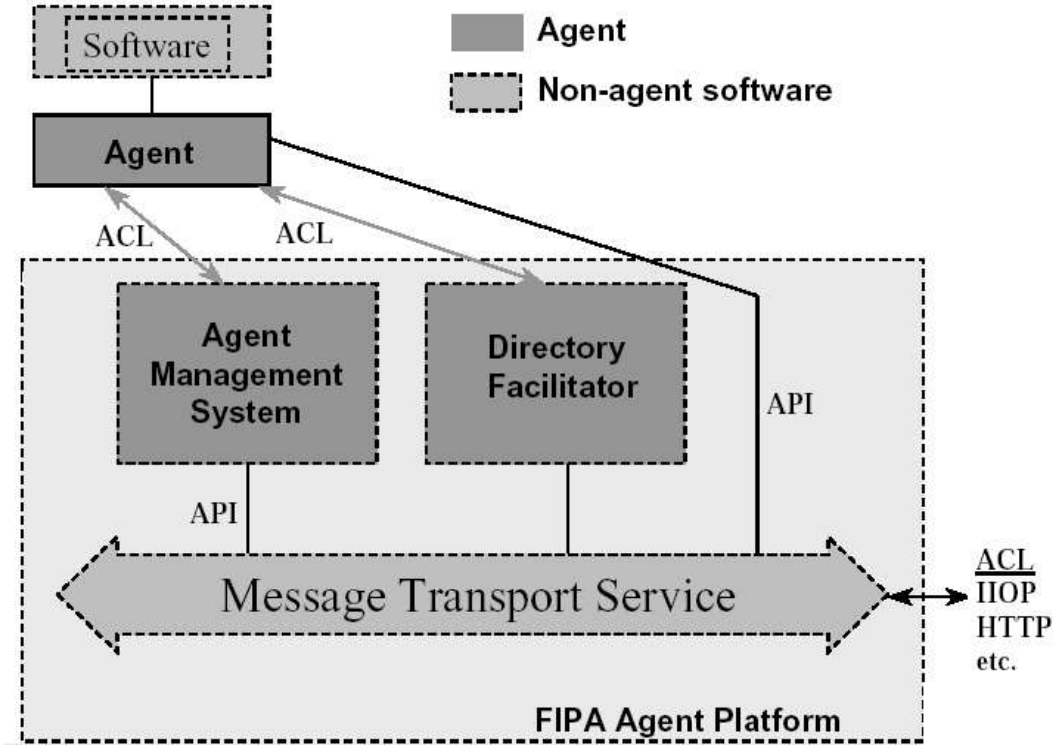


Figura 3-3.5 Struttura di un'AP

3.4.4 Agent Management System (AMS)

È il componente che esercita il compito di supervisore, controllando l'accesso e l'uso della piattaforma: è responsabile dell'identificazione e della registrazione degli agenti residenti (solitamente è a sua volta implementato come un agente).

L'AMS gestisce l'intero ciclo di vita di ogni agente e rappresenta una specie di unità di controllo all'interno di una AP. Può richiedere ad un agente di svolgere uno specifico compito e, se necessario, è in grado di forzare l'esecuzione di un particolare task. Fornisce inoltre un servizio di "elenco del telefono" (quelle che in inglese si chiamano "white pages", "pagine bianche") tenendo aggiornato un indice di tutti gli agenti presenti in ogni istante sulla piattaforma, compresi i loro AID. Tutti gli agenti devono registrarsi presso l'AMS della propria HAP (*Home Agent Platform*), cioè la AP dove sono stati creati e responsabile dell'identità degli agenti stessi. La registrazione comporta l'autorizzazione, previa

fase di autenticazione, ad accedere al MTP (*Message Transport Service*) per svolgere le mansioni di invio e ricezione di messaggi. Le descrizioni degli agenti possono essere modificate in qualsiasi istante e per molti motivi. La vita di un agente presso una AP termina con la sua deregistrazione. In questa fase il suo AID diviene disponibile per essere assegnato, eventualmente, ad un altro agente. Trattandosi di un agente, anche l'AMS possiede un AID, riservato e rappresentato nella forma seguente:

```
(agent-identifier
                :name ams@hap
                :addresses (sequence hap_transport_address))
```

Esistono due modi con cui un agente si registra presso l'AMS:

- l'agente viene creato sulla AP,
- l'agente è stato creato su un'altra AP ed è migrato su quella corrente. Vale solo per le piattaforme che supportano gli agenti mobili.

Un agente può richiedere quattro diversi servizi all'AMS:

- *Register*: realizza le operazioni di registrazione di un agente
- *Deregister*: realizza le operazioni di deregistrazione di un agente
- *Search*: realizza le operazioni di ricerca di un agente
- *Modify*: realizza le operazioni di modifica dei parametri descrittivi di un agente.

3.4.5 Directory Facilitator (DF)

Quando un agente desidera far conoscere le proprie funzionalità ad altri agenti, si serve del Directory Facilitator (DF). Il DF fornisce un servizio di

"pagine gialle" ("yellow pages"), ovvero mantiene una lista di agenti dando origine ad un dominio (AD, Agent Domain), cioè un insieme di agenti e servizi; normalmente anche il DF, come l'AMS, è implementato come agente. L'utilità del DF consiste nella possibilità di informare il sistema, l'AMS ed altri agenti – intra ed extra piattaforma – dell'esistenza e delle funzionalità degli iscritti (l'AMS gestisce l'esistenza ma non le funzionalità). In ogni piattaforma deve esistere almeno un agente DF che, tra l'altro, è abilitato al mantenimento di collegamenti ad altri DF situati su altre piattaforme.

Il DF ha le stesse funzioni dell'AMS (*register*, *deregister*, *search*, *modify*): tuttavia, a differenza dell'AMS, il DF consente agli agenti di registrarsi indicando non solo il nome, ma anche i particolari *servizi* esposti. Altri agenti possono usare la funzione di ricerca del DF per trovare tutti gli agenti che offrono un particolare servizio.

3.4.6 Message Transport Service (MTS) e ACC

L'MTS viene descritto dalle specifiche FIPA in modo astratto come il servizio che consente agli agenti di comunicare tra di loro e con AMS e DF. Generalmente è implementato in un componente chiamato ACC (*Agent Communication Channel*) che utilizza le informazioni dell'AMS (riguardanti lo stato, l'identità e la locazione di appartenenza degli agenti) per gestire lo scambio di messaggi tra agenti all'interno o all'esterno della piattaforma.

Gli agenti comunicano tra loro scambiandosi messaggi FIPA-ACL (descritti in §3.4.7): tali messaggi vengono fisicamente trasportati da un agente all'altro (e, se necessario, da una piattaforma all'altra) dall'MTS utilizzando un protocollo di trasporto (MTP, Message Transport Protocol). Affinché gli agenti residenti su due piattaforme diverse possano operare, le due AP devono implementare i protocolli di trasporto definiti da FIPA, che allo stato attuale sono i seguenti:

- HTTP (*HyperText Transfer Protocol*): il noto protocollo applicativo per la trasmissione di testo e contenuti multimediali su cui si basa il World Wide Web.

- WAP (*Wireless Application Protocol*): insieme di protocolli che specificano come i dispositivi senza fili (ad es. telefoni cellulari) possono accedere a Internet.

- IIOP (*Internet Inter-ORB Protocol*): protocollo che consente ad applicazioni distribuite scritte in linguaggi diversi di comunicare attraverso Internet. IIOP fa parte dello standard CORBA (Common Object Request Broker Architecture)[6].

FIPA non impedisce, all'interno di una particolare AP, di utilizzare altri tipi di protocolli di trasporto, ma stabilisce che per la comunicazione inter-piattaforma deve essere utilizzato uno dei tre protocolli sopra elencati.

Si noti inoltre che non necessariamente MTS e ACC coincidono: molti framework (come quelli presi in esame nel Capitolo 4) indicano con MTS il sistema di trasporto interno alla singola piattaforma (che utilizza spesso un protocollo proprietario) e con ACC il componente che gestisce la comunicazione con altre piattaforme. Si noti che, a differenza di AMS e DF, generalmente ACC non è un agente.

3.4.7 Il linguaggio FIPA-ACL

La comunicazione è uno degli aspetti fondamentali della "vita" di un agente. Poiché essa avviene attraverso lo scambio di messaggi, si è resa necessaria la definizione di un linguaggio univoco chiamato ACL (Agent Communication Language). L'ACL di FIPA è un linguaggio basato sulla "Teoria degli Atti Comunicativi" (*Speech Act Theory*), derivata dall'analisi linguistica della comunicazione umana e incentrata sull'idea che tramite il dialogo non solo si esprima un'affermazione ma, contemporaneamente, si compia una vera e propria

[6] Per ulteriori informazioni si veda http://www.omg.org/corba/

azione. Questo modello di rappresentazione risulta particolarmente intuitivo e permette di definire protocolli d'interazione ad alto livello (vedi §3.2.3 e §3.6.3).

L'intento di FIPA con la definizione di ACL è quello di gettare le basi per un linguaggio comune fra gli agenti ed una semantica indipendente dalla struttura interna. Il linguaggio FIPA-ACL e i relativi protocolli di interazione non si occupano dei protocolli di rete e di trasporto a basso livello dell'architettura di comunicazione in un sistema distribuito, ma assumono l'esistenza di tali servizi ed il loro utilizzo per la realizzazione degli "atti comunicativi"

Struttura astratta di FIPA-ACL

Un messaggio FIPA-ACL è costituito da un insieme di uno o più elementi: i principali sono definiti dalle specifiche FIPA e sono illustrati nella tabella seguente, ma gli sviluppatori sono liberi di aggiungere anche elementi user-defined.

performative	Tipo di atto comunicativo del messaggio (vedi tabella successiva).
sender	Mittente del messaggio.
receiver	Destinatario del messaggio. Può contenere il nome di uno o più agenti.
reply-to	Indica che i successivi messaggi della conversazione dovranno essere inviati all'agente indicato in questo elemento.
content	Contenuto del messaggio.
language	Il linguaggio in cui è espresso il contenuto del messaggio (content language).
encoding	La specifica codifica con cui è rappresentato il contenuto del messaggio.
ontology	L'ontologia utilizzata per dare un significato ai simboli che costituiscono il contenuto del messaggio.
protocol	Protocollo di interazione a cui la conversazione corrente deve attenersi.
conversation-id	Espressione che identifica la sequenza di atti comunicativi che insieme formano una conversazione.
reply-with	Espressione che dovrà essere usata dal destinatario per rispondere a questo messaggio.
in-reply-to	In questo campo il destinatario dovrà inserire il contenuto del campo "reply-with" quando risponderà.
reply-by	Istante di tempo entro il quale il mittente desidera riceve una risposta.

Tabella 3-3.I Elementi di un messaggio ACL

Quali di questi elementi siano effettivamente necessari dipende dalla situazione. Molti possono essere omessi quando il loro valore è facilmente ricavabile dal contesto.

L'unico elemento obbligatorio è il *performative* (sebbene la maggior parte dei messaggi conterrà anche gli elementi *sender*, *receiver* e *content*): a questo proposito FIPA incoraggia gli sviluppatori ad utilizzare, quando possibile, i tipi di atti comunicativi standard qui di seguito descritti:

Accept-proposal	Informa il destinatario che una proposta fatta in precedenza (ad es. con un atto Propose) è stata accettata. In pratica significa che il mittente desidera che il destinatario compia l'azione indicata nella proposta.
Agree	Segnala l'accettazione dei fatti indicati nel messaggio (ad es. può seguire un atto Request). Usato ad esempio per indicare che il mittente ha deciso di compiere un'azione che gli era stata richiesta.
Cancel	Annulla un'azione. Il significato può variare a seconda del contesto, ma generalmente indica che il mittente non vuole più che il destinatario compia una certa azione.
Cfp	(Call for proposals) Invita il destinatario a fare una proposta. L'argomento della proposta è indicato nel contenuto del messaggio (generalmente si tratta di un'azione che il destinatario dovrà compiere).
Confirm	Variante di Inform, usato quando il mittente vuole confermare dei fatti su cui il destinatario è incerto.
Disconfirm	Opposto a Confirm, il mittente informa il destinatario che alcuni fatti (che il destinatario crede veri o su cui è incerto) sono falsi.
Failure	Termina una conversazione indicando che si è verificato un errore: il mittente ha tentato di eseguire un'azione (ad es. che gli era stata richiesta dal mittente) ma che l'azione è fallita.
Inform	Informa il destinatario che i fatti indicati nel messaggio sono veri. Da un punto di vista ontologico, il destinatario (se si fida della sincerità del mittente) dovrebbe inserire il contenuto del messaggio nella propria base di conoscenza.
Inform-if	Utilizzato per la creazione di messaggi "macro" (messaggi incapsulati in altri messaggi). Ad esempio, se si trova dentro un messaggio Request, chiede al destinatario se un certo fatto sia vero o falso (stesso significato di Query-if).
Inform-ref	Utilizzato per la creazione di messaggi "macro". Ad esempio, se si trova dentro un messaggio Request, ha lo stesso significato di Query-ref.

Not-understood	"Non capito". Generalmente indica che il mittente non è riuscito a comprendere un messaggio che gli è stato inviato in precedenza dal destinatario.
Propagate	Indica che il destinatario deve ricevere il messaggio ed elaborarlo, inoltre deve inoltrarlo agli agenti indicati in un apposito campo del contenuto.
Propose	Il mittente fa una proposta al destinatario.
Proxy	Come Propagate, con due differenze: 1) il destinatario non deve elaborare il messaggio; 2) il destinatario deve inoltrare non il messaggio stesso, ma un altro messaggio che è incapsulato al suo interno.
Query-if	Chiede al destinatario se un certo fatto è vero o no.
Query-ref	Chiede al destinatario quale oggetto corrisponda all'identificatore contenuto nel messaggio.
Refuse	Il mittente si rifiuta di compiere una certa azione. Può ad esempio seguire un Cfp o un Request.
Reject-proposal	Informa il destinatario che una precedente proposta non è stata accettata.
Request	Chiede al destinatario di compiere un'azione.
Request-when	Chiede al destinatario di compiere un'azione, ma solo quando una certa condizione sarà vera.
Request-whenever	Chiede al destinatario di compiere un'azione tutte le volte che una certa condizione diventa vera.
Subscribe	Versione "persistente" di Query-ref: il mittente chiede al destinatario quale oggetto corrisponda all'identificatore contenuto nel messaggio, inoltre vuole ricevere una notifica tutte le volte che il valore di tale oggetto cambia.

Tabella 3-3.II Atti comunicativi FIPA

In genere si preferisce dare ai vari atti il significato definito da FIPA e riportato nella tabella (formalizzato in [FIPA37]). Tuttavia, in altri contesti questi atti comunicativi possono anche assumere significati diversi. Ad esempio in Zeus (uno dei framework descritti nel prossimo capitolo) Confirm è utilizzato per confermare la prenotazione di un risorsa, mentre Disconfirm annulla una conferma precedente.

Ovviamente lo sviluppatore è libero di definire nuovi tipi di atti comunicativi, quando il problema lo richiede.

Rappresentazione dei messaggi FIPA-ACL

Finora un messaggio è stato definito in modo astratto, cioè come una sequenza di elementi indipendente dall'implementazione: questo perché un messaggio può essere fisicamente rappresentato in modi diversi. FIPA include le specifiche di tre formati: stringa, XML e formato binario (bit-efficient).

Il formato di gran lunga più utilizzato è quello di stringa, illustrato nell'esempio seguente:

```
(request
        :sender (agent-identifier :name Agent1)
        :receiver (set (agent-identifier :name ams))
        :content
        :language
        :ontology FIPA-Agent-Management
        :protocol FIPA-Request
        :conversation-id 12345
)
```

La sintassi degli elementi, separati tra loro da parentesi e "due punti", è chiamata Semantic Language (SL). Pur assomigliando apparentemente al KQML, SL è in realtà un linguaggio molto complesso in grado di esprimere proposizioni in una logica di "attitudini mentali" e "azioni", formalizzate come predicati del primo ordine. Per questo motivo SL viene spesso utilizzato anche per rappresentare il contenuto del messaggio, oltre che il messaggio stesso.

Il primo elemento che segue la prima parentesi aperta è il performative, in questo caso "request", mentre tutti gli altri campi sono indicati dal loro nome preceduto dal simbolo ":" (alcuni per semplicità sono stati omessi).

Nell'esempio, il campo content è stato volutamente lasciato in bianco: un messaggio ACL può incapsulare infatti qualsiasi tipo di contenuto, espresso in un qualsiasi linguaggio ("content language", indicato nell'elemento language).

A questo proposito, il dibattito all'interno di FIPA è ancora aperto: la maggior parte dei membri della fondazione ritiene che l'eventuale adozione di un content language "obbligatorio" possa costituire una limitazione per gli utenti; altri tuttavia sostengono che i contenuti dovrebbero almeno seguire uno "schema logico" comune e ben definito.

Al momento, quindi, FIPA non definisce un content language standard, ma si limita a consigliarne alcuni, di cui include le specifiche, lasciando però agli sviluppatori la libertà di utilizzare anche altri linguaggi, se ritenuti più appropriati (ad es. XML, Prolog).

I linguaggi specificati da FIPA sono:

- SL (Semantic Language, con alcune varianti semplificate chiamate SL0, SL1 e SL2)
- RDF (Resource Description Framework, basato su XML)
- KIF (Knowledge Interchange Format)
- CCL (Constraint Choice Language, linguaggio per la definizione di vincoli).

3.5 Agenti e oggetti

Lo sviluppo del software non è soltanto un problema di tecnologia, ma anche di metodologia. L'avvento di una nuova tecnologia come quella degli agenti comporta sicuramente molti rischi, e l'unico modo per ridurli è costruire su solide

basi preesistenti. Dobbiamo quindi presentare la tecnologia ad agenti come un'estensione di un paradigma già noto e ben radicato, ovvero lo sviluppo di software orientato agli oggetti.

3.5.1 Caratteristiche della programmazione ad oggetti

Il concetto di programmazione orientata agli oggetti (OOP, object-oriented programming) è nato negli anni '80 e ha influenzato notevolmente lo sviluppo dell'informatica negli ultimi due decenni.

Rispetto alla metodologia di programmazione tradizionale (nota come procedurale) nella quale le procedure e le strutture dati coesistono all'interno di un programma come entità separate e blandamente interconnesse, nel paradigma object-oriented gli elementi costitutivi il sistema sono invece gli oggetti, entità computazionali che incapsulano uno stato e sono in grado d'eseguire procedure (o *metodi*) che ne definiscono il comportamento [Adiga93]. Gli oggetti di un sistema interagiscono tra loro attraverso l'invocazione di metodi. Esistono all'interno dell'approccio OO alcuni concetti chiave che lo qualificano e differenziano rispetto alle tecniche di programmazione tradizionale.

- *Classi* - Una classe definisce le proprietà e i comportamenti di oggetti simili specificando i tipi di dati e i metodi che essi condividono. La definizione di una classe consente di rappresentare una struttura comune a più oggetti, essendo un modulo software che offre la possibilità di astrarre le caratteristiche condivise dagli elementi componenti dei sistema.

- *Ereditarietà* - La definizione di una classe consente anche di descrivere proprietà e comportamenti comuni richiesti da un gruppo di entità che poi verrà istanziato in più sottogruppi; ciascun sottogruppo sarà caratterizzato dalle stesse funzionalità della classe genitrice (vale a dire stessi attributi e metodi) oltre che da proprietà specifiche. L'ereditarietà consente perciò la costruzione di nuove classi con l'estensione, riduzione, modifica delle funzionalità di classi già esistenti. È quindi evidente il vantaggio derivante

dal riutilizzo di software già scritto, dato che il progettista si deve preoccupare solo della specializzazione occorrente in ogni nuova sottoclasse. In alcuni linguaggi a oggetti (come ad esempio il C++) è consentita inoltre una ereditarietà multipla, per cui una sottoclasse può ereditare attributi e metodi da più classi genitrici; in altri linguaggi (quali Java e C#) ciò non è consentito.

- *Incapsulamento* - Per incapsulamento si intende la possibilità di confinare tutti i dati pertinenti una entità all'interno dell'oggetto e di conseguenza di definire delle restrizioni agli accessi a tali dati. In questo modo si ha una separazione tra informazioni di rilevanza interna, non accessibili dall'esterno, e dati utilizzati durante l'interazione con altri oggetti. L'interfaccia tra l'oggetto e il mondo esterno viene garantita dai metodi, che svolgono quindi la funzione di risponditori dei messaggi provenienti da altri oggetti.

- *Polimorfismo* - In un'applicazione a oggetti il polimorfismo si realizza quando oggetti differenti rispondono ad uno stesso messaggio in un modo diverso, in ragione dei differente metodo che viene attivato. La progettazione di oggetti *plug-compatible* (ovvero intercambiabili) si giova delle proprietà di polimorfismo: quando un nuovo oggetto viene inserito non è necessario apportare sostanziali modifiche al programma, purché i metodi inseriti all'interno dei nuovo oggetto siano attivabili dagli stessi messaggi definiti in precedenza per gli altri oggetti.

Nella programmazione orientata agli oggetti una classe consiste in un insieme di attributi (chiamati anche campi o variabili d'istanza) e di metodi. I metodi sono operazioni (funzioni o procedure) che manipolano gli attributi e possono interagire con altri oggetti. Gli attributi possono essere normali variabili o riferimenti ad altri oggetti.

Una classe descrive un insieme di oggetti concreti, chiamati istanze della classe, tutti dotati della stessa struttura (gli stessi attributi) e lo stesso comportamento (gli stessi metodi). In genere esiste un metodo standard (chiamato

"new" in quasi tutti i linguaggi di programmazione) per creare nuovi oggetti a partire dalle classi.

La definizione di una classe consiste nella dichiarazione degli attributi e nell'implementazione dei metodi. Spesso è divisa in due parti: l'*interfaccia* descrive quali metodi caratterizzano la classe, ciascuno con le sue funzionalità, mentre l'implementazione (che di solito è nascosta agli utenti della classe) specifica come tali operazioni sono realizzate. I metodi sono caratterizzati da diritti di accesso, che indicano quali operazioni sono visibili all'utente e quali no.

Alcuni linguaggi di programmazione consentono anche la definizione di *variabili di classe*, cioè condivise da tutti gli oggetti di una classe (al contrario delle variabili d'istanza, i cui valori sono diversi da oggetto a oggetto), e di analoghi *metodi di classe*. Variabili e metodi di classe sono spesso usati al posto delle variabili e procedure globali, non ammesse in OOP.

3.5.2 Differenze tra agenti e oggetti

Un agente può essere definito in modo molto naturale come un oggetto, o meglio come una classe che incapsula le funzionalità implementate dall'agente, i cui metodi e proprietà pubblici definiscono i servizi che esso mette a disposizione degli altri agenti, servizi realizzati tramite metodi e proprietà privati o protetti. Si può ovviamente usare un qualunque linguaggio a oggetti, ma la natura inerentemente distribuita di un'applicazione multi-agente e l'eterogeneità a livello hardware e di sistema operativo che si può incontrare in una rete di computer fanno preferire il linguaggio che per definizione è nato per programmare in ambienti con tali caratteristiche: Java[7].

Anche se un agente spesso è implementato come un oggetto (o più oggetti) esso è molto più di un insieme di oggetti. Lo schema seguente riassume le principali differenze tra gli agenti e gli oggetti:

---

[7] In §4.2.4 vengono descritte le principali caratteristiche di Java e le motivazioni che lo rendono uno tra i più diffusi linguaggi per la realizzazione di sistemi ad agenti.

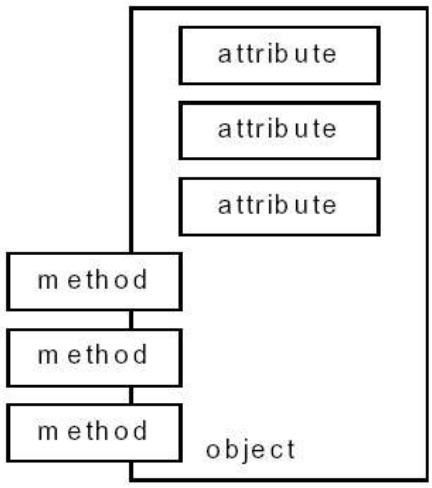
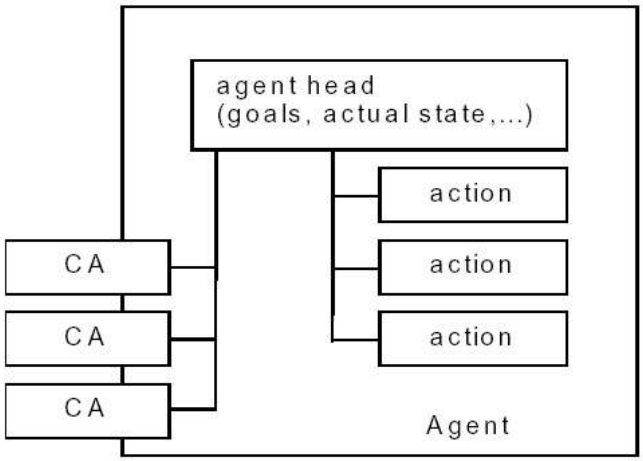
Oggetti: 1. Passivi 2. Completamente controllati 3. Comunicaz. sincrona	**Agenti**: • Attivi • Autonomi • Comunicaz. asincrona
	

Tabella 3-3.III Agenti e oggetti

Per prima cosa, rispetto agli oggetti, gli agenti sono *attivi* in quanto possono prendere iniziative e hanno il controllo su come e quando elaborare le richieste esterne. In secondo luogo, gli agenti non agiscono isolati, ma in cooperazione o coordinazione con altri agenti.

Spesso si parla di scambio di messaggi sia nel contesto degli oggetti che in quello degli agenti, ma i due concetti sono diversi:

- *per gli oggetti*, scambio di messaggi significa invocazione di metodi, ossia chiamate di procedure (generalmente sincrone).

- *per gli agenti*, scambio di messaggi significa invio e ricezione di atti comunicativi (CA, generalmente asincroni).

Gli agenti, se implementati come oggetti, devono scambiarsi messaggi facendo riferimento ad un nome, o comunque ad un identificativo (ad esempio l'AID definito da FIPA), ma mai conservare riferimenti espliciti ad altri agenti né invocare direttamente i metodi.

3.6 Agent UML

3.6.1 La programmazione object-oriented e l'UML

Poiché, come si è già detto in §3.5, il concetto di agente può essere "visto" come un'estensione del concetto di oggetto, anche le metodologie di sviluppo dei sistemi ad agenti possono essere costruite sulla base di quelle ad oggetti.

Nell'evoluzione delle tecniche object-oriented, un ruolo fondamentale è stato svolto dall'**UML**, *Unified Modeling Language*. Tale linguaggio, creato inizialmente da Grady Booch, James Rumbaugh e Ivar Jacobson (noti anche, nel mondo informatico, come "I Tre Amigos"), è stato adottato come standard dall'OMG (Object Management Group) nel 1997 ed è in continua evoluzione (la versione corrente dell'UML è la 1.4).

É possibile trovare una trattazione completa del linguaggio in [Booch99], mentre una buona introduzione all'UML si trova in [Schmul99].

L'UML è nato per unificare e formalizzare i numerosi approcci esistenti al ciclo di vita del software object-oriented, e viene utilizzato in tutte le fasi dello sviluppo, dall'analisi dei requisiti al rilascio del sistema. Ad un livello molto concreto, l'UML può essere semplicemente visto come un insieme di diversi diagrammi, divisi in quattro gruppi o modelli (vedi figura 3-6 a pag. 69):

- **Modello della struttura statica**:
- **Diagrammi delle classi** (*Class diagrams*): rappresentano un insieme di classi e le relazioni tra di esse. Non si tratta necessariamente delle classi che verranno realizzate nell'implementazione: l'UML può e deve essere usato anche nelle fasi di analisi e progetto, durante le quali le classi possono rappresentare ad esempio concetti o entità del mondo reale.
- **Diagrammi degli oggetti** (*Object diagrams*): come i diagrammi a classi, con la differenza che rappresentano oggetti concreti, istanze delle classi.

- **Diagrammi dei package** (*Package diagrams*): dividono le classi e/o gli oggetti in raggruppamenti (*package*). Un package può contenere classi, altri package, o entrambe le cose. I package formano una struttura ad albero che consente di organizzare meglio un progetto.

- **Modello dinamico**:

- **Diagrammi di sequenza** (*Sequence diagrams*): rappresentano le interazioni tra gli oggetti (passaggio di messaggi) mettendo in evidenza la dimensione temporale.

- **Diagrammi di collaborazione** (*Collaboration diagr*ams): semanticamente equivalenti ai diagrammi di sequenza, mettono tuttavia in risalto la dimensione "spaziale" (ovvero le relazioni tra gli oggetti) rispetto a quella temporale.

Nota: i diagrammi di sequenza e di collaborazione sono chiamati, nel loro complesso, **diagrammi di interazione** (*Interaction diagrams*).

- **Diagrammi di attività** (*Activity diagrams*): simili ai "classici" diagrammi di flusso (e anche, in un certo senso, alle reti di Petri), descrivono la sequenza di operazioni che devono essere compiute per risolvere un determinato problema. Non è detto che queste operazioni siano svolte tutte da uno stesso oggetto: è possibile dividere un diagramma in "corsie" (*swimlanes*) per rappresentare le diverse entità che svolgono le azioni.

- **Diagrammi di stato** (*State diagrams* o *Statecharts*): simili ai "classici" automi a stati finiti, rappresentano i possibili stati (e le transazioni tra essi) di un singolo oggetto. Uno stato può contenere altri stati "annidati".

- **Modello use case**:

- **Diagrammi Use Case**: specificano il modo in cui un sistema interagisce con "attori esterni" (che possono essere gli utenti o altri sistemi).

- **Modello dell'implementazione**:
- **Diagrammi dei componenti** (*Component diagrams*): rappresentano insiemi di componenti software, come programmi, librerie, database, ecc, e le relazioni tra essi. I componenti spesso sono istanze concrete di classi definite nel modello statico.
- **Diagrammi di deployment** (tradotto talvolta come diagrammi di *distribuzione* o di *rilascio*): rappresentano i componenti hardware di un sistema e le relazioni tra essi. Le entità hardware possono contenere i componenti descritti dei diagrammi dei componenti.

L'UML è molto più di un insieme di diagrammi: è un vero e proprio linguaggio, con una grammatica ed una sintassi, che fornisce tra l'altro molti meccanismi di estensione, risultando così molto flessibile e facilmente adattabile alle esigenze degli sviluppatori. L'esempio più comune è quello dei diagrammi ibridi, dove vengono usati simboli presi da diversi diagrammi.

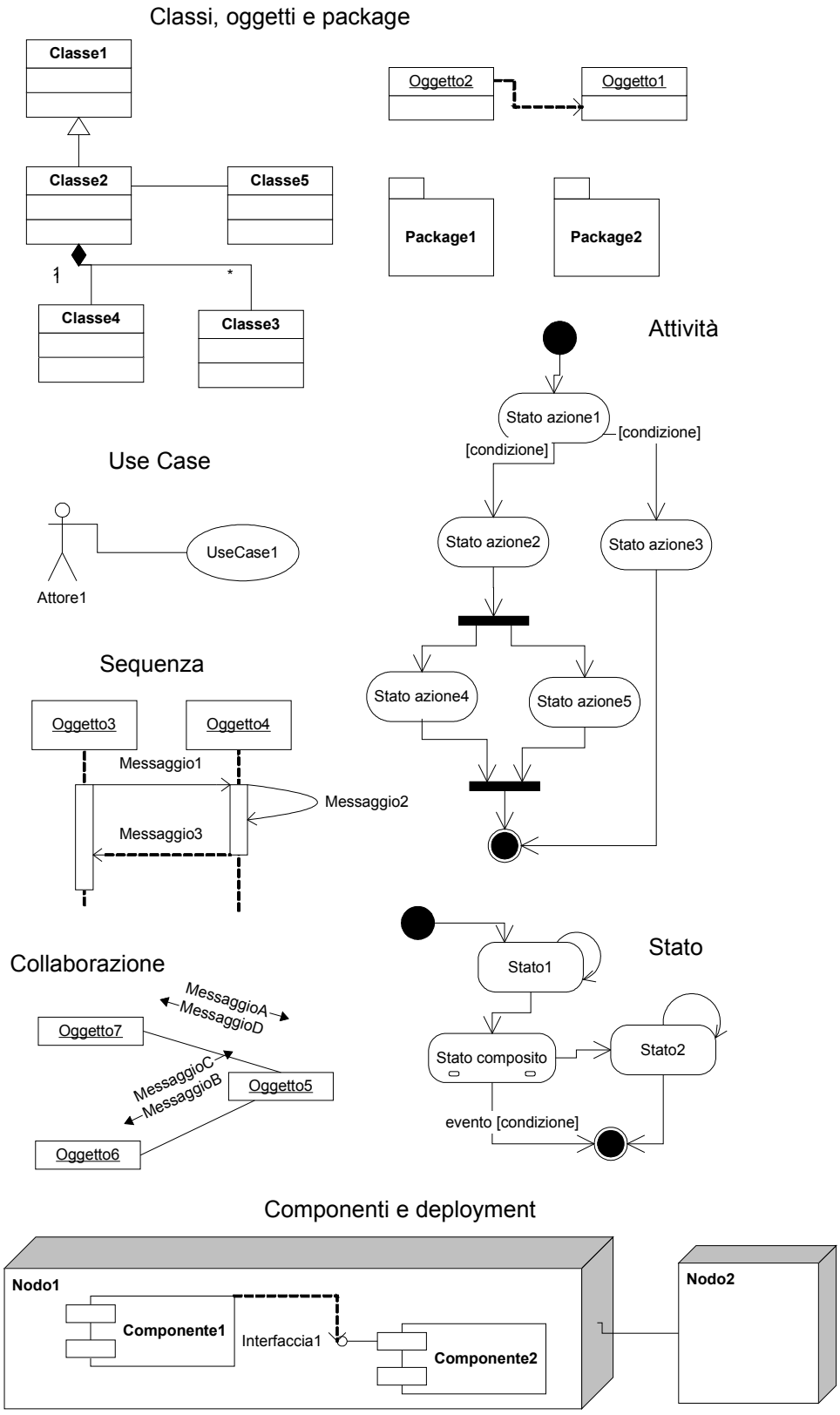


Figura 3-3.6 I principali diagrammi UML

Allo stato attuale, l'UML non è sufficiente per modellare gli agenti e i sistemi multi-agente. Quello che occorre è una metodologia che supporti l'intero processo di sviluppo ad agenti, attraverso le fasi di pianificazione, analisi, progetto, realizzazione del sistema, collaudo e manutenzione; numerosi ricercatori in tutto il mondo stanno lavorando su questi problemi, ed esistono già numerose proposte di metodologie e strumenti per lo sviluppo di sistemi ad agenti. Tra le principali organizzazioni che si occupano di studiare e raccomandare estensioni alle metodologie esistenti troviamo FIPA (Foundation for Intelligent Physical Agents) e Agent Work Group (un sottogruppo di OMG).

OMG prevede di incorporare estensioni "ad agenti" nella prossima versione dell'UML (la 2.0).

3.6.2 I diagrammi AUML

Nella presente tesi verrà presa in esame una delle più note proposte di estensione dell'UML: l'*Agent UML* (AUML).

Attualmente l'AUML non è ancora un vero standard, quanto piuttosto un obiettivo. Le estensioni finora proposte sono il frutto degli sforzi di diversi ricercatori, tra cui James Odell, H.Van Dyke Parunak, Bernhard Bauer, Federico Bergenti, Agostino Poggi, Jörg P.Müller e molti altri. Le principali estensioni dell'AUML riguardano i seguenti argomenti:

- **Protocolli di interazione**: formalizzano le sequenze di messaggi scambiati tra due o più agenti. Generalmente vengono rappresentati con diagrammi di sequenza (opportunamente modificati), ma possono anche essere modellati con diagrammi di attività, di collaborazione o di stato.

- **Classi di agenti**: rappresentate con diagrammi delle classi.

- **Elaborazione interna agli agenti**: rappresentata con diagrammi dinamici, soprattutto attività e stato.

- **Modellamento dei ruoli**: rappresentazione dei molteplici ruoli che un agente può assumere in una conversazione.

L'AUML propone inoltre formalismi per modellare numerose altre caratteristiche tipiche degli agenti, come ad esempio l'eventuale mobilità, l'uso di agenti come interfacce verso sistemi, e relazioni complesse tra agenti.

Nei prossimi paragrafi verranno presi in esame i principali aspetti dell'AUML, evidenziando in particolare le differenze rispetto all'UML. E' necessario premettere che, non essendo ancora AUML uno standard, ne esistono numerose versioni, ognuna con qualche piccola differenza rispetto alle altre. L'AUML qui descritto di basa sui seguenti documenti di lavoro (*working documents*): [Odell00], [Bauer00], [Bauer01].[8]

3.6.3 Protocolli di interazione: i diagrammi di protocollo.

Un protocollo di interazione tra agenti (AIP, Agent Interaction Protocol) descrive un modello di comunicazione, ovvero la sequenza consentita dei messaggi scambiati ed eventuali vincoli sui contenuti di tali messaggi.

L'AUML rappresenta un AIP con un'estensione del diagramma di sequenza: il diagramma di protocollo (Protocol diagram).

L'esempio che segue mostra un esempio del Contract Net, uno dei protocolli di negoziazione più diffusi.

[8] Ulteriori documenti sono disponibili presso il sito www.auml.org.

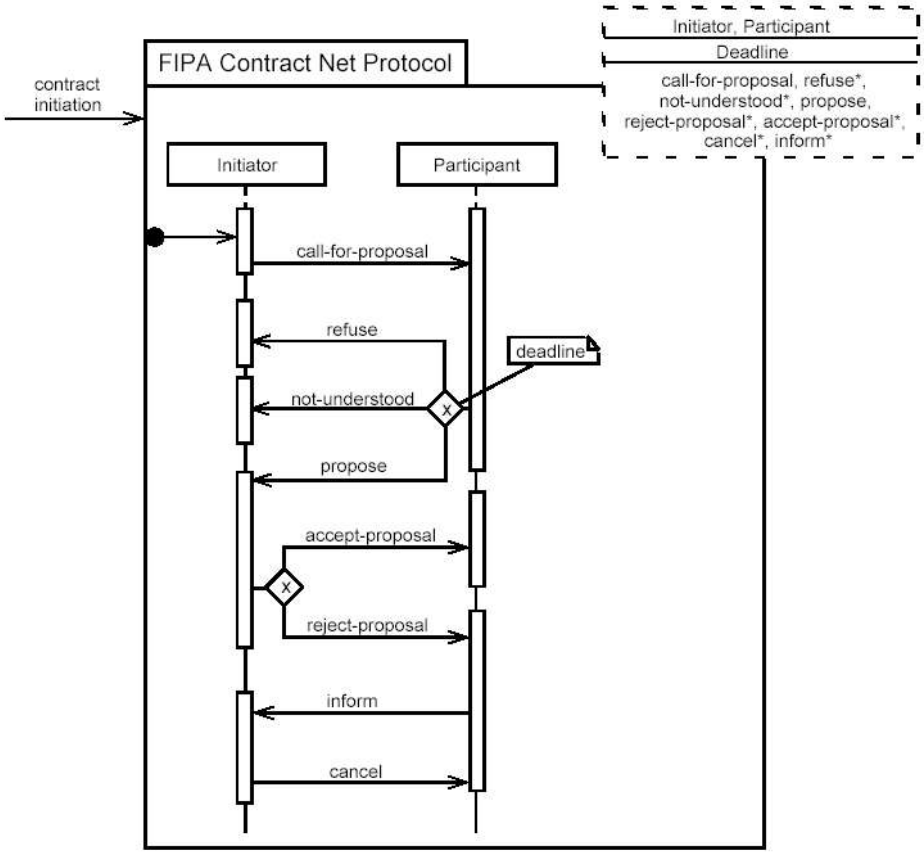


Figura 3-3.7 Protocollo Contract Net secondo [Odell00]

L'intero protocollo è racchiuso in un package: in questo modo il protocollo rappresenta un'entità a sé all'interno del progetto (nell'UML i package raggruppano soltanto classi, mentre in AUML possono raggruppate qualsiasi elemento). Il protocollo, inoltre, è anche un pattern, un modello personalizzabile, i cui parametri sono elencati nel riquadro tratteggiato.

Come nei normali diagrammi di sequenza UML, la dimensione verticale rappresenta il tempo, che scorre dall'alto verso il basso (la linea verticale è chiamata lifeline, linea di vita, mentre le parti spesse sono dette attivazioni). Nella dimensione orizzontale non si trovano oggetti (come in UML), bensì ruoli. In questo esempio i ruoli sono *Initiator* (l'entità che dà inizio alla conversazione) e *Participant* (l'entità che riceve il primo messaggio, chiamata anche *Respondent*): i nomi dei ruoli sono tra i parametri del protocollo. Le frecce a punta sottile indicano l'invio di messaggi asincroni da un'entità all'altra (è possibile

aggiungere dei numeri a destra e a sinistra delle frecce per indicare la molteplicità).

Rispetto ai diagrammi classici, sono stati aggiunti dei simboli per indicare delle "decisioni" circa i messaggi da inviare. Il significato dei simboli è il seguente:

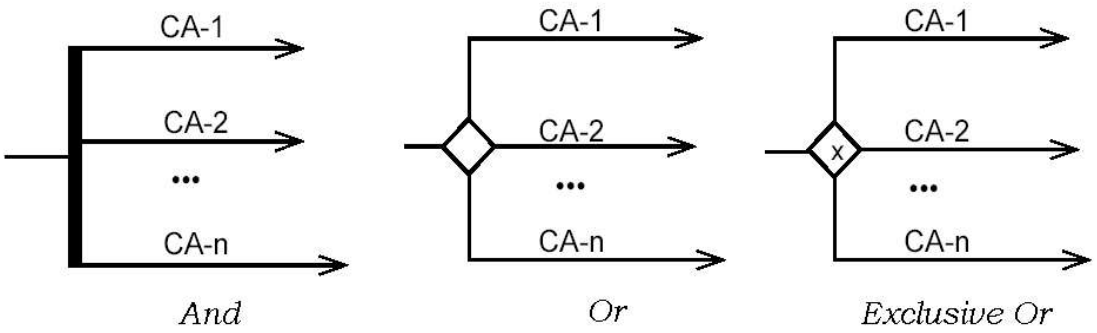


Figura 3-3.8 Relazioni tra atti comunicativi

I messaggi sono anche chiamati "atti comunicativi" (CA, Communication Act). Nell'esempio del Contract Net, l'Initiator inizia la conversazione con un atto "call-for-proposal", e il Participant può scegliere di rispondere con "refuse", "propose" o "not-understood" (la "x" indica l'or esclusivo tra i messaggi). La nota "deadline" indica il tempo limite oltre il quale un'eventuale proposta da parte di Participant verrà automaticamente rifiutata dall'Initiator.

Il diagramma di protocollo può esprimere anche la concorrenza dell'elaborazione: ad esempio, i due frammenti di diagramma che seguono sono equivalenti (in figura viene mostrato il caso "exclusive or", ma esistono rappresentazioni analoghe nei casi "and" e "or").

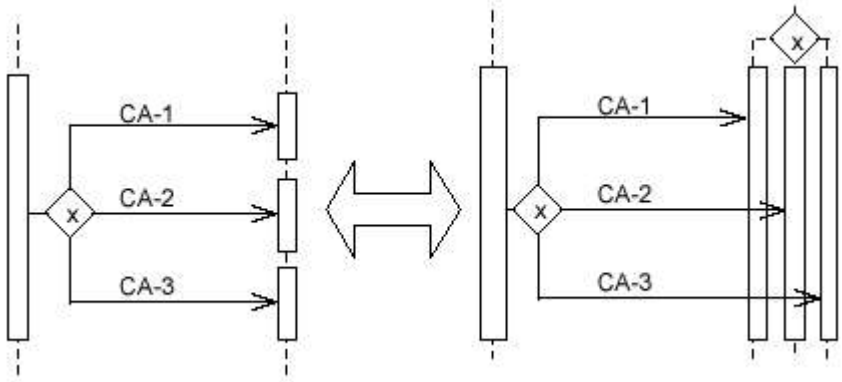


Figura 3-3.9 Thread concorrenti

La notazione di sinistra è quella utilizzata nell'esempio del Contract Net. La notazione di destra "spezza" la linea di vita in più linee (il significato di tali linee, tuttavia, varia a seconda del contesto: può indicare che i messaggi sono raccolti da thread differenti all'interno dell'agente, oppure che l'agente assume un ruolo diverso a seconda di quale messaggio elabora).

Come si è detto, un protocollo è un modello (template), i cui parametri generici sono chiamati *unbound parameters*. Volendo applicare il pattern ad un caso concreto, occorre sostituirli con i loro valori effettivi (*bound parameters*).

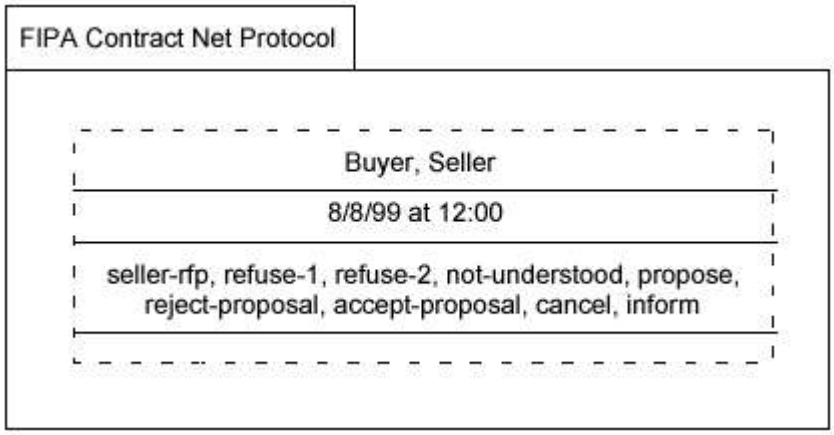


Figura 3-3.10 Istanza di un template di protocollo

Questo modo di utilizzare i modelli fa parte dell'AUML, non dell'UML. In questo esempio, i nomi dei ruoli sono stati sostituiti dai nomi effettivi degli agenti,

la deadline generica è diventata un istante di tempo; addirittura sono stati introdotti due tipi diversi di messaggio “refuse”.

I diagrammi di protocollo saranno probabilmente parte integrante della versione 2.0 dell’UML. FIPA, già dal 1999, ha adottato tali diagrammi per rappresentare tutti i protocolli descritti nelle proprie specifiche [FIPA25], sebbene in una versione lievemente diversa, come si può vedere dalla figura seguente.

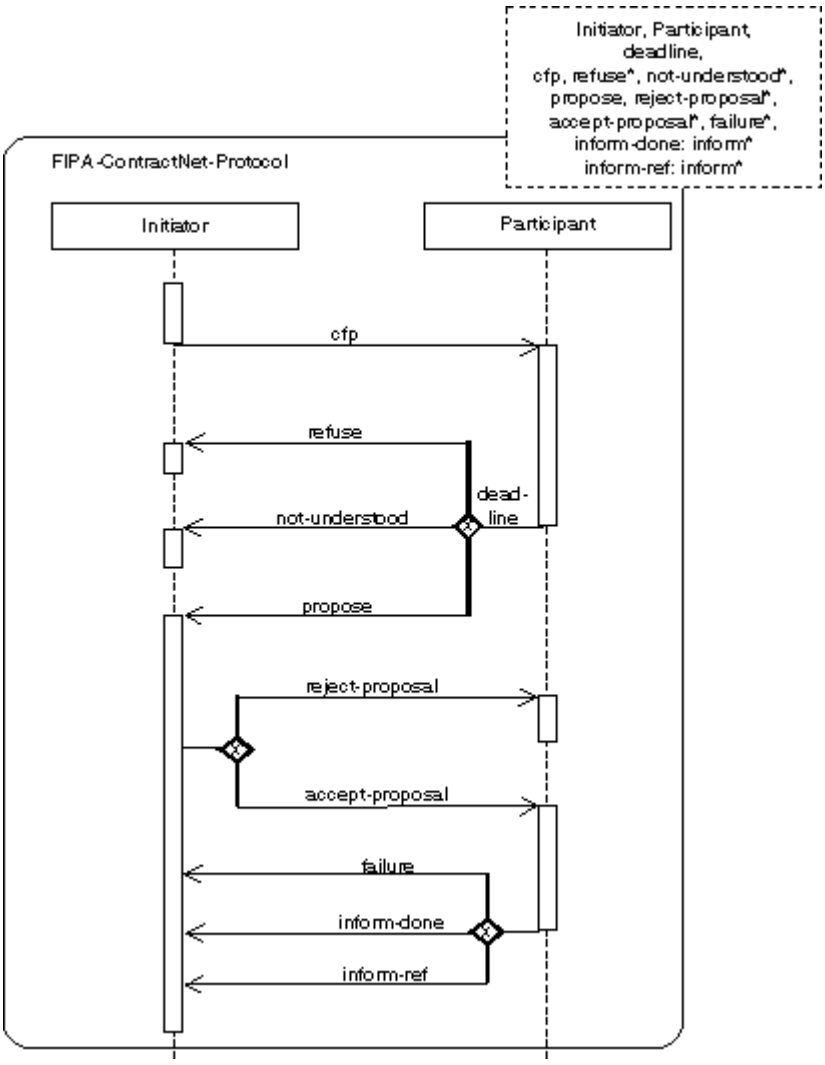


Figura 3-3.11 Formulazione alternativa del Contract Net

Il protocollo non è racchiuso in un package ma in un rettangolo con gli angoli arrotondati. La deadline non è una nota ma una scritta accanto al punto in cui viene inviato un messaggio.

3.6.4 Altre rappresentazioni delle interazioni tra agenti

I protocolli e, più in generale, le interazioni tra due o più agenti possono essere visualizzati anche con altri tipi di diagrammi. Non esiste un diagramma migliore in assoluto: dipende sia dalla persona che lo traccia, sia dal protocollo specifico. Inoltre, creare più diagrammi per rappresentare la stessa cosa aiuta a considerare il problema da tutti i punti di vista.

I diagrammi di collaborazione, ad esempio, sono equivalenti ai diagrammi di sequenza, ma gli oggetti (o i ruoli, o gli agenti) possono essere posizionati in qualunque punto, mentre nei diagrammi di sequenza devono trovarsi tutti nella prima riga in alto. La dimensione temporale non è esplicita, ma indicata da numeri progressivi associati ai messaggi scambiati. Un esempio è riportato più avanti (§3.6.7).

Per altri tipi di protocolli possono risultare più utili i diagrammi di attività. Essi forniscono una rappresentazione grafica che rende possibile una visualizzazione molto semplice dei processi e degli eventi (trigger) che li attivano, soprattutto per quanto riguarda le elaborazioni concorrenti asincrone. Inoltre sono molto indicati per esprimere comunicazioni simultanee tra più agenti.

Nell'esempio che segue viene utilizzato un diagramma di attività per descrivere un'interazione complessa tra quattro agenti (facenti parte di un ipotetico sistema di commercio elettronico), rappresentati da altrettante corsie. Si noti come alcune attività siano svolte concorrentemente, tranne in un punto dove la sequenza è sincronizzata dall'agente ECN (che qui sta per Electronic Commerce Network). La notazione è quella classica dell'UML.

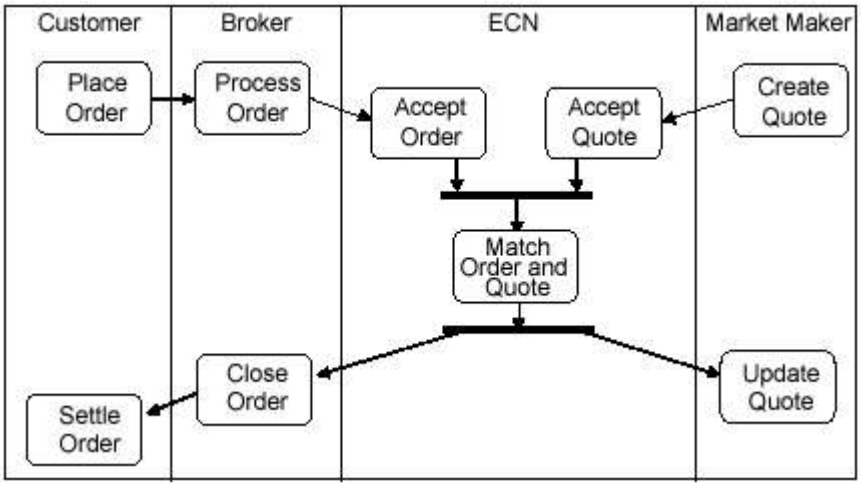


Figura 3-3.12 Protocollo rappresentato come Activity Diagram

Un altro modo di rappresentare i protocolli è con diagrammi di stato. Essi sono usati raramente, in quanto in genere si preferisce avere una visione "orientata agli agenti" (diagrammi di interazione), oppure "orientata ai processi", (diagrammi di attività). Il terzo tipo di visione è quella "orientata allo stato":

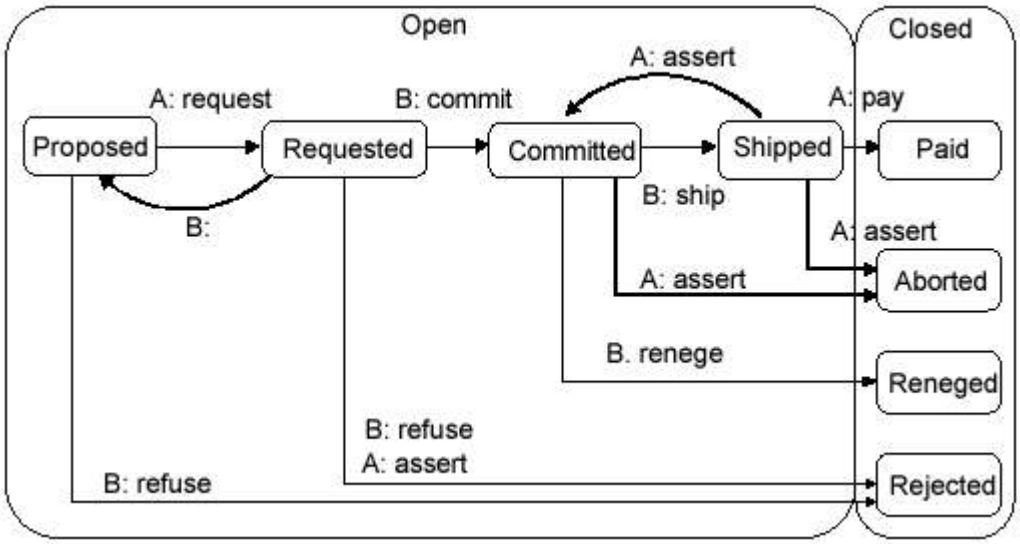


Figura 3-3.13 Protocollo rappresentato come Statechart

In figura 3-13, il protocollo attraversa due stati principali, ognuno dei quali contiene al proprio interno altri stati annidati; le transizioni tra gli stati sono del tipo "*Agente:messaggio*".

Gli stati del diagramma hanno un significato concettuale, e generalmente non corrispondono ad alcun oggetto implementato: ad esempio, esistono gli agenti A e B, ma non esiste un oggetto "Requested" in quanto si tratta semplicemente di uno stato del protocollo (questi diagrammi risulteranno particolarmente utili per implementare protocolli con FIPA-OS, vedi §5.6.2).

Questo modo di vedere il problema ha un punto di forza molto importante: fornisce dei vincoli ben precisi al protocollo. Ad esempio, osservando la figura, ci si rende subito conto che l'agente B può inviare il messaggio "commit" all'agente A soltanto se il protocollo è nello stato "Requested".

3.6.5 Classi di agenti

Nella programmazione orientata agli oggetti, una classe è un modello per creare oggetti concreti, chiamati istanze della classe: analogamente, in un paradigma ad agenti, possiamo definire una "classe di agenti" ("**agent class**") come un modello per creare agenti concreti.

Quando si progetta un sistema ad agenti occorre dunque una notazione per distinguere, in un diagramma delle classi, quali tra esse rappresentano gli agenti e quali sono invece classi ordinarie.

L'AUML propone un nuovo simbolo per indicare un'agent class, mostrato nella figura a sinistra. Alternativamente, è possibile utilizzare la classica notazione UML (figura a destra) con l'aggiunta di uno stereotipo "agent".[9]

[9] In UML uno stereotipo rappresenta una "versione specializzata" di un simbolo già esistente (sia esso una classe, una relazione, ecc) ed è indicato da una scritta tra virgolette.

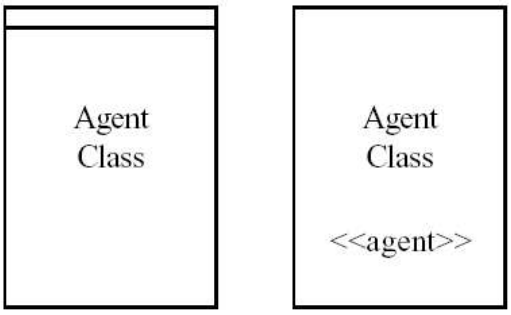


Figura 3-3.14 Diverse rappresentazioni di una classe di agenti

Un'altra proposta dell'AUML è quella in [Figura 3-15], che permette di specificare informazioni più dettagliate circa un agente. Alternativamente, è possibile utilizzare la classica notazione UML in [Figura 3-16].

Nella prima riga si trova prima di tutto il nome della classe di agenti, seguita, dopo la barra, da un elenco di **ruoli** che l'agente può ricoprire (l'argomento ruoli sarà approfondito nel prossimo paragrafo). Il nome può anche essere sottolineato per indicare che si sta descrivendo un agente concreto (individual agent, ovvero un'istanza) e non una classe.

La **descrizione dello stato** contiene un normale elenco di campi, come in UML: tuttavia viene definito un nuovo tipo di dato, chiamato "wff" ("well formed formula", formula ben formata), utilizzato per rappresentare qualsiasi tipo di descrizione logica dello stato, indipendentemente dalla particolare implementazione. Ad esempio, se si modella un agente con la tecnica BDI, è possibile definire quattro variabili: beliefs, desires, intentions e goals (questi ultimi, eventualmente, suddivisi in permanent-goals e actual-goals), tutte di tipo "wff". Alcuni attributi possono essere persistenti, in tal caso ad ognuno di essi è aggiunto lo stereotipo "persistent".

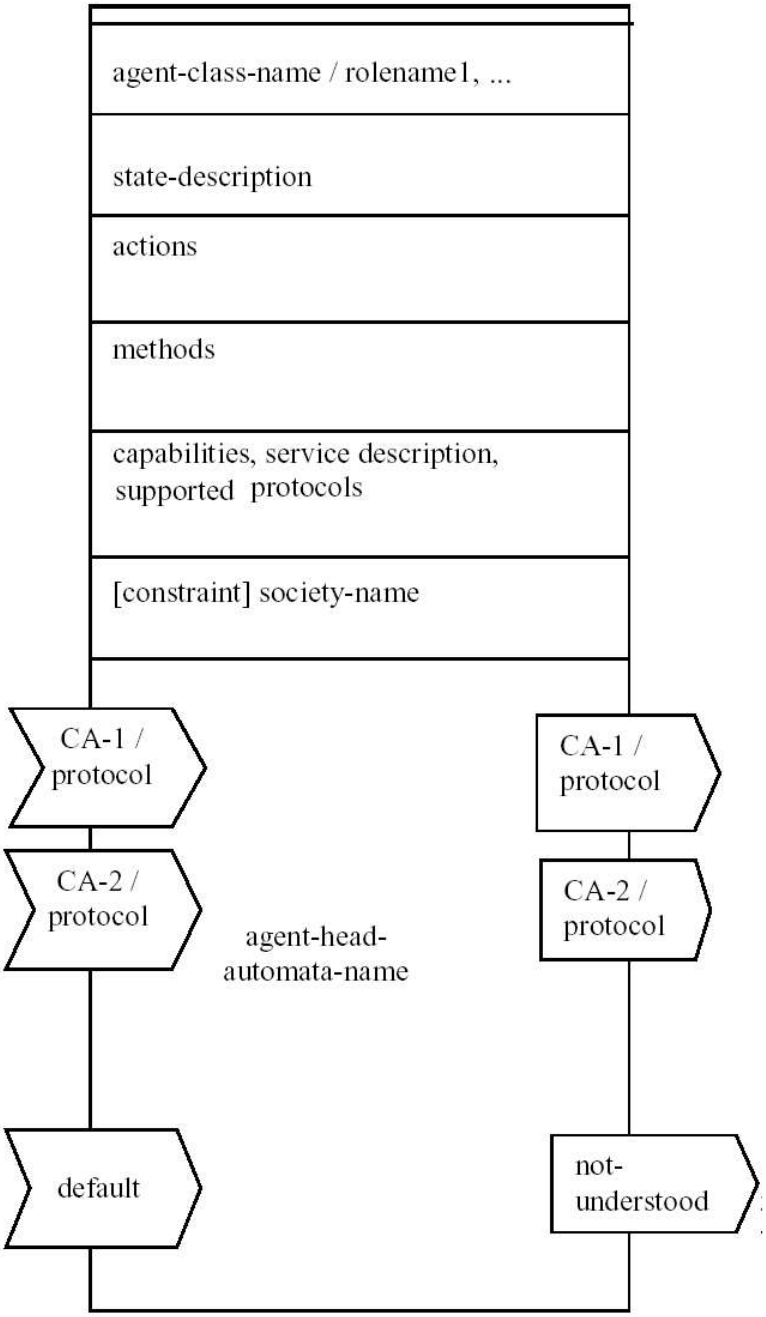


Figura 3-3.15

Ovviamente lo stato può anche essere definito da normali campi UML, nel caso ad esempio in cui si voglia implementare un agente con uno dei framework object-oriented basati su Java di cui si parlerà nel prossimo capitolo.

Ad ogni modo, nei diversi stadi della progettazione potrebbero essere più adeguati diversi tipi di modelli: ad esempio, un agente può essere definito con una

logica BDI a livello concettuale, e poi essere implementato su una piattaforma object-oriented.

Le **azioni** si differenziano dai metodi in quanto modellano i comportamenti reattivi e proattivi dell'agente, rispettivamente rappresentati dagli stereotipi "re-active" e "pro-active" (non mostrati nella figura); le azioni sono visibili agli altri agenti, mentre i metodi sono utilizzabili solo dall'agente stesso. Per il resto, azioni e metodi sono specificati in modo identico, secondo le consuete regole dell'UML.

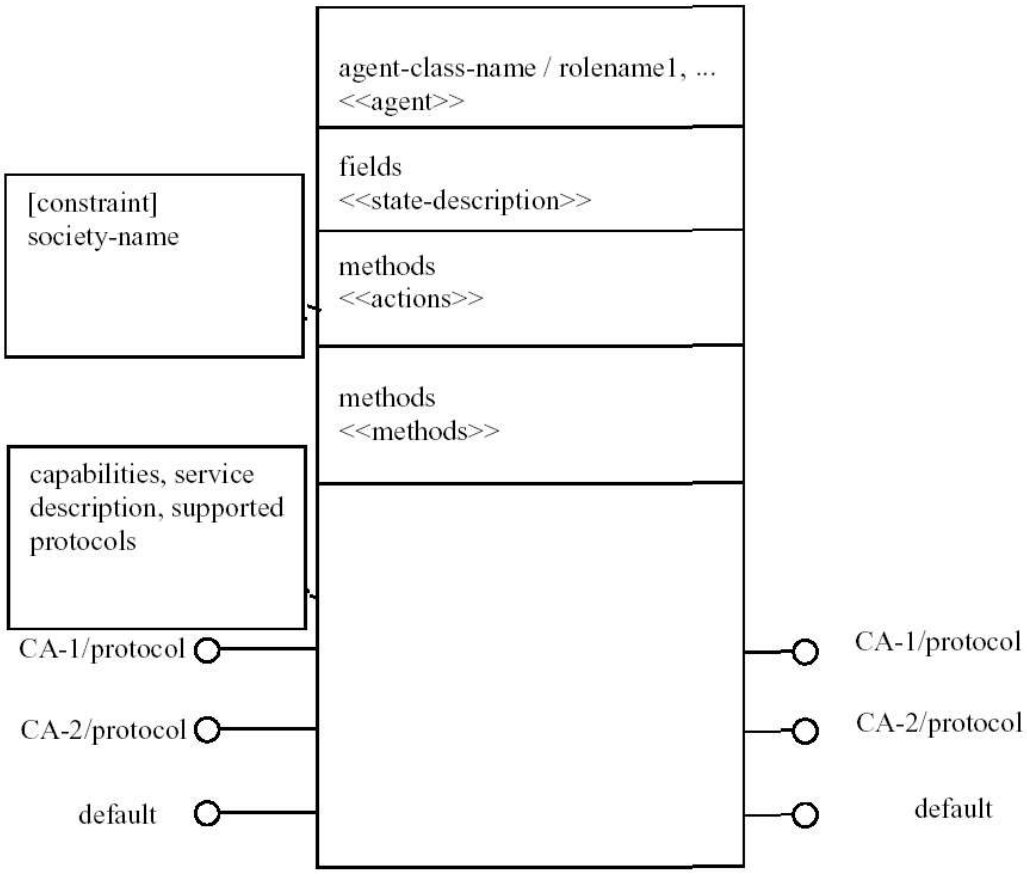


Figura 3-3.16

L'interfaccia di un agente verso il mondo esterno è costituita dall'invio e ricezione di atti comunicativi, rappresentati con i due simboli a forma di frecce spesse presi in prestito dai diagrammi di attività (oppure, in figura 3-16, con i cerchi che in UML rappresentano il concetto di "interfaccia"). La notazione "atto/protocollo" (ad esempio "CA-1/protocol" nella figura) indica che un

messaggio contenente un certo tipo di atto comunicativo è ricevuto nell'ambito di un certo protocollo ("default" indica un atto comunicativo qualsiasi). Al posto di "atto/protocollo" si può scrivere "protocollo[atto]".

Infine, l' "agent-head-automata" rappresenta il comportamento dell'agente, il suo "motore". La parola "automata" può far pensare ad una macchina a stati, ma in realtà il cuore di un agente può essere descritto con tutti i tipi di diagrammi dinamici che l'UML mette a disposizione: sequenza, collaborazione, attività e stato. In particolare, nel prossimo paragrafo saranno trattate alcune estensioni proposte per gli ultimi due tipi di diagrammi.

3.6.6 Elaborazione interna agli agenti

Nel descrivere i processi interni di un agente, spesso è necessario rappresentare attività svolte da altri agenti: l'AUML introduce nuovi simboli per rendere possibile tutto ciò. Nei diagrammi di attività, i rettangoli con interruzioni agli angoli indicano attività esterne.

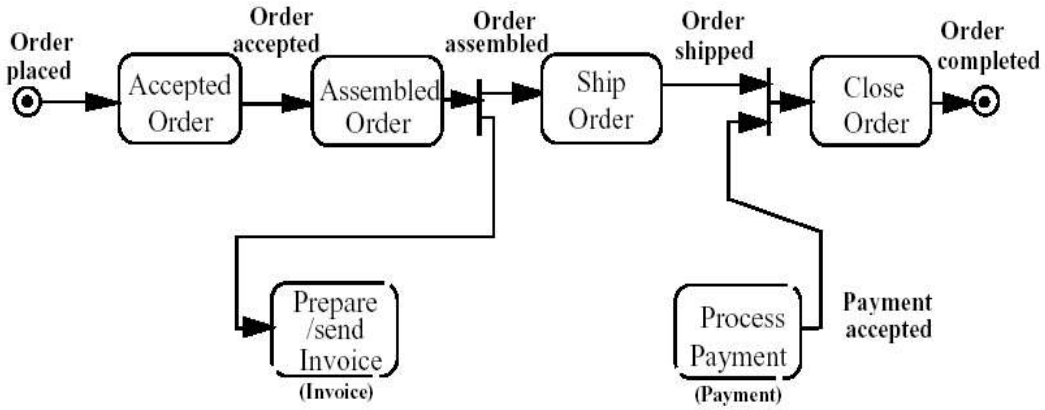


Figura 3-3.17 Activity Diagram esteso

Invece nei diagrammi di stato le attività esterne sono modellate con frecce tratteggiate.

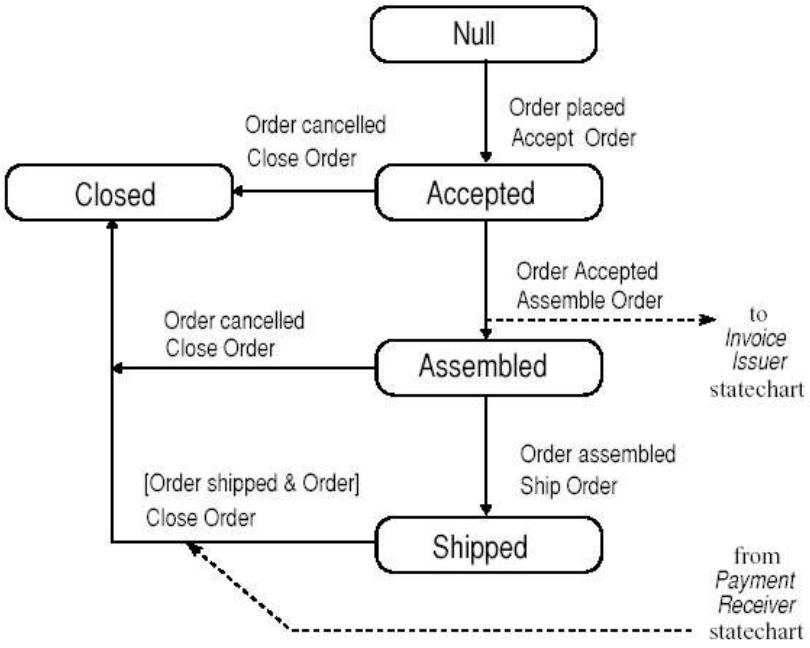


Figura 3-3.18 Statechart esteso

Inoltre è possibile aggiungere i simboli di ricezione/invio messaggio nei diagrammi di stato, per rappresentare il modo in cui l' "agent head automata" reagisce ad un determinato tipo di atto comunicativo:

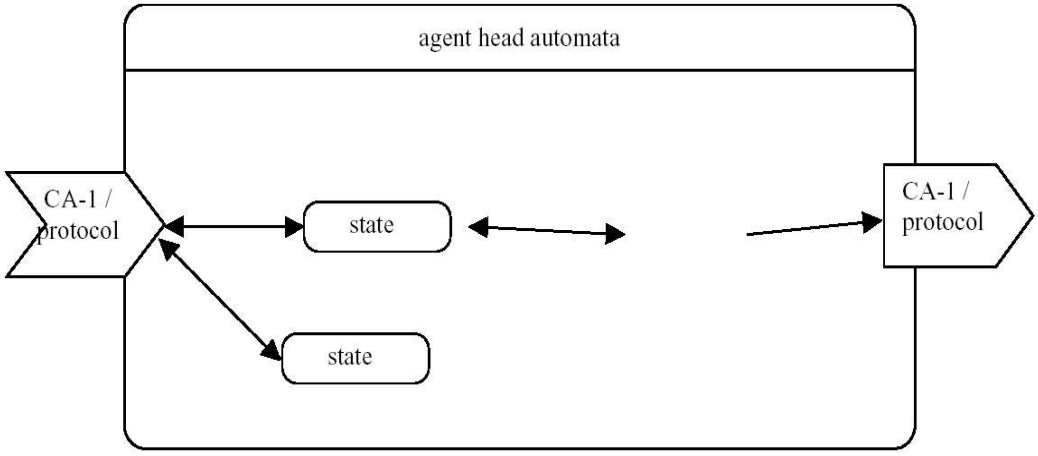


Figura 3-3.19 Altro tipo di Statechart esteso

3.6.7 Modellamento dei ruoli

Nell'ambito dei sistemi multi-agente, un ruolo ("agent-role") è un insieme di agenti che soddisfano certe proprietà, in termini ad esempio di interfacce, servizi offerti o comportamento. Il concetto di ruolo è già presente nell'UML, che prevede anche la possibilità di classificazione multipla (ad esempio, in un sistema di commercio elettronico, un agente può agire contemporaneamente da venditore e da acquirente) e di classificazione dinamica (un agente cambia ruolo nel tempo). L'implementazione di un agente può dunque soddisfare numerosi ruoli, e un sistema multi-agente può essere definito, oltre che in termini di classi di agenti, anche in termini di ruoli.

Per indicare che una classe di agenti supporta alcuni ruoli, si usa la notazione già vista nel paragrafo precedente:

nome-classe-agente / ruolo1, ruolo2, …

Inoltre, per indicare che un certo numero di istanze (agenti concreti) soddisfano certi ruoli e appartengono ad una classe, si usa la notazione:

istanza1, istanza2, … / ruolo1, ruolo2, … : nome-classe-agente

Indicare i ruoli in un diagramma delle classi è semplice; i problemi sorgono quando si vogliono utilizzare diagrammi dinamici per rappresentare interazioni tra agenti in cui intervengono cambiamenti di ruolo.

Le due figure seguenti mostrano come sia possibile modellare i ruoli con i classici diagrammi di sequenza UML: tuttavia, gli schemi che ne derivano risultano spesso confusi e di difficile lettura.

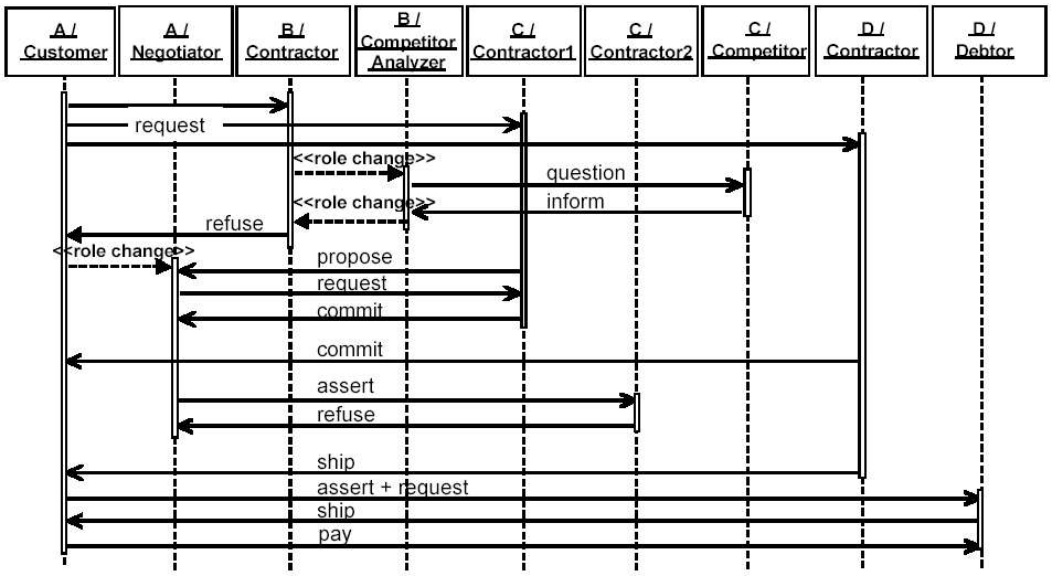


Figura 3-3.20 Modellamento dei ruoli conforme all'UML

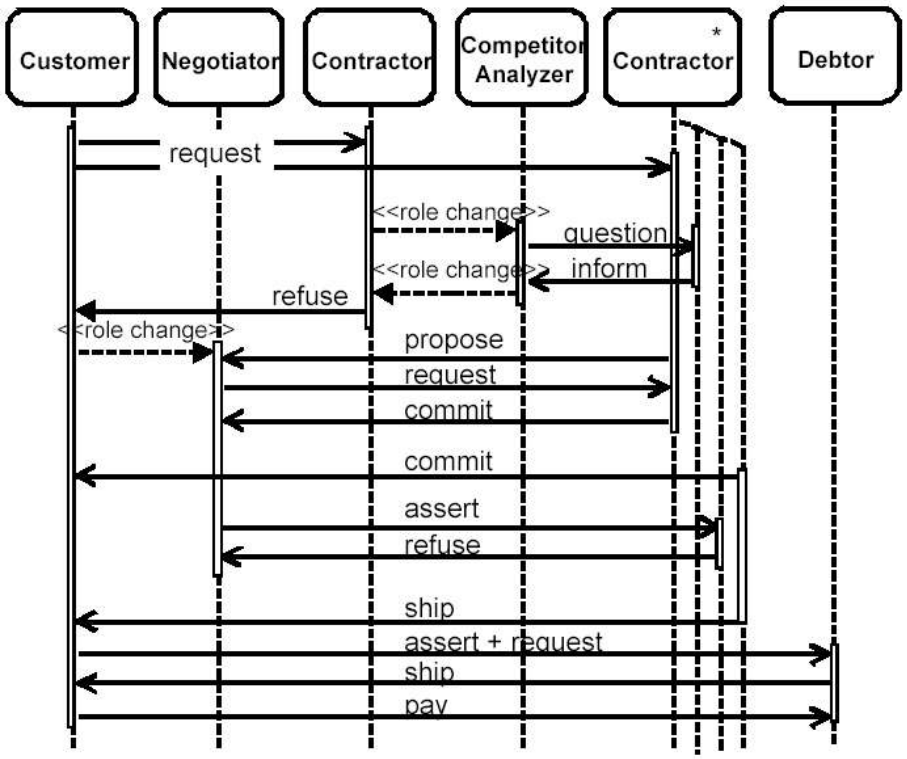


Figura 3-3.21 Rappresentazione alternativa di figura 3-20

La figura 3-20 riporta diverse linee di vita per ogni agente, una per ogni ruolo che rappresenta; la figura 3-21, invece, si concentra soltanto sui ruoli (il

protocollo rappresentato è lo stesso). La complessità di questi diagrammi è evidente, anche se gli agenti descritti sono solo quattro, ognuno con due o tre ruoli.

L'AUML propone due possibili estensioni per rendere più agevole la rappresentazione dei ruoli nelle conversazioni. La stessa interazione delle figure 3-20 e 3-21 viene descritta nelle due figure seguenti utilizzando le nuove notazioni.

In figura 3-22, gli agenti sono indicati soltanto una volta ciascuno, e ad ogni linea di vita è associato un ruolo indicato dallo stereotipo "role"; in figura 3-23, invece, le barre di attivazione vengono "etichettate" con il nome del ruolo.

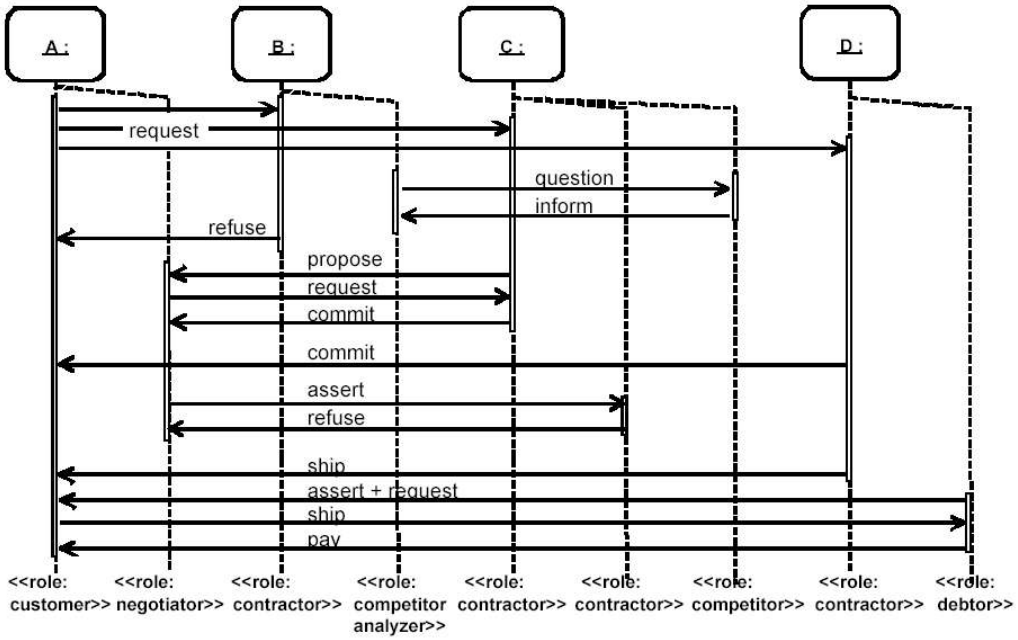


Figura 3-3.22 Modellamento dei ruoli con l'AUML

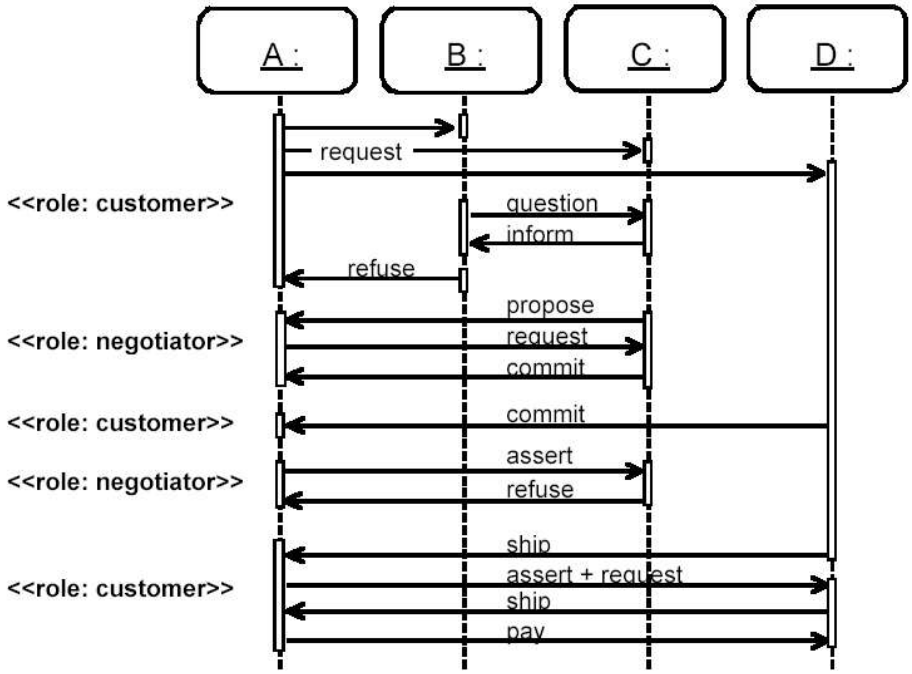


Figura 3-3.23 Rappresentazione alternativa di figura 3-22

Ovviamente è possibile anche utilizzare i diagrammi di collaborazione, che, come si è già detto, hanno lo stesso potere espressivo dei diagrammi di sequenza.

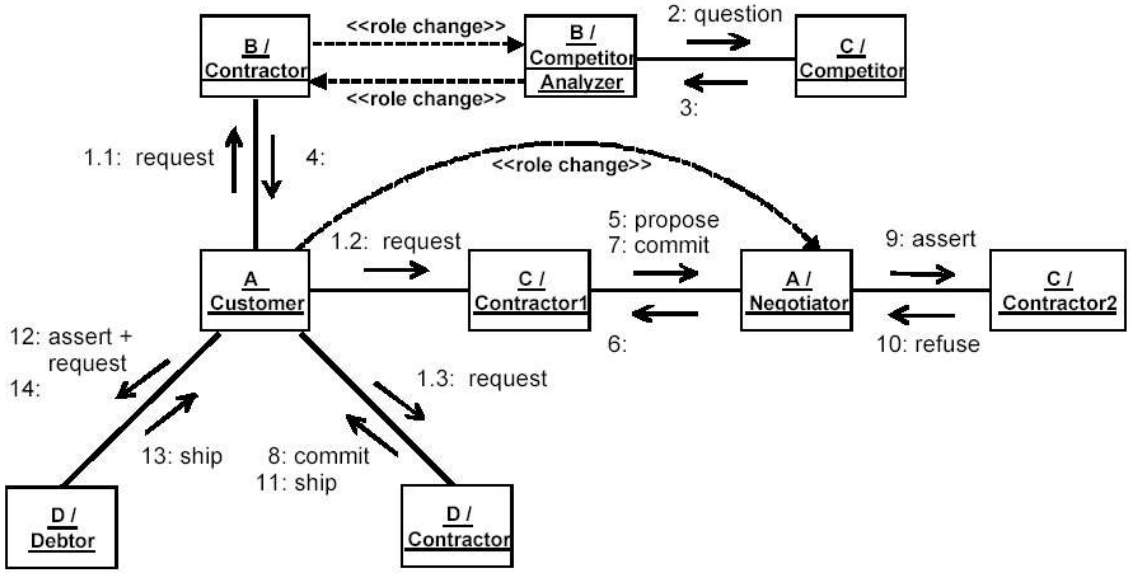


Figura 3-3.24 Collaboration diagram AUML del protocollo di figura 3-20

Altre estensioni proposte per i diagrammi di collaborazione e per i diagrammi di attività sono riportate in [Odell00].

3.6.8 Stereotipi

In questa sezione sono presentati brevemente alcuni stereotipi proposti per rappresentare concetti utilizzati di frequente dalla comunità degli sviluppatori di sistemi ad agenti. Si noti come molti stereotipi siano metafore ispirate al mondo biologico.

Stereotipo	Diagrammi	Significato
at-home	deployment	Un agente mobile è "at-home" quando si trova nella macchina dove è stato creato.
clone	classi	Un agente è un clone di un altro agente.
clone	interazione	Un agente crea un clone di un altro agente.
emergence	classi	L'interazione tra gli agenti può dar luogo ad effetti secondari (molti agenti si comportano come un'unica entità).
host/parasite	classi	Due agenti stanno tra loro come un organismo ospite ed un parassita.
mitosis	interazione	Un agente "si divide in due" come avviene per le cellule.
mobile	deployment	Aggiunto ad una freccia tratteggiata per indicate che un agente può migrare da una macchina ad un'altra.
prototype	classi	Un agente è il prototipo da cui altri agenti vengono creati per clonazione.
reproduction	interazione	L'unione di due agenti dà luogo alla nascita di un nuovo agente, come avviene nella riproduzione sessuata.
symbiosis	classi	Due agenti stanno tra loro come due organismi in simbiosi.

Tabella 3-3.IV Stereotipi AUML

Capitolo 4: Strumenti software per lo sviluppo di MAS

4.1 Introduzione

Esistono numerosi strumenti software creati per semplificare la programmazione ad agenti: sono costituiti per lo più da librerie e tool che guidano l'utente durante le fasi di progetto, implementazione e collaudo di sistemi multi-agente.

Non vi è una denominazione universalmente accettata per tali pacchetti, che sono chiamati di volta in volta "infrastrutture ad agenti" (agent infrastructures), "piattaforme ad agenti" (agent platforms), "toolkit per la costruzione di sistemi multi-agente" (multi-agent systems building toolkits) e così via. La denominazione adottata in questa tesi è *framework per lo sviluppo di sistemi multi-agente*.

In questo capitolo vengono analizzati alcuni dei più diffusi framework, e sono esposte le ragioni che hanno portato alla scelta di FIPA-OS per lo sviluppo del sistema di scheduling.

4.2 Criteri di analisi

Il confronto tra framework è un problema complesso, fin dalla scelta di quali piattaforme prendere in analisi: gli strumenti disponibili sono infatti troppo numerosi perché sia possibile portare a termine una ricerca esaustiva sull'argomento (si vedano i link riportati in §4.5 per rendersi conto della quantità di framework esistenti). Nuove piattaforme vengono rilasciate di frequente, mentre quelle esistenti sono in continua evoluzione; da ciò segue inevitabilmente che le valutazioni effettuate nella presente tesi si riferiscono soltanto all'attuale stato dell'arte, e che i framework qui descritti potrebbero essere soggetti a radicali mutamenti nel tempo.

La creazione di strumenti per lo sviluppo di MAS è iniziata nelle università e nei laboratori di ricerca, ma recentemente l'interesse in questo campo si è allargato anche all'industria del software. Ciò è certamente un fatto positivo per l'intera comunità degli sviluppatori di agenti: questa evoluzione prospetta molti vantaggi, soprattutto per quanto riguarda il raggiungimento di livelli di qualità "industry-like" da parte dei MAS (ad esempio attraverso l'introduzione di tecniche avanzate di ingegneria del software).

È bene sottolineare che i criteri con cui si giudicano i framework dipendono fortemente dal particolare problema che si intende risolvere; inoltre un framework può essere valutato solo confrontandolo con altri: questo "relativismo" è dato dal fatto che attualmente non esistono "scale di misura" assolute per le piattaforme ad agenti.

4.2.1 Confronti tra framework

Il problema di confrontare tra loro diversi strumenti per lo sviluppo di MAS è già stato affrontato da altri ricercatori.

Alcuni svolgono uno studio molto dettagliato ma limitato a particolari argomenti: ad esempio [Demaz00] si propone di determinare se i framework

esaminati forniscano un supporto per tutte le fasi del *ciclo di vita* di un sistema multi-agente: **analisi**, **progetto**, **implementazione** e **deployment**[10]. Al contrario, [Fons01] si concentra sul *behaviour*, ovvero su come un framework permette allo sviluppatore di definire il comportamento degli agenti.

In altri casi l'analisi è troppo superficiale: [CybSurv] e [JivComp] sono confronti tratti da manuali di framework specifici, che si limitano ad elencare le principali caratteristiche di alcune piattaforme ed a confrontarle con il "proprio" framework (soprattutto per mettere in risalto i pregi di quest'ultimo).

L'obiettivo di questo capitolo è esaminare alcuni tra i più diffusi framework, cercando di descriverne, per quanto possibile, tutte le caratteristiche in modo né troppo superficiale né troppo dettagliato, in parte prendendo come modello la bibliografia sopra citata.

Le principali linee guida dell'analisi effettuata sono descritte nei paragrafi seguenti.

4.2.2 Architettura di un framework

Un framework per lo sviluppo di sistemi multi-agente è generalmente composto di tre elementi essenziali:

- Strumenti di sviluppo (analisi e progetto)
- Libreria (implementazione)
- Strumenti runtime (deployment)

Gli *strumenti di sviluppo* sono programmi, solitamente dotati di interfaccia grafica, che assistono l'utente nella fase di analisi e progetto di un sistema; ne

[10] Il deployment (rilascio o distribuzione) è l'applicazione della soluzione sviluppata al problema reale in un dato dominio. In pratica consiste nell'avviare il sistema ad agenti (tipicamente su una rete di computer) e quindi di collaudarlo, effettuarne la manutenzione e/o estenderne le funzionalità.

esistono di diversi tipi, dai tool più elementari fino a veri e propri strumenti CASE[11]. Spesso generano automaticamente parte del codice necessario per l'implementazione.

La *libreria* è una collezione di API (generalmente si tratta di una *libreria di classi*) che forniscono funzionalità specifiche per gli agenti – principalmente per quanto riguarda l'identificazione, il ciclo di vita di un agente e soprattutto la comunicazione e la coordinazione.

Gli *strumenti runtime* si dividono in

- Ambiente di esecuzione: un programma o un insieme di programmi che costituiscono una "cornice" per il mondo degli agenti, e che possono fornire diversi servizi agli agenti stessi.

- Strumenti di monitoraggio, controllo e debugging degli agenti.

Dei tre elementi, soltanto la *libreria* è sempre presente, infatti esistono framework che non comprendono alcun tipo di tool. Per ogni framework è indispensabile studiare a fondo ciascuna delle tre componenti (quando presenti), esplorandone le funzionalità.

4.2.3 Il software "open source"

Nella pletora di framework esistenti, si è deciso di prendere in esame soltanto quelli open source e basati su Java.

La seguente definizione di open source è tratta da [Cammarata]: "modello di diffusione dei sistemi operativi e dei programmi caratterizzato da due elementi essenziali: il primo è la diffusione del codice sorgente, cioè delle istruzioni scritte dagli umani per far funzionare le macchine, in modo che chiunque ne abbia la capacità possa capire come sono fatte, suggerire correzioni, proporre

[11] Computer-Aided Software Engineering.

miglioramenti e aggiunte; il secondo è la gratuità o il basso costo dei codici stessi e delle soluzioni applicative che su tali codici sono sviluppate".

Open source dunque non significa semplicemente che l'utente ha libero accesso al codice sorgente. Un programma o framework open source è, come dice la parola stessa, un "sistema aperto" che chiunque può contribuire a sviluppare: l'esempio più famoso in tal senso è il sistema operativo Linux. Spesso si è erroneamente portati a credere che open source sia sinonimo di aleatorietà ed incertezza, ma non è così.

I vantaggi dell'open source possono essere così riassunti (tutte le sezioni tra virgolette sono tratte da [Cammarata]):

- "trasparenza dei prodotti software, con la possibilità di controllare che cosa c'è realmente dentro e di modificarli liberamente per adattarli a specifiche esigenze";
- "non dipendenza da un singolo fornitore, ma possibilità di scegliere su un mercato più vasto e concorrenziale, con evidenti riflessi sui prezzi";
- "sensibili risparmi che derivano dalla riusabilità dei codici e delle applicazioni, non legata ai contratti-capestro oggi imposti dall'industria del software: non è vero che "l'open source è gratis", come a volte si afferma con una certa superficialità, ma è la materia prima che costa poco o nulla, mentre il lavoro di personalizzazione deve essere pagato";
- "sviluppo delle economie locali e dell'occupazione, determinato dal fatto che la produzione degli applicativi può essere affidata ad aziende indipendenti".

Lo svantaggio dell'open source può essere dato da "una scelta di applicazioni ancora non abbastanza vasta e da una non completa interoperabilità".

I framework studiati nel seguito sono tutti open source: alcuni di essi sono sistemi accademici, creati cioè da ricercatori universitari. Altri invece sono stati

sviluppati da imprese private che hanno deciso tuttavia di non farne pacchetti commerciali, ma di metterli a disposizione della comunità internazionale.

4.2.4 I sistemi ad agenti e il linguaggio Java

Per quanto riguarda i **linguaggi di programmazione**, non ci sono dubbi sul fatto che **Java** sia il preferito dalla comunità degli sviluppatori di agenti.

La maggior parte dei framework disponibili sono scritti in Java: sono molto poche le piattaforme che utilizzano altri linguaggi, e tra i più diffusi troviamo C++, Lisp, Smalltalk/VisualWorks e Prolog. È opportuno citare anche alcuni linguaggi speciali creati appositamente per sviluppare agenti: tra essi ricordiamo COOL, (COOrdination Language) e ADL (Agent Definition Language).

Creato dall'azienda americana Sun Microsystem, Java presenta diverse caratteristiche che lo rendono molto attraente per lo sviluppo di sistemi ad agenti. Si tratta di un linguaggio indipendente dalla piattaforma, la cui architettura è studiata in modo che risultino completamente portabili sia il codice sorgente che il codice compilato ("bytecode"). La sintassi di Java deriva dal C++, uno dei linguaggi di programmazione più popolari, quindi non è difficile da imparare, ma a differenza del suo predecessore può essere definito un linguaggio *puramente object-oriented* (evita cioè il "compromesso" con la programmazione procedurale tipico del C++). Altre funzionalità interessanti sono la gestione semplificata del multi-threading e della comunicazione attraverso la rete, il trattamento delle eccezioni, la serialization e la reflection.

Per quanto riguarda la gestione delle applicazione in rete, Java adotta diversi meccanismi, ma il più utilizzato è **RMI** (Remote Method Invocation), una tecnica di comunicazione tra oggetti che estende la "classica" architettura RPC (Remote Procedure Call): in pratica consente di invocare metodi di oggetti che si trovano fisicamente su un computer remoto come se fossero oggetti locali. Molti framework sfruttano RMI per la comunicazione tra agenti.

La **serialization** (serializzazione) è un meccanismo che permette di rendere persistenti gli oggetti, salvandone lo stato (ad esempio su disco).

La **reflection** (riflessione) è una funzionalità che pochi ambienti di programmazione possiedono: la possibilità di accedere dinamicamente a classi, oggetti e metodi, "scoprendo" a runtime i loro nomi (rappresentati da stringhe). In Java questa è una caratteristica intrinseca del linguaggio[12].

La versione di Java utilizzata è una caratteristica importante: alcune piattaforme ad agenti si basano ancora sulle vecchie versioni del linguaggio, ovvero Java 1.0 e 1.1. Nell'analisi sono stati presi in considerazione soltanto quei framework basati sulle versioni più recenti, 1.2 e 1.3, chiamate anche, nel loro complesso, Java2 (Sun ha rilasciato recentemente la versione 1.4, ma ancora non è supportata da nessuna piattaforma ad agenti).

Per concludere il discorso su Java, l'estrema popolarità di questo linguaggio è dimostrata anche dall'esistenza di una comunità[13], chiamata "*Progetto JAS*" (Java Agent Services), che si propone di estendere la libreria di Java definendo un insieme di API standard che implementino le specifiche FIPA.

4.2.5 Standard e metodologia

I sistemi multi-agente sono software molto complessi; la costruzione di tali sistemi necessita dunque di un approccio metodico e ingegneristico. In questo campo, molto più che in altri settori dell'ingegneria del software, è essenziale

[12] Un altro ambiente molto noto che fornisce in modo "nativo" la reflection è il framework Microsoft .NET.

[13] Per la precisione si tratta di un "Java Community Process", si veda a questo proposito l'indirizzo http://jcp.org/

l'adozione di uno standard o di una metodologia rigorosa: spesso, purtroppo, i framework si rivelano essere semplici infrastrutture di implementazione, e i manuali e tutorial disponibili spiegano *che cosa* fare (come utilizzare la libreria ed i tool di implementazione), mentre l'utente vorrebbe sapere soprattutto *come* fare (date certe specifiche, come costruire un sistema ad agenti).

Per questo motivo si è deciso di considerare solo i framework che soddisfassero una delle due seguenti condizioni:

1) Aderire ad uno standard

2) Adottare una ben precisa metodologia

Al momento, gli unici standard a cui è possibile fare riferimento sono le specifiche FIPA, introdotte nel capitolo precedente. In questo capitolo verranno anche studiati alcuni framework non-FIPA perché ugualmente interessanti, soprattutto dal punto di vista della metodologia di sviluppo adottata.

Poiché, come si è già detto, la scelta di un framework dipende anche dal tipo di utilizzo a cui esso è destinato, si è deciso di privilegiare quelli che offrivano un'implementazione semplice e ben supportata. La maggior parte dei toolkit promette di consentire uno "sviluppo rapido" delle applicazioni ("*rapid developement*"), ma non sempre queste ottimistiche promesse vengono mantenute.

Infine, in questa ricerca non sono stati presi in considerazione quei framework che si concentrano soprattutto sugli "agenti mobili", in quanto al momento la mobilità non è necessaria al sistema per lo scheduling ad agenti che verrà presentato nel Capitolo 5.

4.2.6 Selezione dei framework e testing

Oltre ai criteri sopra riportati, se ne può aggiungere un altro: le piattaforme prese in esame devono essere diffusamente utilizzate e far parte di progetti ancora

attivi, in modo da essere soggetti ad una regolare manutenzione (sia correttiva che evolutiva).

In base a tutte queste considerazioni si è scelto di valutare i seguenti sei framework:

- AgentTool
- FIPA-OS
- JADE
- JiVE
- OpenCybele
- Zeus

Ciascun framework è stato scaricato ed installato su una sola macchina, quindi lo si è esaminato in tutte le sue parti. Per valutare le capacità di ogni piattaforma, la semplicità d'uso e le altre caratteristiche, si è costruito un semplice caso di test qui di seguito descritto.

Si supponga di avere un sistema composto da due soli tipi di agenti, *RequesterAgent* e *ProviderAgent*. Ad intervalli di tempo regolari, un RequesterAgent invia un messaggio richiesta ("request") a tutti i ProviderAgent. Ciascun ProviderAgent, quando riceve una richiesta, decide se soddisfarla o no (il criterio decisionale qui è irrilevante, può essere semplicemente estratto un valore casuale), e manda di conseguenza al mittente una risposta "agree" oppure "refuse". Un RequesterAgent muore quando riceve almeno una risposta "agree".

Come si avrà modo di vedere nel prossimo capitolo, tale test non è stato scelto a caso, ma è vagamente "ispirato" al problema reale che si intende affrontare in questa tesi: lo scheduling.

4.2.7 Struttura dell'analisi

Ad ogni framework è dedicato un paragrafo separato, diviso in diverse sezioni:

Introduzione

Nell' *Introduzione* si trovano informazioni generali sul framework, chi lo ha prodotto, qual è l'ultima versione disponibile e a che data risale. Inoltre viene specificata la versione di Java su cui si basa il framework.

Caratteristiche del framework

In questa sezione sono presi in esame tutti gli strumenti di supporto all'analisi/progetto e al deployment, ovvero i tool grafici e le altre utility caratteristiche del framework, quando presenti (il supporto per l'implementazione, cioè la libreria, viene esaminato in una sezione a parte). Particolare importanza viene data a eventuali strumenti per la generazione automatica del codice Java (i quali, se efficienti, semplificano molto l'implementazione) e a strumenti di *debugging* e controllo degli agenti durante l'esecuzione.

Sempre in questa sezione viene espresso un giudizio personale sulla documentazione fornita; inoltre, viene indicato se il framework è conforme allo standard FIPA (*FIPA-compliant*) e/o adotta qualche metodologia di sviluppo (es. MaSE).

Descrizione della libreria:

Indipendentemente dai tool presenti, il cuore di un framework è la *libreria di classi Java* che fornisce. In questa sezione sono descritte le caratteristiche principali della libreria, in che cosa essa è diversa dalle altre, se è semplice da usare, quali i suoi punti di forza e gli svantaggi. Fonti per tale analisi sono non

soltanto i manuali forniti e la documentazione delle API[14], ma soprattutto l'esperienza personale derivata dal testing e, quando necessario, l'analisi del codice sorgente.

Esaminando la libreria bisogna trovare risposte a diverse domande, tra cui:

- Che cos'è un agente, come viene definito dal programmatore? Qual è il meccanismo che consente di creare gli agenti?

  Generalmente un agente è definito come l'estensione (per ereditarietà) di una data classe base, che fornisce alcuni servizi predefiniti all'agente.

  Di solito un agente viene creato insieme all'oggetto corrispondente, con l'invocazione di un costruttore, oppure attraverso metodi speciali forniti dalla libreria. Alcuni framework supportano la creazione di più agenti nella stessa Java Virtual Machine, mentre con altri framework l'utente deve eseguire ogni agente in una JVM separata, oppure gestire esplicitamente i thread.

- Come è definito il comportamento degli agenti?

  Alcuni framework adottano formalismi per la definizione del comportamento, ad esempio sistemi rule-based o macchine a stati.

- In che modo comunicano gli agenti?

  Questo punto è di fondamentale importanza, dato che le potenzialità dei sistemi multi-agente risiedono proprio nella capacità degli agenti di interagire. Occorre analizzare la tecnica adottata per lo scambio di messaggi, nonché l'eventuale presenza di agenti con funzioni di coordinamento, come router/nameserver (chiamato anche AMR o AMS) per trovare gli agenti in base al nome, e/o facilitator per trovare agenti in base alle caratteristiche.

[14] La documentazione delle classi Java viene solitamente generata automaticamente dallo strumento Javadoc (fornito da Sun) che crea un insieme di file HTML estraendo informazioni dallo stesso codice sorgente e dai commenti.

- Qual è il formato dei messaggi scambiati?

  I "linguaggi di comunicazione tra agenti" (agent communication languages) più diffusi sono KQML, FIPA-ACL ed eventuali formati "proprietari".

  Supporto:

Quando possibile, è stata valutata l'efficienza del supporto on-line. Particolarmente importanti sono, quando presenti, le mailing list di sviluppatori e i sistemi di bug-report.

Commenti:

Nell'ultima sezione viene espresso un giudizio globale sul framework, basato soprattutto sull'esperienza di utilizzo e sui risultati del testing.

4.3 Analisi dei framework

4.3.1 AgentTool

Introduzione:

AgentTool 1.8.3-beta, nato da una collaborazione tra la Kansas State University (KSU) e l'Air Force Institute of Technology (AFIT) è stato sviluppato dal team "Agent Lab" guidato dal professor Scott DeLoach. L'ultima versione è stata rilasciata l'11 febbraio 2002. É basato sulla versione 1.3 di Java.

Caratteristiche del framework:

AgentTool è principalmente uno strumento di *progetto*. Possiede un ambiente grafico che consente di tracciare diagrammi che descrivono un sistema ad agenti in ogni sua parte.

Non è FIPA-compliant, ma segue una metodologia di sviluppo chiamata MaSE (Multi-agent System Engineering, anch'essa sviluppata da DeLoach) che è in grado di coprire le seguenti fasi di sviluppo:

Passaggio MaSE	***Diagrammi AgentTool***
Definizione degli obiettivi	Gerarchia degli obiettivi (goals)
Applicazione degli Use Case	Sequence diagrams
Definizione dei ruoli	Diagrammi dei ruoli e dei task
Progetto delle classi di agenti	Diagrammi a classi degli agenti
Progetto delle conversazioni	Diagrammi a stati delle conversazioni
Costruzione degli agenti	Diagrammi dei componenti degli agenti
Progetto e rilascio del sistema	Deployment diagrams

I diagrammi MaSE utilizzati in AgentTool sono simili a quelli UML.

Il framework include Spin, un tool per la verifica automatica della coerenza delle conversazioni. Non sono presenti tool per il controllo in runtime, come del resto non esiste un vero e proprio ambiente di esecuzione.

Per quanto riguarda l'implementazione, AgentTool è in grado di generare tre tipi diversi di codice: per AgentMom, per Carolina e per AWL. AgentMom è una libreria sviluppata da DeLoach ed è inclusa in AgentTool; Carolina è un framework in corso di sviluppo presso l'Università del Connecticut nell'ambito del progetto MADSG; AWL è un linguaggio di modellamento sviluppato dall'Air Force Institute of Technology.

La documentazione di AgentTool e di MaSE è molto buona, mentre quella di AgentMom è piuttosto scarsa.

Descrizione della libreria:

Nel seguito viene presa in considerazione soltanto AgentMom, una libreria di classi semplice, ma estremamente limitata.

- Che cos'è un agente, come viene definito dal programmatore? Qual è il meccanismo che consente di creare gli agenti?

  Un agente è una classe che eredita dalla classe `Agent`, la quale di per sé fornisce pochissime funzionalità. AgentTool crea automaticamente una sottoclasse di `Agent` per ogni agente definito, generando buona parte del codice necessario.

  Un agente è creato invocandone il costruttore. Poiché `Agent` implementa l'interfaccia `Runnable`, ogni agente può essere eseguito in un singolo thread, ma la gestione dei thread è a carico dell'utente.

- Come è definito il comportamento degli agenti?

  Il comportamento è definito da estensioni della classe `Conversation`. AgentTool definisce le conversazioni con un formalismo a stati, e per ogni conversazione genera due sottoclassi di `Conversation`: una per l'agente che inizia la conversazione, e una per l'agente che risponde. Ogni conversazione è eseguita in un thread separato.

- In che modo comunicano gli agenti?

  Gli agenti comunicano direttamente via socket: ognuno ascolta una diversa porta TCP, e possiede un oggetto `MessageHandler` che gestisce le nuove conversazioni in arrivo. Un agente invia e riceve messaggi in modo sincrono o asincrono tramite i metodi della classe `Conversation`. Non esistono agenti coordinatori, se un agente vuole comunicare con un altro agente deve conoscerne la porta TCP.

- Qual è il formato dei messaggi scambiati?

Gli agenti si scambiano un oggetto `Message` proprietario, i cui campi ricordano la sintassi del KQML.

Supporto:

Non esiste un supporto on-line esplicito, a parte ovviamente l'indirizzo di posta elettronica di Scott DeLoach. Esiste una mailing list, ma serve solo per ricevere notizie (gli utenti non possono spedire messaggi alla lista).

Commenti:

AgentTool è ottimo come strumento di progetto: in particolare la metodologia MaSE è molto interessante, e può essere impiegata per progettare un sistema ad agenti indipendentemente dall'implementazione. Tuttavia i diagrammi MaSE sono diversi dai diagrammi AUML (vedi §3.6).

Il test di AgentTool si è svolto correttamente, tranne per quanto riguarda l'esecuzione di Spin, il tool di verifica automatica, che ha dato dei problemi.

I maggiori svantaggi si presentano per quanto riguarda l'implementazione: la libreria AgentMom non è adatta per realizzare progetti complessi, presenta qualche incongruenza con il codice generato da AgentTool, e la comunicazione socket ha dato problemi in fase di test. Il principale difetto di AgentMom è l'assenza di un facilitator, da cui segue la necessità per gli agenti di conoscere a priori le rispettive porte.

4.3.2 FIPA-OS

Introduzione:

FIPA-OS 2.1.0 è stato sviluppato da Nortel Networks/Emorphia, un'impresa inglese; OS sta per open source. L'ultima versione è del 3 luglio 2001.

Si basa sulla versione 1.2 di Java, ma sono anche disponibili due edizioni per 1.1 (una normale e una "compatta", adatta ad esempio per palmari).

Caratteristiche del framework:

Per la fase di sviluppo FIPA-OS fornisce soltanto lo strumento TaskGenerator: data una definizione di protocollo, il tool genera automaticamente del codice Java che semplifica l'implementazione del protocollo stesso (in pratica crea delle sottoclassi di `Task` – si veda la sezione seguente)

Per quanto riguarda il supporto runtime, sono presenti alcuni semplici strumenti: AgentLoader (che consente di avviare ed arrestare gli agenti), Thread Pool Monitor e Task Manager Monitor (per controllare lo stato degli agenti) e IOTestAgent (invia manualmente messaggi agli agenti). Per tenere traccia degli eventi che si susseguono in un MAS, FIPA-OS non fornisce tool grafici, bensì una tecnica semplificata per la generazione di un file log.

La documentazione è molto buona, sono anche disponibili dei tutorial. Come dice il nome stesso, questo framework è FIPA-compliant.

Descrizione della libreria:

La gestione degli agenti segue gli standard definiti da FIPA.

- Che cos'è un agente, come viene definito dal programmatore? Qual è il meccanismo che consente di creare gli agenti?

Un agente è costituito da una parte centrale (l'agente stesso) e da nessuno o più task. Un agente è una classe che eredita dalla classe `FIPAOSAgent`,

quindi estende i metodi delle classi base (analogamente, un task eredita dalla classe `Task`).

Un agente può essere creato attraverso l'interfaccia grafica di AgentLoader, oppure da un altro agente che ne invoca il costruttore. FIPA-OS supporta bene la creazione di più agenti nella stessa Virtual Machine: tali agenti condividono la console e non possono generare output usando i classici metodi di output. FIPA-OS fornisce una classe di utilità `DIAGNOSTICS`, che consente ad un agente di scrivere sullo schermo dando al proprio output una priorità – inoltre tutti gli output vengono anche registrati automaticamente su un file di log che può essere analizzato in seguito.

- Come è definito il comportamento degli agenti?

Lo sviluppatore definisce il comportamento estendendo la classe `Task` (esistono alcuni task predefiniti, ma non sono molti). I task sono eseguiti concorrentemente, in modo sincrono o asincrono, e la gestione del multi-threading è totalmente svolta dal framework, grazie alla classe `TaskManager` (né `FIPAOSAgent` né `Task` ereditano da `Thread`: il sistema gestisce un "thread pool" e i thread vengono assegnati dinamicamente ad un agente o ad un task quando necessario). Un task genera, se lo desidera, dei task-figli, e può decidere se condividere o no dei dati con essi. Quando un task-figlio termina, l'evento viene comunicato al genitore, attraverso l'invocazione di un metodo `doneNomeDelTask()`: in questo modo un figlio può "restituire" un valore al genitore. I nomi dei metodi da invocare vengono determinati al momento dell'esecuzione tramite la reflection di Java.

Di default, non c'è un formalismo per definire il comportamento di un agente, anche se FIPA-OS comprende alcune classi che consentono di definire un comportamento rule-based utilizzando JESS, Java Expert

System Shell. Un agente che utilizza un motore JESS deve ereditare da `JessAgent` invece che da FIPAOSAgent.

- In che modo comunicano gli agenti?

  Le classi base (`FIPAOSAgent` e `Task`) forniscono numerosi metodi per semplificare la gestione della comunicazione. L'ambiente di esecuzione comprende, secondo le specifiche FIPA, un agente manager (AMS), un agente facilitator (DF) che consente di cercare gli agenti in base alle loro caratteristiche, e (opzionalmente) un Agent Communication Channel (ACC) che consente di dialogare con altre piattaforme FIPA-compliant (ma non necessariamente FIPA-OS).

  Un agente invia messaggi ACL attraverso metodi ereditati da Task, quali `forward()` e `searchDF()`. La comunicazione è soltanto punto-punto e asincrona. Un agente riceve messaggi attraverso metodi invocati dinamicamente (con la reflection) in base al tipo di messaggio in arrivo (ad esempio una richiesta viene gestita dal metodo `handleRequest()`). I messaggi vengono sempre inviati al task effettivamente coinvolto nella conversazione (il tutto è svolto automaticamente dal framework), e se il messaggio in arrivo non appartiene a nessuna conversazione già iniziata viene trattato dal task di default (l'*ascoltatore*, detto anche "idle task").

  FIPA-OS possiede un meccanismo automatico di "protocol check": un protocollo di interazione tra agenti è definito come un grafo all'interno di una classe particolare (tutti i protocolli standard FIPA sono già definiti).

  A livello più basso (trasporto) FIPA-OS usa le tecniche RMI e IIOP[15]. La prima privilegia l'efficienza ed è di solito utilizzata tra agenti FIPA-OS, la seconda è FIPA-compliant ma meno efficiente, ed è utilizzata per dialogare tra piattaforme diverse.

- Qual è il formato dei messaggi scambiati?

[15] Vedi §3.4.6

FIPA-OS supporta il linguaggio FIPA-ACL tramite varie classi di utilità, come la classe `ACL`; inoltre possiede classi per codificare-decodificare i vari linguaggi utilizzati come *contenuto* dei messaggi ACL (XML, SL, RDF).

Supporto:

Il supporto on-line è molto efficiente (è possibile iscriversi alla mailing list degli sviluppatori di FIPA-OS). Inoltre è presente una pagina per il bug report.

Commenti:

Anche se offre un supporto molto piccolo per l'analisi/progetto, si tratta di un ottimo framework. I suoi vantaggi e svantaggi rispetto alle altre piattaforme vengono riassunti nella sezione §4.4, insieme ai motivi per cui si è scelto di utilizzarlo per lo sviluppo del sistema di scheduling multi-agente.

4.3.3 JADE

Introduzione

JADE 2.5 ("Java Agent Developement framework") è stato sviluppato da TILab S.p.A in collaborazione con l'Università di Parma. L'ultima versione è del 5 febbraio 2002. È basato sulla versione 1.2 di Java.

Caratteristiche del framework

Non sono presenti strumenti di sviluppo, mentre il supporto runtime è molto buono: dall'interfaccia grafica di Jade (che è essa stessa un agente, chiamato RMA, Remote Management Agent) è possibile gestire e monitorare anche gli agenti che risiedono su altre piattaforme. Uno strumento estremamente utile per il collaudo di un sistema è l'agente Sniffer, che analizza le conversazioni che avvengono all'interno del sistema: l'utente sceglie gli agenti che desidera

monitorare, e lo Sniffer traccia in tempo reale un diagramma di sequenza UML comprendente tutti i messaggi scambiati da quegli agenti.

La documentazione della libreria di classi è molto buona. Il framework JADE è FIPA-compliant.

Descrizione della libreria:

La libreria di JADE segue le specifiche FIPA ed è nel complesso molto simile a quella di FIPA-OS.

- Che cos'è un agente, come viene definito dal programmatore? Qual è il meccanismo che consente di creare gli agenti?

  Come in FIPA-OS, un agente è costituito da una parte centrale (l'agente stesso) e da nessuno o più comportamenti. Un agente è una classe che eredita dalla classe `Agent`, mentre un comportamento eredita dalla classe `Behaviour`.

  L'ambiente di JADE si avvia tramite la classe `Boot`, e un agente può essere creato attraverso l'interfaccia grafica RMA, oppure da un altro agente: un particolare interessante è che un agente che ne crea un altro non riceve mai indietro un riferimento esplicito all'oggetto-agente creato (questo garantisce una maggiore indipendenza tra gli agenti). JADE supporta bene la creazione di più agenti nella stessa Virtual Machine. Un agente può creare un proprio log tramite il metodo `write()`.

- Come è definito il comportamento degli agenti?

  Lo sviluppatore definisce il comportamento estendendo la classe `Behaviour`: JADE fornisce già alcune sottoclassi predefinite. Il più semplice comportamento possibile è `OneShotBehaviour`, che svolge i propri compiti una sola volta, mentre per ripetere più volte delle operazioni è necessario usare `CyclicBehaviour`. Un behaviour può essere

complesso, ovvero composto da più behaviour-figli: a seconda di come si vuole che tali figli siano eseguiti, JADE include le classi `SequentialBehaviour`, `ParallelBehaviour` e `FMSBehaviour` (quest'ultimo tratta i behaviour-figli come stati di una macchina a stati finiti).

Ad ogni agente è assegnato un thread (`Agent` implementa `Runnable`), ai behaviour no: l'esecuzione concorrente dei behaviour è gestita da uno scheduler interno all'agente (nascosto all'utente). Il codice principale di un behaviour è contenuto nel metodo `action()`.

Di default, non c'è un formalismo per definire il comportamento di un agente; il framework include un esempio che illustra come un agente JADE possa interagire con il sistema esperto JESS.

- In che modo comunicano gli agenti?

L'ambiente di esecuzione, secondo le specifiche FIPA, comprende gli agenti AMS e DF, oltre all'Agent Communication Channel (ACC).

Un agente invia messaggi ACL attraverso il metodo `send()` della classe `Agent` (i behaviour non hanno metodi per gestire i messaggi, e devono quindi accedere alla classe `Agent`). La ricezione dei messaggi può essere asincrona o sincrona, a seconda che si usi `receive()` o `blockingReceive()` (il secondo metodo è tuttavia pericoloso, perché sospende tutte le attività dell'agente, compresi i behaviour). La comunicazione può essere punto-punto o multicast. La coda dei messaggi in arrivo di un agente è unica (non esiste un meccanismo di dispatching dinamico tra i behaviour), e per decidere quali messaggi ricevere è possibile specificare un `MessagePattern` (argomento opzionale di `receive()`).

JADE include il meccanismo AchieveRE (Rational Effect) per implementare i protocolli di interazione FIPA o altri protocolli, ma il suo

uso non è semplice come il TaskGenerator di FIPA-OS (non utilizza la reflection).

A livello più basso JADE usa i protocolli di trasporto RMI e IIOP, proprio come FIPA-OS, oltre ad HTTP.

- Qual è il formato dei messaggi scambiati?

  JADE supporta il linguaggio FIPA-ACL tramite la classe `ACLMessage`, inoltre include un codec per il linguaggio SL. Una caratteristica interessante di JADE è il supporto per la definizione dell'ontologia: lo sviluppatore crea un insieme di classi che rappresentano concetti, predicati, azioni e proposizioni, oltre ad una classe che includa riferimenti a tutte le classi che costituiscono l'ontologia. Una volta registrata tale classe (con il metodo `registerOntology()` della classe `Agent`), la gestione dell'ontologia è svolta dal framework (che usa la reflection per invocare i metodi delle classi).

Supporto:

Il supporto on-line è molto efficiente (è possibile iscriversi alla mailing list degli sviluppatori di JADE).

Commenti:

Anche questo è un ottimo framework. In generale, le funzionalità offerte sono molto simili a quelle di FIPA-OS (ulteriori dettagli sulle differenze tra i due saranno chiariti in §4.4).

4.3.4 JiVE

Introduzione:

JiVE 1.0 ("JAFMAS integrated Visual Environment") è stato sviluppato presso la Cincinnati University. Si basa sulla libreria JAFMAS 1.0 ("Java-Based Framework for Multi-Agent Systems"), una piattaforma preesistente e ideata presso la stessa università. Ricalcando la distinzione già vista per AgentTool, possiamo dire che JiVE è uno strumento di *progetto* mentre JAFMAS è un'infrastruttura di *implementazione*. È basato sulla versione 1.2 di Java (anche se il server è compatibile con 1.1).

Caratteristiche del framework:

L'ambiente di JiVE è basato su un'architettura client-server che consente, tra l'altro, a più sviluppatori di lavorare allo stesso progetto, anche *contemporaneamente* (funzioni collaborative). L'interfaccia grafica ricorda quella di AgentTool, anche se quest'ultimo è uno strumento molto più completo: in JiVE è possibile solo definire gli agenti e le conversazioni che essi instaurano. JiVE genera automaticamente il codice necessario per la libreria JAFMAS, inoltre possiede un strumento per la verifica automatica della coerenza delle conversazioni (PetriTool).

JiVE e JAFMAS sono basati su una metodologia senza nome, qui indicata semplicemente come "metodologia JAFMAS". Essa si articola in cinque passi: identificare gli agenti, identificare le conversazioni, identificare le regole delle conversazioni, analizzare il modello delle conversazioni e implementare il sistema. Uno dei punti chiave della metodologia è l'assenza di agenti coordinatori, poiché l'autore di JAFMAS considera uno svantaggio la necessità di un punto centrale a cui riferirsi.

La documentazione è buona. JiVE non può essere definito FIPA-compliant, anche se utilizza il linguaggio FIPA-ACL.

Descrizione della libreria:

La libreria è estremamente elementare (e il primo indizio di ciò è dato dal fatto che non è suddivisa in package[16], al contrario delle altre).

- Che cos'è un agente, come viene definito dal programmatore? Qual è il meccanismo che consente di creare gli agenti?

  Un agente JAFMAS è una classe che eredita dalla classe `Agent`, che a sua volta eredita da `Thread` (la gestione dei thread è a carico dell'utente). Un agente è creato invocandone il costruttore. Come nel caso di AgentTool, l'ambiente grafico di JiVE genera buona parte del codice necessario. Per default, un agente possiede una propria interfaccia grafica con un menù e un'area di testo in cui può visualizzare informazioni. La creazione effettiva degli agenti è lasciata quasi completamente all'utente, che estendendo la classe `CreateAgent` può definire un'interfaccia grafica nella quale lanciare gli agenti.

- Come è definito il comportamento degli agenti?

  Il comportamento di un agente è definito estendendo la classe `Conversation`: ogni `Conversation` è eseguita in un thread separato e possiede uno stato interno e una propria GUI. Le conversazioni sono diverse dai task e behaviour visti negli altri framework: in questo caso si tratta sostanzialmente di macchine a stati, dove ogni transizione (implementata estendendo la classe `ConvRule`) è costituita da precondizioni (stato iniziale, messaggio ricevuto), elaborazioni da compiere, e postcondizioni (messaggio inviato, stato finale). Le conversazioni sono generate dall'ambiente di JiVE.

- In che modo comunicano gli agenti?

  Grazie ai metodi della classe `Agent` un agente può comunicare direttamente con altri agenti, oppure formare "gruppi" di agenti (canali) e

[16] In Java un package è una collezione di classi e altri sotto-package, proprio come in UML.

gestire messaggi multicast con la tecnica publish&subscribe. Il modello publish&subscribe funziona in questo modo: un agente si registra (subscribe) presso uno o più canali, inoltre può inviare messaggi a tali canali. Un messaggio inviato ad un canale sarà ricevuto da tutti gli agenti iscritti a quel gruppo. Ogni agente possiede automaticamente un oggetto `MsgRouter` e tanti `MulticastCom` quanti sono i canali a cui è iscritto.

Come si è già detto, non esistono agenti nameserver o facilitator. La comunicazione tra agenti avviene solo attraverso Java-RMI.

- Qual è il formato dei messaggi scambiati?

  Sono supportati i linguaggi FIPA-ACL e KQML, oltre ad un formato proprietario di JiVE (impropriamente definito "Standard"). Il contenuto dei messaggi, tuttavia, non è definito (può essere usato qualsiasi oggetto Java).

Supporto:

Per JAFMAS esiste una mailing list dedicata, mentre non sembra essere disponibile alcun supporto on-line per JiVE (a parte, ovviamente, l'indirizzo e-mail del suo sviluppatore, Alan K. Galan).

Commenti:

JiVE è uno strumento di progetto molto buono, sebbene limitato. La documentazione teorica e metodologica di JiVE e JAFMAS è molto buona; al contrario, quella relativa alla libreria JAFMAS non è molto ricca, e per comprenderne il funzionamento è stato necessario studiare parte del codice sorgente.

4.3.5 OpenCybele

Introduzione:

OpenCybele 1.0 è la versione open source di un progetto più esteso chiamato Cybele, sviluppato da IAI, Intelligence Automation Inc, un'impresa privata americana che si occupa di ricerca nei campi più disparati. L'ultima versione è del maggio 2001. È basato sulla versione 1.3 di Java.

Caratteristiche del framework:

OpenCybele non possiede alcuno strumento di sviluppo grafico, né per la fase di sviluppo né per quella di deployment. La libreria di classi è semplice e molto ben documentata (sono disponibili manuale, esempi e Javadoc).

OpenCybele non è FIPA-compliant; è basato su una metodologia chiamata ACP (*Activity-Centric Programming*). ACP non è una metodologia di sviluppo, ma un modello concettuale basato sulle Attività, definite come insiemi di metodi event-driven. Un'Attività può essere dunque un oggetto i cui metodi vengono invocati quando si verificano particolari eventi (es. ricezione di un messaggio). Un' Attività può trovarsi in sei stati: Runnable (stato iniziale), Active (quando uno dei suoi metodi è in esecuzione), Hold (in attesa che un'altra Attività liberi risorse), Event-blocked (in attesa della risposta ad un messaggio asincrono), Activity-blocked (in attesa del completamento di un'Attività-figlia) e Done (terminata). Il modello descrive inoltre le relazioni che possono intercorrere tra due Attività, quali sono le possibili transizioni tra gli stati sopra elencati; ed il modello di concorrenza che controlla l'esecuzione delle Attività.

Un'Attività assomiglia molto ad un Task di FIPA-OS e, in misura minore, a un Behaviour di JADE (i Behaviour non sono event-driven), ma il modello utilizzato da OpenCybele è molto più ricco di quello degli altri due framework.

Descrizione della libreria:

Il modello di programmazione di OpenCybele è molto diverso da quelli degli altri framework esaminati: in tutti gli altri, generalmente, l'utente deve scrivere classi che estendono per ereditarietà un insieme di classi base definite nella libreria. OpenCybele adotta un approccio basato su *metodi di classe* e *classi delegate*, inoltre fa uso intensivo della *reflection*.

- Che cos'è un agente, come viene definito dal programmatore? Qual è il meccanismo che consente di creare gli agenti?

  Le classi che rappresentano gli agenti e le attività non sono direttamente istanziabili dall'utente, che può operare soltanto attraverso i *metodi di classe* (ovvero *statici*) delle classi `Cybele`, `Agent` e `Activity`. Ad esempio, l'ambiente runtime di OpenCybele si avvia invocando `Cybele.startUp()`. L'utente scrive alcune classi, chiamate *classi delegate*, che vengono associate a runtime ad agenti e attività, e i cui metodi vengono invocati quando si verificano determinati *eventi*. Per lanciare un agente è necessario invocare `Cybele.createAgent()`, a cui va passato il nome della classe delegata che rappresenta l'attività principale dell'agente. Come in JADE, se un agente crea un altro agente non riceve mai indietro un riferimento esplicito all'oggetto agente creato.

- Come è definito il comportamento degli agenti?

  Una classe delegata (che, come si è detto, rappresenta un'attività) deve implementare l'interfaccia `Handler`, un'interfaccia senza metodi: infatti i nomi dei metodi da invocare viene determinato a runtime per mezzo della reflection. Un agente crea le proprie attività-figlie invocando `Agent.createActivity()`. Una classe delegata, per poter funzionare correttamente, deve associare (all'interno del costruttore) i nomi dei propri metodi pubblici agli eventi a cui è interessata; gli eventi possono essere costituiti dallo scadere di timer o da messaggi ricevuti.

- In che modo comunicano gli agenti?

  La comunicazione tra agenti (ma sarebbe meglio dire tra attività) può essere punto-punto o multicast, sincrona o asincrona. La tecnica di comunicazione più semplice da utilizzare è quella multicast, basta sul modello publish&subscribe: un'attività, nel suo costruttore (e, se lo desidera, anche nei propri metodi) dichiara di voler ricevere tutti gli eventi-messaggi legati ad un particolare identificatore (una stringa chiamata *tag*) invocando `Activity.openChannel()`, a cui passa, oltre al tag, il nome di un proprio metodo pubblico. Tale metodo verrà invocato ogni volta che un altro agente (o meglio un'altra attività) invia un messaggio multicast associato a quella stessa tag (i messaggi sono inviati invocando `Activity.sendAll()` o `Activity.sendAllBlock()`).

  La comunicazione punto-punto è più complessa da utilizzare, e richiede l'uso di particolari oggetti chiamati `ChannelHandler`, che rappresentano canali di comunicazione diretta tra due attività.

  OpenCybele non ha agenti coordinatori, ovvero non esistono i concetti di nameserver o facilitator. A livello di trasporto, non è chiaro quale sia il protocollo utilizzato dal framework.

- Qual è il formato dei messaggi scambiati?

  Un messaggio in OpenCybele è rappresentato semplicemente da un array di oggetti `Serializable`. Non vi è alcun supporto per specifici formati di messaggi, né per content languages.

Supporto:

È disponibile soltanto un indirizzo e-mail, oltre ad un semplice sistema di bug report. Nel complesso, non presenta un supporto efficiente come quello di altri framework.

Commenti:

L'unico punto di forza di OpenCybele è la sua libreria, che risulta essere la più stabile ed efficiente tra quelle prese in esame, oltre alla più semplice da utilizzare (il codice degli agenti implementati con OpenCybele risulta molto chiaro e sintetico, al contrario di ciò che avviene praticamente in tutti gli altri framework). Un altro vantaggio è la gestione molto semplice del multicast, grazie al modello publish&subscribe; inoltre gli "eventi" possono essere generati anche da programmi che non sono agenti (mentre per gli altri framework ciò non è sempre possibile). I suoi difetti sono le scarse funzionalità aggiuntive e la non conformità allo standard FIPA.

4.3.6 Zeus

Introduzione:

Zeus 1.2.1 è stato sviluppato da BT-Lab (British Telecom Laboratories) ed è il framework più complesso tra quelli presi in esame. L'ultima versione è del 23 maggio 2001. La versione di Java su cui si basa è 1.2.

Caratteristiche del framework:

Zeus appare come lo strumento più completo per lo sviluppo ad agenti, in quanto copre esplicitamente tutte e quattro le fasi viste nell'introduzione del capitolo. Per l'analisi usa la metodologia "Role Modelling" (modellamento dei ruoli), basata su diagrammi simili a quelli UML (tuttavia per questa fase Zeus non fornisce alcuno strumento grafico). Per il progetto Zeus possiede un ambiente visuale dove è possibile definire l'ontologia, gli agenti, i task che essi sono in grado di eseguire, e molte altre caratteristiche. Per l'implementazione, Zeus genera automaticamente buona parte del codice necessario e, come tutti gli altri framework, fornisce una libreria di classi Java per sviluppare gli agenti. Per la fase

di deployment il framework presenta numerosi strumenti per la visualizzazione e il controllo in run-time degli agenti.

La documentazione è molto buona; inoltre sono disponibili dei tutorial. Tuttavia, la documentazione della libreria di classi è molto scarsa (è presente un Javadoc con poche informazioni). Zeus è FIPA-compliant; inoltre supporta il "*Progetto JAS*".

Descrizione della libreria:

Come si è detto nel paragrafo precedente, la libreria di Zeus (Zeus Agent Component Library) è complessa da comprendere e poco documentata, quindi la descrizione che ne verrà data non sarà dettagliata come quella vista per gli altri framework.

Con gli strumenti di sviluppo visuale è possibile definire un'ontologia (una gerarchia di "fatti", che costituiscono le conoscenze degli agenti), i task che gli agenti sono in grado di eseguire, e altre caratteristiche. Gli agenti sono "goal-driven" e il comportamento standard di un agente è definito da un grafo orientato, cioè una macchina a stati, la cui esecuzione è guidata da un "coordination engine". Il modello di comportamento rule-based non è supportato.

Gli agenti comunicano tra loro direttamente, e apprendono i rispettivi indirizzi tramite un agente *nameserver*. Gli agenti espongono le loro funzionalità agli altri agenti (grazie ad un agente *facilitator*) e la libreria fornisce diverse tecniche per far sì che gli agenti contrattino tra loro i task da eseguire. Tuttavia sembra che Zeus non fornisca supporto per tutti i protocolli FIPA. Un agente può decidere quali messaggi ricevere in base al contenuto, grazie all'uso delle "regular expressions".

I linguaggi per la comunicazione tra agenti supportati sono FIPA-ACL e KQML.

Normalmente non supporta l'esecuzione di più agenti nella stessa Virtual Machine, ma nelle ultime versioni è stato aggiunto un tool runtime (ZSH) per supportarla.

Supporto:

Zeus non presenta un vero e proprio supporto ufficiale. È disponibile una mailing list di tutti gli utenti (compresi i creatori del framework stesso) presso la quale si possono riportare bug o chiedere informazioni, ma non sempre si riceve risposta.

Commenti:

L'impostazione teorica di Zeus è molto interessante, soprattutto per quanto riguarda la particolare attenzione rivolta alla collaborazione tra agenti (*collaborative agents*); inoltre l'infrastruttura di deployment è la più avanzata tra quelle esaminate. Tuttavia l'implementazione del più semplice degli esempi si è rivelata difficoltosa, e per questo ed altri motivi (si veda §4.4) si è ritenuto più opportuno orientarsi verso framework più semplici da utilizzare e meglio supportati.

4.3.7 Altri framework

In questa sezione sono elencati in breve alcuni altri framework ad agenti che non sono stati esaminati approfonditamente, ma che presentano comunque alcune caratteristiche degne di nota.

Aglets

Sviluppata da IBM, l'Aglets Software Development Kit è una delle più diffuse librerie per lo sviluppo di *agenti mobili*, cioè in grado di migrare da una macchina all'altra durante la loro stessa esecuzione. Utilizza lo standard FIPA-ACL per la comunicazione tra gli agenti, ed è sviluppata interamente in Java 1.1.

Cormas

Questo framework è basato su VisualWorks, un ambiente di sviluppo che utilizza il linguaggio Smalltalk. Cormas sta per "Common-pool Resources and Multi-Agent Systems". A differenza di tutti gli altri strumenti visti finora, è dedicato allo sviluppo di sistemi di simulazione, in particolare per quanto riguarda la gestione di risorse naturali. Gli agenti "vivono" in un ambiente che simula un mondo fisico, costituito di "entità spaziali" (che sono a loro volta agenti) e di "risorse" che possono essere rinnovabili o no.

JATLite

JATLite 0.4 beta è una libreria di classi sviluppata dall'Università di Stanford, e deriva da una semplificazione di un progetto precedente, denominato JAT (Java Agent Template). L'ultima versione disponibile è del 1997 (da allora non sono stati rilasciati aggiornamenti), non possiede alcun ambiente grafico e utilizza il KQML (versione del 1993) come linguaggio di comunicazione tra gli agenti.

È particolarmente indicato per lo sviluppo di sistemi che coinvolgano *agenti-applet*: gli agenti non comunicano mai direttamente, ma solo attraverso un "router" centrale, risolvendo così i problemi causati dalle restrizioni imposte alle applet[17]. Il framework è completamente basato su Java 1.1 proprio per garantire la compatibilità con i browser (che normalmente non supportano Java 2 senza un plug-in).

Un'interessante estensione di JATLite è JATLiteBean: si tratta di un componente JavaBean che incapsula tutte le funzionalità di JATLite semplificando molto la programmazione, inoltre può essere utilizzato in ambienti visuali.

[17] Le *applet* sono applicazioni Java che vengono scaricate da un server ed eseguite dal client all'interno di un browser Web. Un'applet Java può instaurare connessioni di rete soltanto con il server da cui è stata scaricata.

MadKit

Si tratta di un framework sviluppato presso l'Università di Montpellier. Gli agenti sono definiti come estensione di una classe Agent astratta, e vengono eseguiti dal micro-kernel di MadKit come thread separati. Tuttavia, a parte questo, il framework non fornisce servizi di gestione dei thread, degli eventi e della concorrenza; inoltre gli agenti sono essi stessi responsabili della lettura dei messaggi presenti nella propria mailbox tramite un meccanismo di polling. MadKit non adotta nessuna specifica architettura ad agenti, il che può essere considerato un pregio o un difetto, a seconda dello specifico problema da affrontare. È basato sulla versione 1.1 di Java.

MASSYVE-Kit

Si tratta di uno dei pochi framework ad agenti che utilizza il linguaggio C++. È interessante in quanto fa parte di un progetto (condotto dalla Federal University of Santa Catarina e dal gruppo GSigma) che si occupa soprattutto dei sistemi manifatturieri: MASSYVE infatti sta per "Multiagent Agile Manufacturing Scheduling Systems for Virtual Enterprise". Possiede un'interfaccia grafica per guidare l'utente durante lo sviluppo, e fornisce un meccanismo per lanciare gli agenti sia localmente che in remoto. Funziona soltanto in ambiente Windows.

TeamBot

Un framework ad agenti esplicitamente dedicato allo sviluppo di applicazioni robotiche: gli agenti simulano il comportamento di robot mobili, e inoltre il codice sviluppato con TeamBot può essere eventualmente integrato con sistemi robotici "fisici".

4.4 Confronti tra framework

La tabella riportata nella pagina seguente riassume e mette a confronto le principali caratteristiche dei framework ad agenti esaminati. Nelle prime tre righe viene indicato se il framework possiede strumenti di sviluppo o runtime, insieme ad un giudizio sulla libreria di classi (sono le tre caratteristiche fondamentali, come si è detto in §4.2.2). Le altre righe riportano altre caratteristiche non essenziali ma comunque importanti relative ai vari strumenti.

AgentTool e **JiVE** sono framework interessanti in quanto veri e propri strumenti CASE che guidano lo sviluppatore attraverso le varie fasi di realizzazione di un sistema. Dal punto di vista dell'analisi e del progetto, AgentTool è senza dubbio il migliore: MaSE è una metodologia potente ed intuitiva al tempo stesso, i diagrammi sono numerosi e semplici da costruire e modificare; al contrario, JiVE offre una metodologia meno completa (quella di JAFMAS) e soltanto due tipi di diagrammi. Il punto di forza di JiVE, la funzionalità che lo rende unico, è la sua architettura collaborativa – tuttavia, per gli obiettivi di questa tesi, tale caratteristica non è di alcuna utilità.

Come si è già detto, il difetto di entrambi è lo scarso supporto per le fasi di implementazione (l'eccessiva limitatezza delle librerie su cui si appoggiano, ovvero AgentMom e JAFMAS) e di deployment (totale assenza di un ambiente di esecuzione e di strumenti per il debugging).

Quadro riassuntivo delle caratteristiche dei framework esaminati

	AgentTool	FIPA-OS	JADE	JiVE	OpenCybele	Zeus
Caratteristiche principali						
Strumenti di sviluppo	Si	In parte	No	Si	No	Si
Libreria	Molto limitata	Buona	Buona	Limitata	Medio-Buona	Buona
Strumenti runtime	No	Si	Si	No	No	Si
Altre caratteristiche						
Versione di Java	1.3	1.2	1.2	1.2 / 1.1	1.3	1.2
FIPA-compliant	No	Si	Si	In parte	No	Si
Metodologia	MaSE	Nessuna	Nessuna	JAFMAS	ACP	Role-Modelling
Documentazione	Medio-Buona	Buona	Buona	Medio-Buona	Buona	Medio-Buona
Agenti coordinatori	No	AMS, DF, ACC	AMS, DF, ACC	No	No	AMS, DF, ACC
Tecnica di comunic.	Diretta	Diretta	Diretta	Diretta/P&S	Diretta/P&S	Diretta
Linguaggio comunic.	Proprietario	FIPA-ACL	FIPA-ACL	FIPA-ACL / KQML	Oggetti Java	FIPA-ACL / KQML
Multi-agente in una JVM	No	Si	Si	No	Si	Solo con ZSH

Zeus, in base alla documentazione fornita, dovrebbe essere il framework più completo da tutti i punti di vista. Tra i suoi punti di forza c'è senza dubbio il supporto per tutte le fasi del ciclo di vita (anche se, per essere precisi, la fase di analisi è puramente teorica e cartacea, senza tools) e la presenza di potenti strumenti di monitoraggio e controllo dell'esecuzione degli agenti.

Nonostante tutte queste premesse positive, Zeus si è rivelato molto più difficile da utilizzare rispetto agli altri framework, e la fase di test ha dato non pochi problemi, anche a causa di errori nei tutorial forniti. Altri svantaggi di Zeus sono la scarsa efficienza del supporto on-line e la documentazione delle API, che non è dettagliata come negli altri framework. Inoltre Zeus dichiara di essere conforme allo standard FIPA, ma un'analisi più approfondita rivela che lo è solo in parte.

A causa di tutti questi fattori si è deciso di orientarsi verso altri framework.

OpenCybele, **FIPA-OS** e **JADE** sono tre piattaforme molto simili tra loro: la scelta di quale utilizzare non è stata semplice.

Tutti e tre i framework si interessano molto poco delle fasi di analisi e progetto, e gli strumenti grafici non sono equiparabili a quelli di Zeus, JiVE e AgentTool. Tuttavia tutti e tre supportano la fase di implementazione meglio degli altri: presentano infatti ottime librerie di classi, ben documentate e semplici da usare.

Per quanto riguarda la fase di deployment, soltanto FIPA-OS e JADE la supportano, mentre l'ambiente di esecuzione di OpenCybele ha funzionalità limitate e non possiede strumenti di debugging.

Il test delle tre piattaforme non ha evidenziato particolari problemi, quindi la scelta è stata determinata dalle funzionalità.

Molte sono le caratteristiche comuni ai tre framework. Per prima cosa, tutti e tre scompongono il comportamento di un agente in entità separate che, come si è già visto, sono chiamate *Attività* (*Activities*) in OpenCybele, *Task* in FIPA-OS e *Comportamenti* (*Behaviours*) in JADE, anche se tali nomi si riferiscono in effetti ad oggetti molto simili. Le attività (e così anche i task e i behaviour) di un agente vengono eseguite contemporaneamente, nascondendo allo sviluppatore la gestione della concorrenza, e sollevandolo quindi dall'onere di occuparsi esplicitamente del multi-threading. Le attività generano se necessario attività figlie con cui possono condividere dati; solitamente vengono usate per definire i vari ruoli di un agente, o per portare avanti diverse conversazioni contemporaneamente.

Tutte e tre le piattaforme supportano molto bene la creazione di più agenti nella stessa Virtual Machine.

Tra i vantaggi di OpenCybele troviamo indubbiamente la sua semplicità d'uso: il codice scritto per tale libreria è in genere estremamente "pulito" e chiaro da comprendere. Un'altra caratteristica positiva è la gestione dei messaggi multicast, molto migliore rispetto a quella degli altri framework. Tuttavia OpenCybele, a differenza di tutte le altre piattaforme prese in esame, è un framework di pura implementazione, e non offre alcuno strumento per l'analisi/progetto e soprattutto per il deployment. La libreria è sì semplice e funzionale, ma molto limitata, mentre quelle di FIPA-OS e JADE offrono un maggior ventaglio di funzionalità.

FIPA-OS e **JADE** hanno in comune la conformità allo standard FIPA: entrambi implementano tutti i componenti definiti come "obbligatori" (*normative*) da FIPA, come ad esempio i "platform agents" (AMS, DF) e il canale di comunicazione (ACC); inoltre supportano il linguaggio di comunicazione FIPA-ACL.

Per la fase di progetto, JADE non presenta alcun tool; FIPA-OS ha il TaskGenerator, uno strumento molto efficiente che genera automaticamente il codice per l'implementazione di un protocollo d'interazione. Anche se questo tool

è limitato ad un compito ben preciso, è comunque un vantaggio rispetto a JADE, che fornisce un supporto molto ridotto per implementare protocolli.

Le due librerie forniscono servizi e funzionalità sostanzialmente simili. Nessuno dei due framework adotta un modello concettuale per la definizione del comportamento, ma entrambi forniscono un accesso semplificato al sistema esperto JESS (che però non è incluso in nessuno dei due framework e deve essere installato separatamente). JADE ha in più alcune classi per la definizione di un'*ontologia*, mentre da questo punto di vista FIPA-OS non fornisce alcun supporto.

Una differenza sostanziale è che FIPA-OS (come del resto anche OpenCybele) usa il meccanismo della reflection per selezionare i metodi da invocare: questo, pur comportando una piccola perdita di flessibilità, rende il codice molto più semplice da gestire, eliminando molte delle criptiche sequenze di "if" o "switch" necessarie in JADE ed in altri framework. Inoltre FIPA-OS è in grado di verificare automaticamente se una data conversazione segue o no il proprio protocollo (scelto tra i protocolli FIPA o definito dall'utente): questo controllo automatico è utilissimo per rivelare errori nell'implementazione.

Un altro punto a favore di FIPA-OS è il suo supporto integrato per gestire dati espressi in linguaggio XML (che possono essere incorporati all'interno di messaggi FIPA-ACL), chiamato "XML data binding". Inoltre FIPA-OS supporta un maggior numero di *content languages* rispetto a JADE.

Dal punto di vista del deployment, entrambi i framework presentano diversi strumenti, anche se quelli di JADE sono più avanzati: JADE, ad esempio, è in grado di monitorare lo stato di tutti gli agenti del sistema, anche di quelli in esecuzione su computer remoti, mentre FIPA-OS si limita a quelli eseguiti nella stessa Virtual Machine. JADE possiede inoltre lo strumento Sniffer che rende molto semplice seguire le conversazioni che avvengono tra gli agenti, tramite l'uso dei *sequence diagrams*. FIPA-OS invece fornisce un sistema di registrazione

degli eventi molto flessibile (ma poco user-friendly), che permette di creare un file di *log* a diversi livelli di dettaglio.

In sintesi, la competizione si è ristretta a questi due framework e per questa tesi si è scelto, anche per sperimentare una soluzione diversa da altre tesi del laboratorio Lido, di utilizzare FIPA-OS, rimandando ad ulteriori sviluppi della ricerca sullo scheduling ad agenti la possibilità di testare JADE.

Da tutte le considerazioni fatte dovrebbe risultare chiaro che non esiste un framework migliore in assoluto: FIPA-OS è stato scelto perché realizza un buon compromesso tra vantaggi e svantaggi, e per l'attrattività di molte delle sue funzionalità.

4.5 Collegamenti

In questa sezione sono raccolti gli indirizzi Internet presso i quali è possibile scaricare i framework esaminati e reperire tutte le informazioni necessarie.

AgentTool:	http://www.cis.ksu.edu/~sdeloach/ai/home.html sdeloach@cis.ksu.edu
FIPA-OS:	http://fipa-os.sourceforge.net/ fipa-os-developers@lists.sourceforge.net
JADE:	http://jade.cselt.it/ jade-develop@sharon.cselt.it
JiVE:	http://www.ececs.uc.edu/~abaker/JiVE/ agalan@ececs.uc.edu
OpenCybele:	http://www.opencybele.org/ admin@OpenCybele.org
Zeus:	http://www.labs.bt.com/projects/agents/zeus/ simon.2.thompson@bt.com

4.5.1 Tools per lo sviluppo di MAS:

http://www.agentbuilder.com/AgentTools/

http://www.multiagent.com/Software/

4.5.2 Altri indirizzi:

Aglets: http://www.trl.ibm.com/aglets/index_e.htm

Cormas: http://cormas.cirad.fr/indexeng.htm

JATLite: http://java.stanford.edu/

JATLiteBean: http://kmi.open.ac.uk/people/emanuela/JATLiteBean/

JAS: http://www.java-agent.org/

JESS: http://herzberg.ca.sandia.gov/jess/

MADGS: http://excalibur.brc.uconn.edu/~madgs/

MASSYVE-Kit http://www.gsigma-grucon.ufsc.br/massyve/mkit.htm

Capitolo 5: Il sistema multi-agente per lo scheduling on-line

5.1 Applicazioni dei sistemi multi-agente allo scheduling in ambito manifatturiero

La filosofia dei sistemi ad agenti può essere adottata per compiti di gestione, può indicare la strada per sviluppare nuovi strumenti software, ma può anche essere un metodo da sperimentare per modellare processi decisionali.

Come si è già detto nel Capitolo 1, questa tesi si propone di affrontare il problema dello scheduling applicando un'euristica basata sugli agenti.

Sviluppare un'euristica ad agenti per lo scheduling in modo generico è impossibile: è infatti necessario definire lo specifico problema preso in esame, poiché le sue particolari caratteristiche possono certamente influenzare sia la natura degli agenti che il loro comportamento. Ad esempio, la presenza di tempi di rilascio può implicare vincoli sulla "nascita" degli agenti; la presenza di tool (ovvero di risorse aggiuntive) per eseguire lavorazioni può implicare la possibilità di introdurre agenti associati a tali tool e così via.

In questa sezione, tuttavia, verrà discusso un modello generale – o meglio, i possibili modelli alternativi – con cui la filosofia degli agenti può essere usata in un contesto di scheduling come un nuovo paradigma euristico.

Per quanto si conosce delle caratteristiche degli agenti, si ritiene che questi possano risultare particolarmente utili in presenza di dinamicità, ovvero di problemi con una struttura non completamente deterministica o, analogamente, a decisioni in condizioni di incertezza; tra l'altro questi sono i casi più frequenti in contesti manifatturieri dove gli ordini possono arrivare in qualsiasi momento ed eventi imprevisti possono modificare le caratteristiche del problema (tempi diversi da quelli noti a priori, presenza o assenza di risorse, ritiro di ordini, ecc.). In funzione di quanto si voglia (o si *debba*) garantire in termini di flessibilità in un'azienda produttiva si possono adottare modelli più o meno semplificati per prendere decisioni riguardo lo scheduling ed il suo successivo e continuo adattamento in fase di esecuzione. L'approccio comune è quello di fissare delle decisioni (*schedule*) a priori, quindi di adottare delle opportune politiche di controllo in linea per rispondere a variazioni impreviste. Se tuttavia la regola è che eventi imprevisti, pur restando tali, tendono a verificarsi comunemente, il precedente modo di operare non sembra essere il più efficiente, ovvero la fase di scheduling a priori sembra un momento poco rilevante.

A questo punto è necessario fare una considerazione importante: gli "eventi imprevisti" di cui sopra possono essere di due tipi:

- arrivo di nuovi job non previsti
- disturbi (ad es. guasti).

Il primo caso corrisponde a quello che è stato definito *scheduling on-line*, ossia non si conoscono le caratteristiche dei job da schedulare prima che essi arrivino (distinto da *off-line*, in cui tutti i job sono noti a priori). Nel seguito verrà sviluppato un sistema di schedulazione per problemi on-line, anche se è nostra opinione che possa essere facilmente adattato per trattare situazioni in cui si verificano disturbi, anche se in questa fase tale possibilità non è stata ancora sperimentata.

5.2 Due possibili approcci

Volendo applicare la filosofia ad agenti allo scheduling, le prime domande che ci si deve porre sono:

- a quali entità associare gli agenti
- quale tipo di architettura adottare

Di nuovo qui è abbastanza intuitivo comprendere come queste scelte dipendano dal tipo di problema di scheduling che deve essere affrontato.

Sempre cercando di restare piuttosto generali è possibile distinguere tra due scelte alternative che possono essere considerate due punti estremi tra i quali collocare modelli per così dire ibridi. Queste due scelte corrispondono ad un approccio che potremmo definire "fisico" ed uno invece "funzionale":

1) L'*approccio fisico* associa gli agenti agli elementi presenti nel problema. Quindi gli agenti possono essere associati a job, task, macchine, risorse, tool, o componenti del sistema di trasporto (AGV).

2) L'*approccio funzionale* associa gli agenti alle funzioni che devono essere svolte.

Mentre il primo approccio può risultare, per così dire, più naturale o direttamente comprensibile, il secondo potrebbe nascondere delle insidie. Anche se nella tesi verrà sviluppato un sistema basato su di un approccio di tipo fisico, a questo punto sembrano necessarie alcune considerazioni di tipo generale sull'approccio funzionale.

5.2.1 Possibili approcci funzionali

Nel caso di problemi di scheduling si potrebbe osservare che esiste in realtà una sola funzione da realizzare, ovvero *la funzione di scheduling*. Le funzioni intese come attività svolte dal sistema produttivo e di trasporto sono

sostanzialmente associate ad elementi fisici del sistema stesso o eventualmente a loro aggregazioni. Se si intendesse quindi per approccio funzionale associare agenti a questo tipo di funzione si ricadrebbe nuovamente in un approccio fisico. Dire però che si adotta un approccio funzionale non significa nemmeno che debba essere presente un solo agente responsabile della funzione di scheduling: se così fosse l'agente, in un visione antropomorfa, svolgerebbe le mansioni dell'operatore scheduler e ne risulterebbe un approccio centralizzato allo scheduling in cui la presenza degli agenti sarebbe sostanzialmente irrilevante rispetto all'algoritmo decisionale utilizzato. In questo caso l'agente comunicherebbe con altre entità al di fuori dell'ambiente in cui viene generato lo schedule (ad esempio con l'utente o il sistema gestionale ERP) e percepirebbe dallo shop floor (attraverso i suoi sensori) il verificarsi di variazioni di stato. Lo schedule farebbe parte dei suoi dati privati e gli algoritmi o euristiche di scheduling sarebbero contenuti nella sua "conoscenza" (*knowledge*). Niente di più quindi che un "tradizionale" approccio allo scheduling incapsulato in un sistema (esterno) ad agenti. In questo modo la presenza dell'agente non servirebbe a semplificare o ad affrontare in maniera alternativa la soluzione di un problema di scheduling.

Una diversa possibilità sta nell'assegnare a più agenti cooperanti o in competizione (*Coop-Comp Agents*) il compito di determinare una parte dello schedule. Anche questa volta è facile muoversi nella direzione di un modello fisico se questa associazione si basa su elementi fisici per la sua definizione; se gli agenti fossero responsabili di determinare lo schedule parziale delle macchine, ad esempio, ciò dovrebbe risultare subito evidente. Un'alternativa immediata è assegnare a ciascun agente il compito di determinare lo schedule di un job.

Rispetto all'approccio a singolo agente (che, come si è detto, non corrisponde ad affrontare lo scheduling per mezzo degli agenti) le caratteristiche tipiche dei sistemi ad agenti (autonomia, proattività, ecc.) verrebbero sfruttate (costituendo così un'euristica di scheduling) durante l'attività cooperativa o competitiva degli agenti stessi: gli agenti dovrebbero comunicare per ottenere collaborazione da altri al fine di poter completare il loro compito (ad esempio un agente macchina potrebbe richiedere di eseguire un certo task ad un'altro agente

macchina in modo da poter avanzare nel proprio schedule; oppure un agente job potrebbe chiedere ad un altro agente job di liberare una risorsa). In questi casi i problemi modellistici corrispondono a 1) definire la strategia di comportamento sociale degli agenti; e 2) introdurre in tale strategia un meccanismo per tener conto del fine collettivo (globale), corrispondente all'ottimizzazione dell'intero schedule o introdurre agenti mediatori/supervisori responsabili della fusione degli schedule parziali e capaci di influenzare gli agenti scheduler (stimolare o reprimere, annullare decisioni prese dagli agenti scheduler o sbloccare situazioni).

Questo scenario può avvicinarsi ulteriormente all'approccio fisico (o arrivare a "metà strada" rispetto a questo) facendo coesistere agenti entrambi responsabili delle stesse decisioni di scheduling, ovvero agenti macchine ed agenti job. Chiaramente il meccanismo di Coop-Comp si complica in questo caso: gli agenti non sono più depositari (nei loro dati privati) di uno schedule parziale (cosa che, si noti bene, caratterizza ciò che si intende in questo contesto per approccio funzionale), ma di una "proposta" di schedule che dovrà essere eletta a schedule parziale e introdotta nello schedule globale ad esempio facendo ricorso agli agenti mediatori/supervisori responsabili dei dati relativi allo schedule.

Una visione funzionale alternativa può consistere nel legare il tempo, ovvero un orizzonte temporale, ad un agente responsabile della definizione dello schedule parziale in tale intervallo. In questo caso il sistema ad agenti sarebbe costituito da agenti che operano su intervalli che si sovrappongono parzialmente e che, ovviamente, si influenzano. Un agente scheduler percepirà lo schedule parziale presente all'inizio del proprio intervallo di influenza (schedule fissato da un altro agente) e sarà responsabile delle decisioni nel proprio intervallo. Tale agente leggerà i dati relativi ai task eseguibili ed alle macchine disponibili, dichiarerà i task e le macchine di cui assumerà la responsabilità in quell'intervallo in modo da poter essere avvertito in caso di possibili variazioni (un task non è più eseguibile perché altri agenti scheduler in intervalli precedenti hanno modificato alcune decisioni, una macchina non è più completamente disponibile, ecc.). La comunicazione di questi eventi mette in evidenza la necessità di un agente supervisore responsabile di una visione globale dello schedule; inoltre, tale agente

supervisore – poiché gioca un ruolo di interfaccia del sistema di scheduling con l'utente, lo shop floor ed il sistema gestionale – rende evidente come questo approccio possa essere adottato anche in linea e considerato un sistema di scheduling-rescheduling perennemente attivo. Gli agenti in tale sistema nascono (con un certo anticipo rispetto al verificarsi di un certo intervallo; il supervisore può svolgere anche il ruolo di generatore dell'agente per un intervallo), maturano (prendono sempre più decisioni riguardo al loro intervallo, intensificando la loro attività sociale) e muoiono (quando l'intervallo è completamente trascorso nel tempo reale).

Anche per questo modello deve essere definita una strategia attraverso cui gli agenti agiscono autonomamente e cooperativamente. Comportamenti alternativi per gli agenti potrebbero essere i seguenti:

(a) comportamento autonomo. Quali goal guidano la scelta di un agente scheduler? E' evidente che se tra i task che tale agente dovrebbe/potrebbe schedulare[18] ne esistono alcuni finali per un job (corrispondenti al completamento di una commessa) lo scopo dell'agente rispetto a tali task è evidente (ad esempio, rispettare le due date della commessa). Nel caso di task attivabili di tipo "intermedio", l' agente potrebbe cercare di assegnare i task in modo da ridurre il WIP (*Work In Process*, vedi §2.4) seguendo una filosofia JIT cercando un compromesso tra anticipare o ritardare la loro esecuzione. In un sistema di rescheduling operante in tempo reale, le decisioni di un agente potrebbero tener conto di informazioni provenienti da altre entità del sistema gestionale che, ad esempio, forniscono informazioni sullo stato delle materie prime e sulla loro disponibilità futura;

(b) comportamento cooperativo: gli agenti per svolgere proficuamente il loro compito (una misura dell'efficacia della loro azione può essere valutata tenendo conto dei job che si potevano concludere e di come in realtà si sono conclusi, dei task attivabili, ecc.) possono comunicare con altri agenti, non solo

[18] Si è aggiunto "potrebbe" perché un task attivabile può a causa della presenza di risorse limitate non essere assegnato, ovvero eseguito, nell'intervallo di competenze di un agente ma lasciato in eredità all'agente seguente.

quelli “temporalmente adiacenti”, ma in generale tutti gli altri. Questo è possibile permettendo agli agenti di conoscere da quali altri agenti sono stati schedulati task predecessori dei “propri” task e quali altri agenti ovvero quali altri task in intervalli futuri potrebbero sentire le conseguenze delle loro azioni. La comunicazione può consistere in richieste di attuare modifiche allo schedule (nel passato) e per ottenere il permesso di eseguire modifiche (dal futuro).

In questo modo di cooperare il tempo giocherebbe un ruolo sia fittizio che reale:

- **fittizio**, perché rispetto ad un singolo agente, dal momento della sua nascita al momento della sua morte, esiste una sorta di presente esteso costituito da tutti gli intervalli associati agli agenti attivi; l’estremo inferiore di questo presente si avvicina progressivamente con l’avanzare del tempo nella realtà;

- **reale**, perché la frenetica attività degli agenti viene scandita progressivamente mandando effettivamente in esecuzione i task e congelando così le decisioni.

Tutti questi approcci sono molto interessanti, tuttavia non verranno trattati in questa tesi, ma lasciati come possibili sviluppi futuri.

Nel seguito verrà portato avanti un approccio di tipo fisico, in quanto questo sembra più intuitivo per iniziare una definizione più dettagliata di un sistema di scheduling basato su euristiche ad agenti, ovvero per affrontare il problema della definizione delle componenti interne dei diversi tipi di agente e del loro comportamento.

5.3 Formalizzazione del problema

Come si è detto in §5.2, parlare di scheduling e agenti a livello generale può risultare troppo vago. Da questo punto in avanti, dunque, si considererà un problema specifico e verrà sviluppato un sistema in grado di risolverlo.

Si consideri il seguente problema di scheduling, che rappresenta l'estensione on-line di un problema "a macchine parallele" (la versione off-line di questo problema, pur presentando caratteristiche semplici, appartiene alla categoria dei problemi NP-hard).

È dato un insieme di job indipendenti, j= 1,2,..., che devono essere eseguiti su M macchine identiche parallele, i=1, ... M.

Per ogni job j sono definiti i seguenti dati caratteristici:

- **a_j, arrival time**: l'istante di arrivo del job nel sistema produttivo, ovvero l'istante in cui l'esistenza del job è nota allo shop-floor.
- **r_j, ready time**: l'istante in cui il job è pronto per essere eseguito (può non coincidere con a_j)
- **p_j, processing time**: tempo che occorre al job per essere eseguito.
- **d_j, due date**: data di consegna, ovvero l'istante di tempo in corrispondenza del quale il job è completato senza alcuna penalità.
- **we_j, earliness weight**: parametro utilizzato per "pesare" l'anticipo del job.
- **wt_j, tardiness weight**: parametro utilizzato per "pesare" il ritardo del job.

Dopo l'istante a_j i dati caratteristici del job diventano noti e sono costanti: in modo particolare, p_j dipende esclusivamente dal job e non dalla macchina su cui viene eseguito, essendo esse tutte identiche. Ogni job deve essere processato una volta sola da una qualsiasi delle M macchine presenti nello shop-floor. I due

parametri we_j e wt_j sono generalmente valori compresi tra 0 e 1 e dipendono dalla priorità del job.

Per ogni job j è inoltre possibile definire:

- **S_j, starting time**: istante in cui il job inizia effettivamente la propria esecuzione: è il dato principale che deve essere deciso dal sistema di scheduling.

- **C_j, completion time**: tempo di completamento, definito come segue:

$$C_j = S_j + p_j$$

- **E_j, earliness**: anticipo del job, definito come segue:

$$E_j = \max [0, d_j - S_j - p_j] = \max [0, d_j - C_j]$$

- **T_j, tardiness**: ritardo del job, definito come segue:

$$T_j = \max [0, S_j + p_j - d_j] = \max [0, C_j - d_j]$$

Le macchine sono sempre disponibili, possono eseguire solo un job alla volta e l'esecuzione di un job non può essere interrotta (assenza di preemption).

I job devono essere assegnati e sequenziati sulle macchine, e gli starting time devono essere fissati in modo da minimizzare una somma pesata dei valori di earliness e tardiness. Poiché il problema considerato, come si è detto nel Capitolo 1, è un problema di scheduling on-line (con i job non noti a priori), per definire una funzione di costo dobbiamo fare riferimento ad un insieme di job ben preciso e limitato.

Supponendo che N sia il numero dei job arrivati in un certo periodo di tempo (ad esempio una settimana lavorativa), e indicando con I la corrispondente "istanza off-line" di questo problema (tutti i job noti a priori, ovvero $a_j = 0 \ \forall j$), il costo complessivo di uno schedule S è dato da:

$$C(S,I)= \sum_{j=1,\dots N} (we_j \cdot E_j + wt_j \cdot T_j) \qquad \textbf{(0)}$$

Ovviamente S deve essere uno schedule ammissibile (*feasible*).

Le caratteristiche peculiari di questo problema di scheduling sono due: in primo luogo, la funzione obiettivo (*performance measure*) da minimizzare non è regolare, in quanto considera anche il costo dell'earliness oltre a quello della tardiness. In secondo luogo, i job non sono conosciuti a priori, ma arrivano ciascuno ad un diverso istante a_j, e solo da quell'istante in poi i dati caratteristici di un job diventano noti.

Nel prossimo capitolo verrà presentata una formulazione matematica a numeri interi della versione off-line di questo problema, utilizzata per le prove sperimentali.

5.4 Il sistema multi-agente sviluppato

5.4.1 Definizione delle classi di agenti

Poiché in questa tesi è stato sviluppato un approccio di tipo fisico, si è pensato di associare degli agenti ai job ed alle macchine. Nel sistema sviluppato (MASS, Multi-Agent Scheduling System) vengono introdotte le seguenti classi di agenti:

- un insieme di agenti job, con un JA_j (**Job Agent**) per ogni job j da schedulare;
- un insieme di agenti macchina, con un MA_i (**Machine Agent**) per ogni macchina i presente nel sistema;
- un agente generatore degli agenti job (JGA, **Job Generator Agent**);

- un agente generatore degli agenti macchina (MGA, **Machine Generator Agent**);
- un agente coordinatore del tempo reale (RCA, **Real-time Coordinator Agent**).

Oltre a questi agenti fondamentali, è possibile introdurre un ulteriore agente che ad esempio serva da interfaccia con il sistema informativo aziendale (ERP).

Job Agent

Agenti associati ai singoli job e generati da JGA. Ogni JA conosce i dati caratteristici del job j ad esso associato (j, a_j, r_j, p_j, d_j, we_j, wt_j), e possiede un *virtual budget* B che gli consente di richiedere un servizio agli agenti macchina "pagando" con denaro virtuale. I JA hanno come obiettivo (*goal*) primario quello di far processare il job ad una macchina minimizzando la funzione obiettivo data dalla somma pesata di earliness e tardiness e come goal secondario spendere il meno possibile del loro budget. Per far questo decidono di richiedere il servizio in base ad una politica che tiene conto di vari fattori, come il margine di tempo che resta rispetto all'istante ideale di inizio processing del job, il budget residuo, il comportamento caratteristico (calmo o ansioso). L'agente muore una volta che ha svolto il suo compito.

Machine Agent

Agenti associati alle singole macchine e generati da MGA. Ogni MA_i conosce nei propri dati privati lo schedule relativo alla macchina i, ovvero una lista degli intervalli di tempo (*slot*, caratterizzati da starting time e completion time) in cui la macchina è "occupata" (ad ogni slot è associato un job).

I MA hanno come goal primario servire quanti più job possibile e come goal secondario guadagnare il più possibile dalle offerte fatte dai job. Sono obbligati a rispondere a tutte le richieste (positivamente o negativamente) e il loro

comportamento innato impedisce di accettare indiscriminatamente richieste dai job: tali richieste vengono valutate in base a criteri descritti nel seguito.

Gli agenti macchina non muoiono a meno che non vi sia un evento esterno a costringerli.

Job Generator Agent

E' una parte dell'interfaccia tra il sistema gestionale e l'ambiente di scheduling. Riceve le informazioni sui job che devono essere processati via via che gli ordini vengono accettati e genera i corrispondenti agenti JA_j.

Machine Generator Agent

E' la seconda componente dell'interfaccia tra il sistema gestionale e lo scheduler ad agenti. Riceve dal sistema gestionale i dati relativi alle risorse dello shop floor e genera gli agenti macchina MA_i. Aggiorna l'insieme di tali agenti in seguito a notizie circa variazioni delle condizioni operative da parte del sistema gestionale.

Real-time Coordinator Agent

Gestisce l'avanzare del tempo simulato all'interno al sistema, chiamato anche *tempo fittizio*. Poiché il sistema descritto è on-line, RCA gestisce il tempo fittizio tenendo conto dello scorrere del tempo reale.

RCA scandisce la "vita" del sistema ad agenti dando inizio ai "cicli di contrattazione" (descritti nel prossimo paragrafo), e per ogni ciclo registra quali tra le conversazioni tra i JA e i MA si sono concluse con successo (assegnazioni di job a macchine); costruisce nei propri dati privati lo schedule, comunicando le decisioni fissate all'agente responsabile dell'interfaccia con il sistema gestionale.

Il diagramma seguente schematizza le diverse classi di agenti presenti nel sistema di scheduling. Lo schema è soltanto ad un livello logico, e non rappresenta tutte le possibili interazioni tra gli agenti.

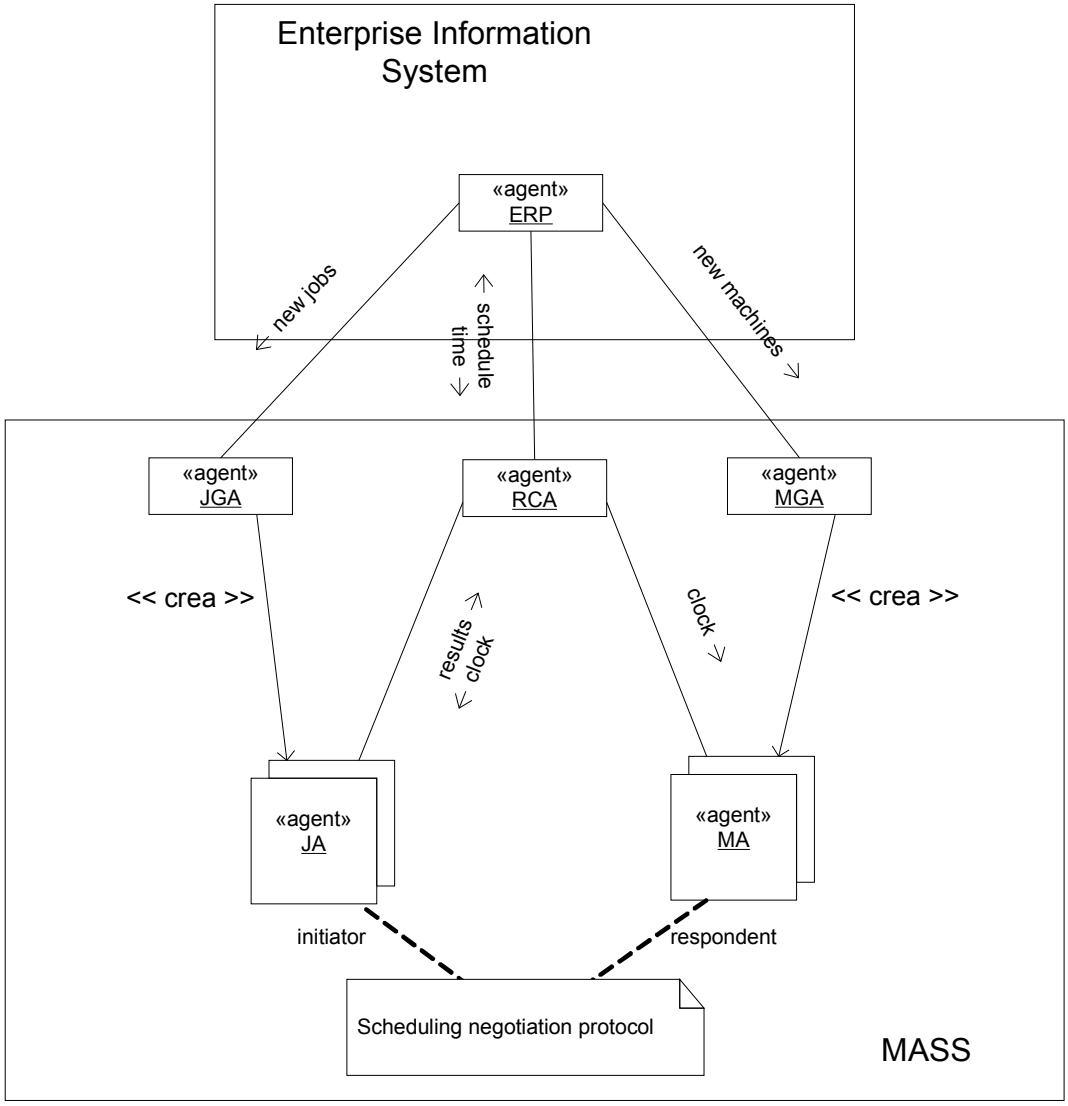


Figura 5-5.1 Il sistema di scheduling ad agenti

5.4.2 Funzionamento del MASS: i cicli di contrattazione

Gli agenti MGA e JGA sono sempre attivi, e generano gli agenti job e macchine quando necessario. In un contesto dinamico MGA opera raramente, ossia in corrispondenza all'introduzione di una nuova macchina, mentre JGA

molto più spesso o regolarmente, in corrispondenza dell'acquisizione di un nuovo ordine di lavoro.

Anche RCA è sempre attivo, in quanto è l'agente che ha il compito di coordinare l'intero sistema.

RCA dà inizio ad ogni ciclo di contrattazione inviando a tutti i JA e i MA un identico messaggio **Inform-clock**, che ha un duplice scopo:

- aggiornare il tempo fittizio (nel seguito chiamato τ) nell'intero sistema. Infatti gli agenti, sia JA che MA, non possono prendere decisioni se non conoscono il valore del tempo corrente;

- informare i JA che è cominciato un nuovo ciclo, e che sono autorizzati a iniziare contrattazioni con i MA.

Quindi RCA si "mette in ascolto" delle conversazioni che avvengono tra JA e MA: queste ultime seguono un ben preciso protocollo che sarà descritto nel prossimo paragrafo. Si noti che il significato di "mettersi in ascolto delle conversazioni" dipende dalla particolare implementazione scelta: nella maggior parte dei framework ad agenti disponibili, un agente non è solitamente in grado di "spiare" i dialoghi tra altri agenti. La possibilità qui ipotizzata è che i JA, alla fine di ogni ciclo, mandino a RCA un messaggio **Inform-results** in modo che il coordinatore sappia quali contrattazioni si sono concluse con successo e quali no.

Quando RCA vede che tutte le conversazioni si sono concluse, può dare inizio ad un nuovo ciclo facendo avanzare il tempo fittizio, in modo che non sia mai "in ritardo" rispetto al tempo reale. Inoltre, RCA nel far procedere il tempo tiene conto delle "esigenze" dei vari JA: in ogni messaggio **Inform-results** gli agenti job comunicano al RCA l'istante di tempo in cui vorrebbero ricevere il successivo messaggio **Inform-clock**.

5.4.3 Il protocollo di interazione tra job e macchine

Un protocollo di interazione è generalmente descritto come la sequenza dei messaggi scambiati da due agenti; in questo caso si tratta degli agenti job e macchine. Il protocollo è un'estensione del classico "Contract Net", ed è rappresentato nel seguente diagramma di protocollo AUML (vedi §3.6.3).

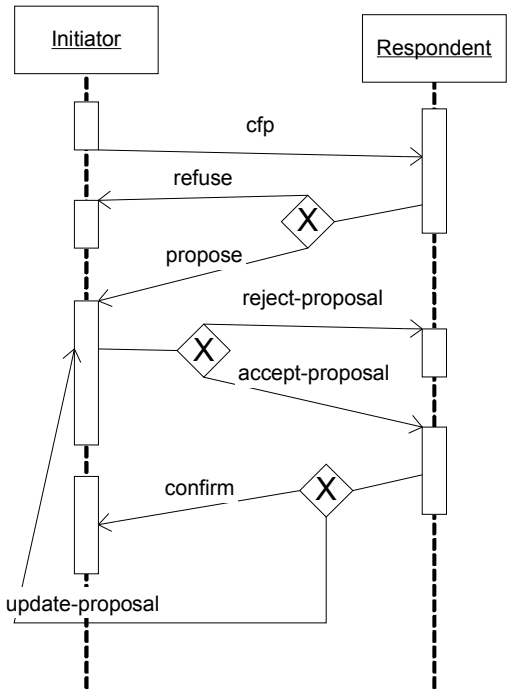


Figura 5-5.2 Protocollo di interazione tra agenti JA e MA

In molti modelli ad agenti, le contrattazioni avvengono per mezzo di uno o più agenti-broker: in questo scenario, per semplicità, gli agenti contrattano tra di loro direttamente, senza intermediari.

L'agente JA ha il ruolo dell'*Initiator*, infatti dà inizio ad una conversazione inviando un atto comunicativo **Cfp** (call for proposals) all'agente MA, il quale ha il ruolo del *Participant* (o *Respondent*). Il messaggio costituisce una richiesta di servizio, e contiene:

- i dati del job (j, a_j, r_j, p_j, d_j, we_j, wt_j);

- lo starting time s_j desiderato;
- un'offerta in denaro b_j (*bid*)

Il MA esamina la richiesta e può decidere se servirla o rifiutarla: nel caso la richiesta non venga accettata, il MA spedisce indietro un atto **Refuse** che termina la conversazione. Se invece decide di considerare la richiesta, il MA emette un atto **Propose**, il cui contenuto è lo stesso del **Cfp** ricevuto ma con in più:

- l'identità della macchina (i)
- lo starting time s_j proposto dal MA

Il valore dell'istante iniziale proposto dall'agente macchina può anche essere diverso da quello richiesto dal JA nel messaggio precedente. Per questo motivo, il JA può decidere di rifiutare la proposta oppure di accettarla. Nel primo caso, invia un atto **Reject-proposal** al MA, concludendo la conversazione. Nel secondo caso, invia **Accept-proposal** al MA.

A questo punto il MA può rispondere con un messaggio **Confirm** (che indica che l'accettazione della proposta è stata confermata, e chiude la conversazione con successo); oppure, nel caso non sia in grado di soddisfare la proposta fatta in precedenza, può modificarla con un messaggio **Update-proposal**, identico a **Propose**, ma con un diverso starting time s_j. In questo caso, si crea un ciclo all'interno del protocollo, in quanto il JA può rispondere nuovamente con un rifiuto o un assenso. Il ciclo continua finché o il JA rifiuta la proposta modificata, o il MA la conferma.

La figura seguente rappresenta lo stesso protocollo formalizzato con un diagramma di stato AUML. Nel diagramma si utilizza la notazione:

I = Initiator = messaggio inviato da JA

R = Respondent = messaggio inviato da MA

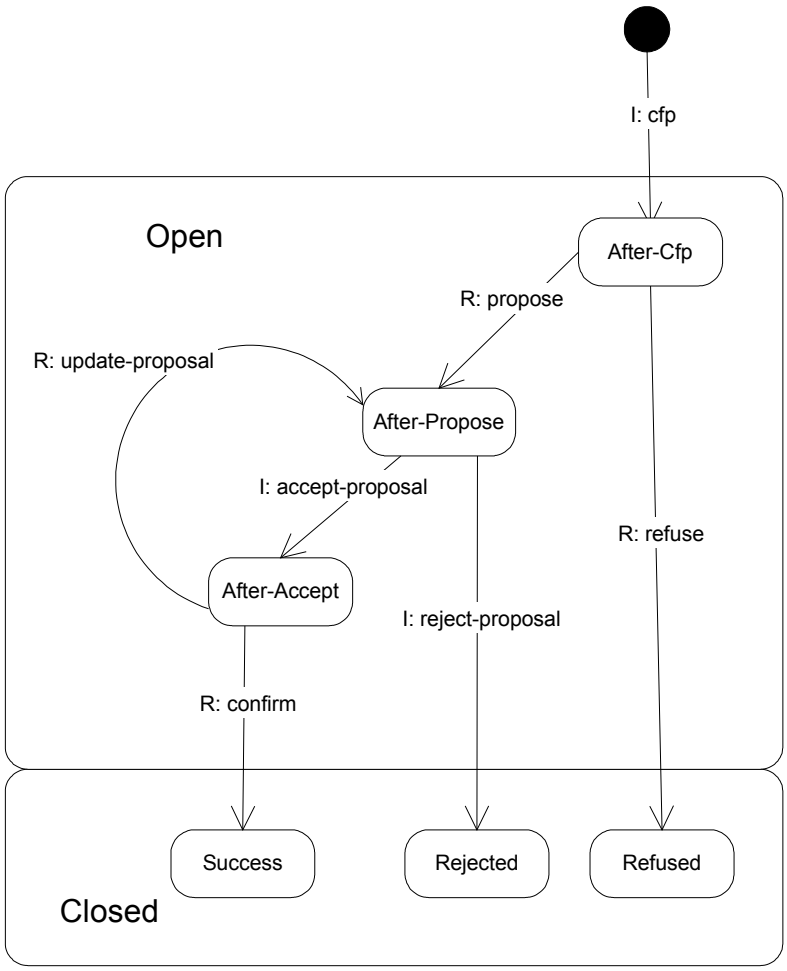


Figura 5-5.3 Protocollo tra JA e MA espresso come Statechart

La definizione del protocollo è un'astrazione che prescinde sia dai criteri decisionali adottati, sia dal fatto che contemporaneamente sono attivi molti agenti job e molti agenti macchina. Questi problemi saranno argomento dei prossimi paragrafi. In breve, un ciclo di contrattazioni tra più JA e MA avviene secondo il seguente schema:

1) gli agenti job (solo quelli che stabiliscono di farlo) emettono una richiesta di servizio (**Cfp**) diretta a tutti gli agenti macchina;

2) gli agenti macchina elaborano le risposte (rifiuti o proposte);

3) gli agenti job a cui è stata effettuata una proposta scelgono da quale macchina farsi servire e lo comunicano agli agenti macchina;

4) gli agenti macchina accettano solo la proposta più conveniente e comunicano il successo o il fallimento della transazione, e l'eventuale modifica delle proposte accettabili in seconda istanza;

5) il ciclo torna al punto 3 finché tutte le conversazioni ancora aperte non sono concluse (da messaggi **Confirm** o **Reject-proposal**).

La contrattazione può avvenire in un tempo quasi istantaneo oppure richiedere una sequenza anche molto lunga di iterazioni (proposta-accettazione/rifiuto-update) durante la quale al passo 4 gli agenti macchina possono modificare la proposta fatta ad un agente job qualora questi abbia accettata una proposta ma contestualmente il servizio sia stato concesso ad un altro job (in competizione) a causa della sua maggiore offerta (bid). Questo ciclo potrebbe in teoria non concludersi in tempi accettabili, ed a questo scopo è possibile introdurre un timeout applicato dall'agente RCA (al momento, tuttavia, questa possibilità non è stata prevista nel MASS sviluppato).

5.4.4 L'agente job

Un agente job JA_j viene creato dall'agente generatore JGA all'istante di tempo a_j. Alla nascita gli vengono trasmessi, oltre ai dati caratteristici del job associato e ad un budget iniziale B_0, anche un insieme di parametri caratteristici che influenzano il suo comportamento. Di questi parametri si parlerà più avanti nel paragrafo.

Il JA entra in azione tutte le volte che viene "risvegliato" dal RCA con un atto comunicativo **Inform-clock**. Tale messaggio viene inviato per la prima volta al JA non appena esso viene creato (ovvero all'istante a_j), mentre per le volte successive è il JA stesso a indicare al RCA quando desidera essere "risvegliato".

Il seguente diagramma di attività schematizza un ciclo di contrattazione per un singolo agente job.

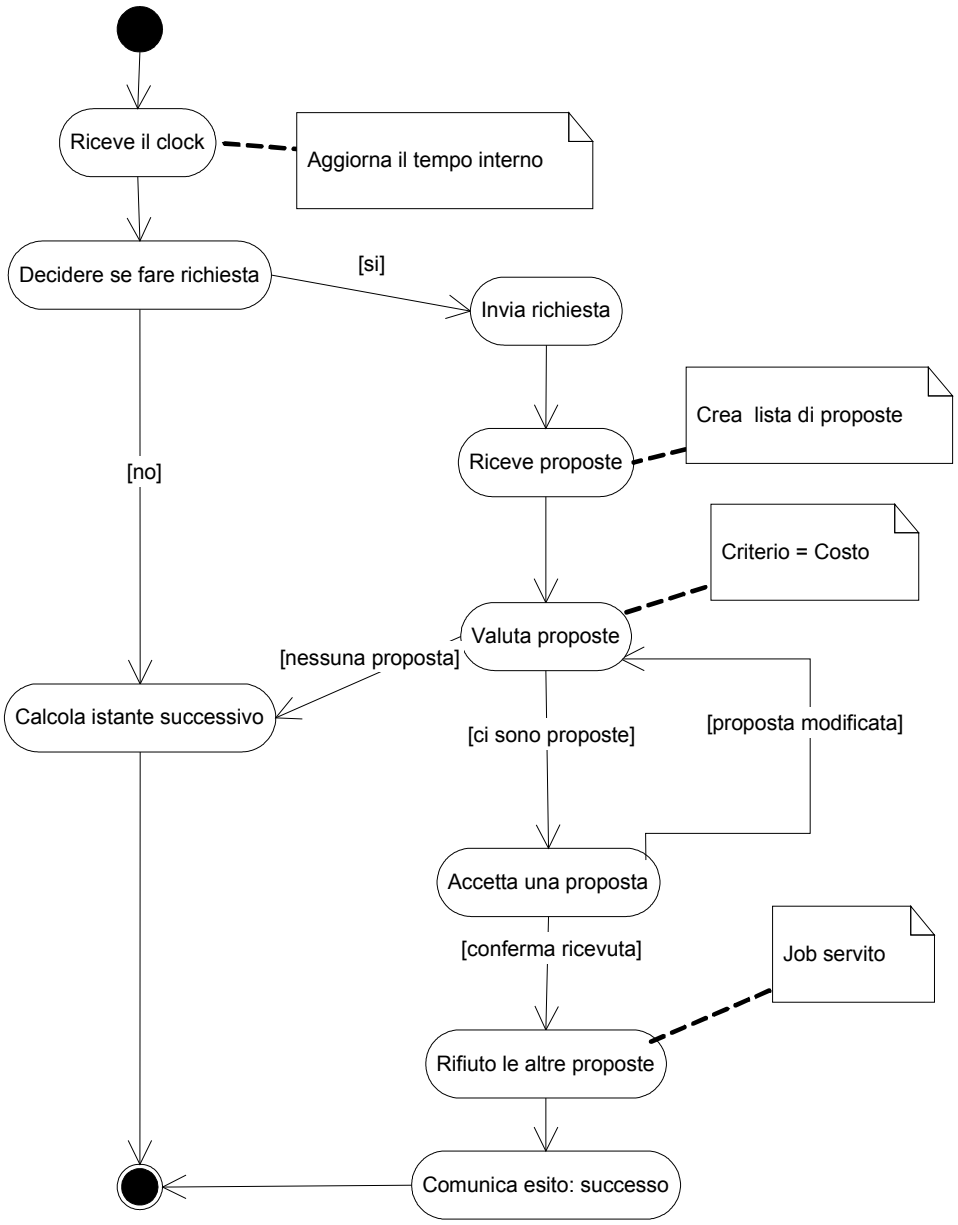


Figura 5-5.4 Activity Diagram di un JA

Un messaggio **Inform-clock** comunica il valore del tempo corrente τ e l'inizio di un ciclo di contrattazioni. Il JA deve decidere se e quando fare richieste di servizio: non è conveniente infatti fare richieste ad ogni ciclo, in quanto per ogni richiesta emessa (anche se con esito negativo) è necessario pagare un costo fisso RC. La strategia utilizzata per decidere se fare richiesta o meno è basata sul valore del budget, del tempo τ e di d_j.

L'obiettivo del JA è trovare una macchina che serva il job j in modo da minimizzare la somma pesata di earliness e tardiness, ovvero in modo che il tempo di completamento c_j coincida con la data di consegna d_j. L'agente job persegue un obiettivo che potremmo definire "egoistico", ovvero tenta di minimizzare il proprio costo parziale senza preoccuparsi di quello globale.

Il JA può dunque calcolare l'istante di starting time ottimo come segue:

$$s^*_j = \min_{s:s\geq r_j} V_j(s) = \min_{s:s\geq r_j} [we_j \cdot \max[0,d_j\text{-}s\text{-}p_j]+wt_j \cdot \max[0,s+p_j\text{-}d_j] \qquad \textbf{(0)}$$

Chiaramente il valore ottimo dello starting time è dato da:

- $s^*_j = d_j - p_j$ se $d_j - p_j \geq r_j$
- $s^*_j = r_j$ altrimenti

Nel primo caso si ha costo zero, nel secondo invece si ha che

$$V_j(s^*_j) = wt_j \cdot (r_j + p_j - d_j).$$

Il JA, dopo aver calcolato lo starting time ottimo, calcola quanto il tempo corrente si discosta da tale istante. Si definisce *float* di un job la quantità:

$$f_j = s^*_j - \tau$$

Quindi decide se fare una richiesta di servizio in base ad una funzione bid (f_j) chiamata *profilo di spesa*, che verrà discussa più avanti. Tale funzione ritorna un valore che indica quale percentuale del proprio budget corrente il JA è disposto a offrire. Se tale valore è zero, non viene emessa alcuna richiesta e, per il JA considerato, il ciclo di contrattazioni termina subito; al contrario, se il valore percentuale è maggiore di zero, viene emessa una richiesta con un bid pari a:

$$b_j = B \cdot bid(f_j)$$

dove b_j è il denaro offerto e B il budget corrente.

Il JA non invia la richiesta (**Cfp**) ad un solo agente macchina, ma a tutti quelli presenti nel sistema: si tratta dunque di una richiesta multicast. Dunque l'agente potrà ricevere sia messaggi **Refuse** che messaggi **Propose**. Se il JA non riceve proposte, dovrà attendere il prossimo ciclo di contrattazioni per poter avanzare un'altra richiesta. Se invece riceve delle proposte, sceglie tra esse la migliore, ovvero quella con costo minore, e la accetta inviando **Accept-proposal** alla particolare macchina da cui l'ha ricevuta, sia essa MA_i. Si noti che al momento JA non invia messaggi **Reject-proposal**, quindi tutte le altre proposte restano in sospeso.

Se MA_i risponde con un **Confirm**, il JA sa di essere stato schedulato, quindi può inviare **Reject-proposal** a tutte le altre macchine MA_k con $k \neq i$ (questo è un aspetto importante del comportamento dell'agente job, anche se non è messo in evidenza dal diagramma di protocollo descritto nel paragrafo precedente).

Se, al contrario, MA_i risponde con un messaggio **Update-proposal**, la proposta potrebbe non essere più la migliore: il JA ripete la valutazione dell'elenco delle proposte ricevute, e decide se inviare un nuovo messaggio **Accept-proposal** a MA_i oppure ad un'altra delle macchine. Il ciclo si ripete finché il JA non trova un agente macchina disposto a servirlo.

Il ciclo termina sempre (sarà chiaro quando verrà analizzato il comportamento dei MA), ma si può anche fare in modo che, dopo aver ricevuto un certo numero di messaggi **Update-proposal**, l'agente job si "spazientisca" e rifiuti tutte le proposte in sospeso.

Ogni volta che un JA termina il ciclo di contrattazioni con un insuccesso, ovvero:

- se decide di non fare richieste;
- se fa richieste ma non riceve alcuna proposta;
- se decide di rifiutare tutte le proposte;

allora deve informare l'agente RCA dell'esito negativo della contrattazione, con un messaggio **Inform-results** contenente il successivo istante di tempo in cui il JA vuole essere "risvegliato" dal RCA stesso, calcolato in base ad una funzione next(τ).

Se il JA è stato servito con successo, ha esaurito il suo compito e dunque muore. Prima di morire invia al RCA un messaggio **Inform-results** contenente una copia del messaggio **Confirm** ricevuto, in modo che il RCA sia informato dell'esito positivo della contrattazione.

Infine, occorre prendere in considerazione una possibilità che ancora non è stata trattata. È possibile che un agente job, a causa delle troppe richieste effettuate, esaurisca il proprio budget; al JA dunque viene concesso di emettere richieste non pagate (indicate dall'offerta simbolica $b_j = -1$) soltanto quando sono vere contemporaneamente due condizioni:

1) $f_j < 0$ (ovvero il job è in ritardo sui tempi di consegna);

2) $B < RC$ (il budget è stato esaurito)

Le richieste che non possono essere pagate vengono considerate dalle macchine solo quando non vi sono richieste a pagamento o quando il float è sufficientemente negativo da evidenziare una situazione critica (il comportamento dei MA è descritto nel prossimo paragrafo; tuttavia, se al problema fossero aggiunti vincoli di deadline, tale comportamento dovrebbe essere modificato in modo che i MA non possano, avendo la possibilità di servire un job, causarne la non ammissibilità).

Funzioni decisionali dell'agente job.

La funzione decisionale bid(f_j), utilizzata per decidere se fare richiesta e, se sì, quanto offrire, ha il seguente aspetto:

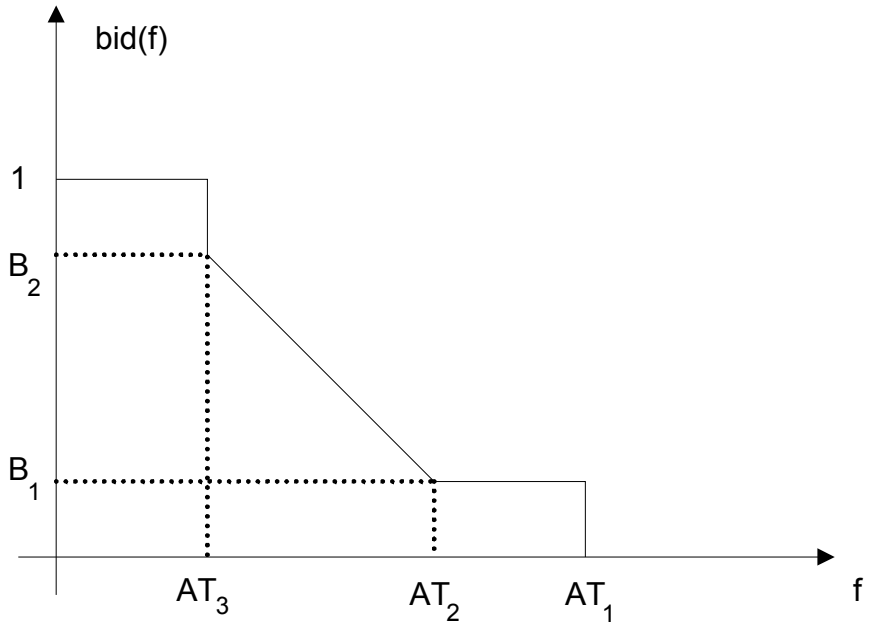


Figura 5-5.5 Profilo di spesa dell'agente job

Spesso il comportamento degli agenti viene modellato con metafore antropomorfe: possiamo quindi definire un JA "ansioso" (o "prepotente") se effettua richieste con molto anticipo. Ognuno dei tre valori del float indicati in ascisse è chiamato "soglia d'ansia" (anxiety threshold). I loro valori sono definiti come segue:

$$AT_1 = th_1 \cdot \Delta$$

$$AT_2 = th_2 \cdot \Delta$$

$$AT_3 = th_3 \cdot \Delta$$

Dove th_1, th_2 e th_3 fanno parte dei parametri caratteristici di comportamento ricevuti da JGA all'atto della nascita, mentre Δ è chiamato anche *tick* ed è un parametro deciso dall'agente RCA.

I valori B_1 e B_2 sono percentuali del budget totale, e anch'essi fanno parte dei parametri caratteristici; al contrario, la *forma* della funzione non ne fa parte.

- Se $f_j > AT_1$, la somma da offrire è zero, dunque il JA non emette richieste.
- Con $AT_2 < f_j < AT_1$, il JA offre un valore costante B_1, per poi salire progressivamente lungo la retta per $AT_3 < f_j < AT_2$ fino al valore B_2.
- Per $f_j < AT_3$, la funzione vale 1, il che corrisponde a spendere tutto il budget residuo. Anche se nella figura non è mostrato, se $f_j < 0$ la somma offerta è sempre massima.

Le tre soglie AT_h scandiscono i momenti critici nella vita del JA. Via via che il tempo avanza (τ aumenta), il float si riduce e l'agente diviene più ansioso di far eseguire il suo job, quindi è progressivamente sempre più disposto a spendere.

La funzione appena descritta è di per sé deterministica: ad ogni valore del float corrisponde una ben precisa percentuale di budget da offrire. Per aggiungere una componente non deterministica, si applica una "correzione" al float, prima di utilizzarlo come parametro della funzione.

La correzione è definita come segue:

$f' = f_j - \delta \cdot h_1 \cdot \Delta$ se $AT_2 < f_j < AT_1$

$f' = f_j + \delta \cdot h_2 \cdot \Delta$ se $AT_3 < f_j < AT_2$

$f' = f_j$ altrimenti

dove δ è un valore casuale estratto da una distribuzione uniforme tra 0 e 1, mentre h_1 e h_2 fanno parte dei parametri caratteristici del JA. Questi ultimi due valori influenzano ulteriormente l'ansietà dell'agente ("h" sta per *haste*, fretta).

Un elevato valore di h_1 e un basso valore di h_2 rendono il JA più ansioso e più propenso a spendere, e viceversa.

Una volta applicata la correzione, si calcola bid(f') invece che bid(f_j).

La funzione che calcola l'istante successivo in cui il JA vuole essere risvegliato, è definita in questo modo:

$$next(\tau) = \begin{cases} s^*_j - (th_1 \cdot \Delta + \Delta) & \text{se } f_j > th_1 \cdot \Delta \\ \tau + \Delta/2 & \text{se } f_j < th_3 \cdot \Delta \\ \tau + \Delta & \text{altrimenti} \end{cases}$$

Riassumendo, i parametri caratteristici dell'agente job sono: (th_1, th_2, th_3, B_1, B_2, h_1, h_2).

Si noti infine che il costo fisso di richiesta (RC) non fa parte di tali parametri, ma è un parametro del protocollo uguale per tutti i JA (viene loro comunicato dal JGA al momento della creazione).

5.4.5 L'agente macchina

Un agente macchina MA_i viene creato dall'agente generatore MGA al momento dell'avvio del sistema. Come nel caso del JA, alla nascita gli vengono trasmessi alcuni parametri caratteristici.

Quando riceve un messaggio **Inform-clock** da parte del RCA, il MA aggiorna il proprio tempo interno τ, ma non compie altre azioni; infatti, poiché il suo ruolo nel protocollo è quello di *Participant*, esso deve attendere che gli agenti job emettano richieste.

Anche nel caso dell'agente macchina è possibile rappresentare schematicamente ciò che avviene in un singolo ciclo di contrattazione con un diagramma di attività.

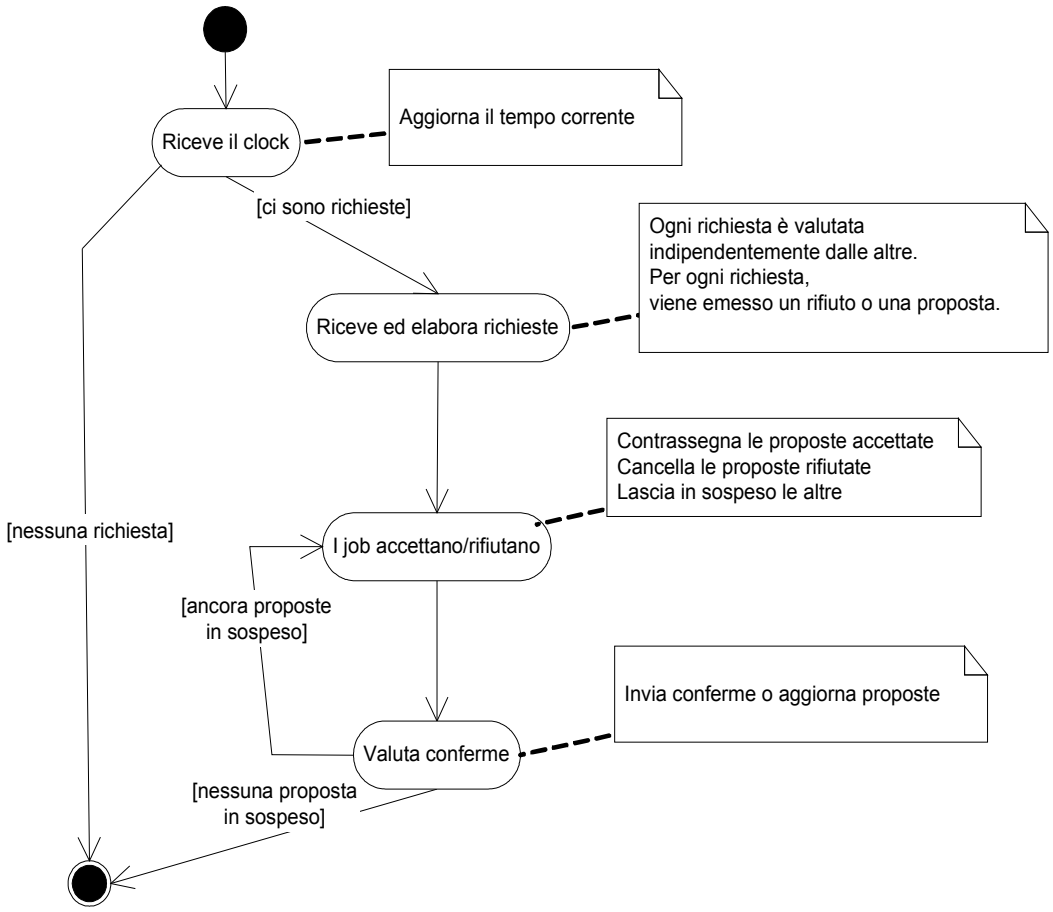


Figura 5-5.6 Activity Diagram di un MA

Ogni volta che un MA riceve un atto **Cfp**, per prima cosa deve decidere se considerarlo oppure no. Per fare questo, si basa su due parametri della richiesta: la distanza temporale dallo starting time ottimo (il *float*, definito nel paragrafo precedente) e l'offerta in denaro b_j (bid). La funzione decisionale sarà descritta più avanti nel paragrafo, ma brevemente si può dire che il MA preferisce considerare richieste con float piccoli e bid grandi piuttosto che il contrario. Se il responso della funzione decisionale è negativo, viene emesso un rifiuto (**Refuse**); viceversa, l'agente macchina cerca di servire il job nel modo migliore possibile. In questo caso, il MA analizza il proprio schedule e cerca un intervallo (*slot*) in cui servire il job, tale che:

- lo slot sia libero, ovvero non occupato da un job già schedulato in un precedente ciclo di contrattazioni;

- lo slot rispetti i vincoli, ovvero non inizi né prima di r_j né prima di τ;
- lo slot abbia il minor costo possibile.

Ovviamente se è disponibile lo slot ottimo, cioè $[s^*_j, s^*_j+p_j]$, esso viene automaticamente scelto. In caso contrario, il MA ne cerca uno del tipo:

$$[s^*_j-d, s^*_j-d+p_j] \qquad \text{(slot in anticipo o } \textit{early}\text{)}$$

oppure del tipo:

$$[s^*_j+d, s^*_j+d+p_j] \qquad \text{(slot in ritardo o } \textit{tardy}\text{)}$$

(con d>0 e tale da non violare i vincoli sopra definiti).

Per l'agente è indifferente proporre uno slot in anticipo o in ritardo: propone sempre quello con il costo più basso (il comportamento potrebbe essere modificato in modo da dare la precedenza a possibili slot early piuttosto che tardy).

Lo slot trovato viene inviato al JA in un messaggio **Propose**, e ogni proposta effettuata viene memorizzata. Si noti che ogni proposta è indipendente dalle altre, e in questo modo alcune tra esse potrebbero essere sovrapposte.

Il MA a questo punto riceverà un certo numero di messaggi **Accept-proposal** da parte dei job che hanno accettato le proposte fatte.

Per risolvere il problema delle eventuali proposte sovrapposte, l'agente macchina procede in questo modo:

- ordina la lista delle proposte accettate in ordine decrescente di offerta (bid), ed in caso di parità in base al valore (crescente) della funzione di costo.
- serve il primo job della lista, quello con l'offerta più alta (sia esso j);
- aggiorna il proprio schedule inserendo un nuovo slot assegnato al job j;
- se JA_j aveva emesso una richiesta pagata, incassa l'offerta;

- invia un messaggio **Confirm** a JA_j;

- se vi erano altri messaggi d'accettazione, riesamina lo schedule e valuta se è possibile servire gli altri job (sempre secondo la priorità data dall'ammontare dell'offerta) o se le proposte originali devono essere modificate;

- se possibile serve gli altri job seguendo i passi dal 2 in poi o comunica l'eventuale proposta modificata (con un messaggio **Update-proposal**), tornando a valutare i messaggi di accettazione al passo 6.

Ovviamente, in caso di arrivo di messaggi **Reject-proposal**, l'agente macchina si limita a cancellare le proposte dalla propria lista interna.

Il ciclo continua finché non ci sono più proposte in sospeso. Tale ciclo non può mai diventare infinito, in quanto, ad ogni iterazione, almeno un JA viene sicuramente servito e abbandona la competizione.

Funzioni decisionali dell'agente macchina.

Come si è detto, il comportamento di un agente MA prevede che decida prima a chi rispondere e quindi verifichi in che modo potrebbe servire le richieste prescelte. La decisione su quali job meritano il servizio viene presa sulla base dei due attributi float e bid attraverso i seguenti passi.

Prima di tutto, MA_i utilizza due valori di soglia per stabilire in base al float di una richiesta se rispondere in ogni caso o meno; siano $f_{min}<0$ e $f_{max}>0$ un valore negativo ed uno positivo del float (parametri caratteristici):

- se $f_j < f_{min}$, l'offerta viene sempre considerata e MA_i emetterà una proposta;

- se $f_j > f_{max}$, l'offerta non viene mai considerata e MA_i emetterà un rifiuto;

- se $f_{min} < f_j < f_{max}$, MA_i calcola la probabilità di accettare l'offerta sulla base della seguente funzione nei parametri f_j e b_j: la probabilità di accettare è p_a, la probabilità di rifiutare è pari a $1 - p_a$.

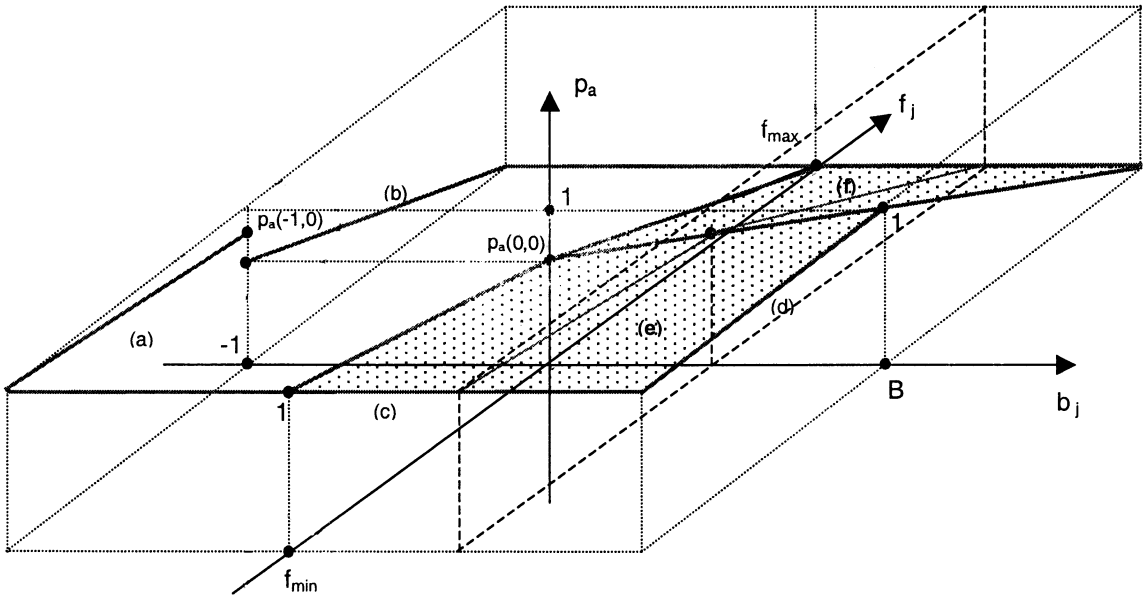


Figura 5-5.7 Funzione decisionale di un MA

Il grafico permette di determinare il valore della probabilità in funzione dei parametri della richiesta da parte di un agente JA_j; tale valore è definito per valori positivi di b_i e per $b_i = -1$, valore quest'ultimo che identifica le situazioni in cui il budget è stato esaurito e la richiesta di JA_j non può essere pagata.

La funzione $p_a(b_j, f_j)$ è solo una tra le diverse possibili che si potrebbero adottare. Tuttavia il comportamento determinato con tale funzione risponde ad una logica abbastanza intuitiva: maggiore è il float di una richiesta minore è la probabilità che questa sia considerata; più grande è l'ammontare dell'offerta maggiore è la probabilità che una richiesta sia considerata a parità di float; se l'agente job ha esaurito il proprio budget e non è stato ancora processato, la probabilità di essere considerato da un agente macchina cresce con il diminuire del float, ma in maniera più rapida rispetto al caso di offerte positive quando il float diventa negativo.

Le linee e le superfici evidenziate in figura 5-7 sono rispettivamente rette e piani che sono definiti come segue:

$$p_a(b_j, f_j) = \begin{cases} p_a(-1,0) + \dfrac{1 - p_a(-1,0)}{f_{min}} \cdot f_j & (f_{min} < f_j \leq 0) \wedge (b_j = -1) \quad (a) \\ p_a(-1,0)\left(1 - \dfrac{1}{f_{max}} \cdot f_j\right) & (0 < f_j \leq f_{max}) \wedge (b_j = -1) \quad (b) \\ 1 & f_j = f \geq_1 \quad \forall b_j \quad (c) \\ 1 & (f_{min} < f_j \leq 0) \wedge (b_j \geq B') \quad (d) \\ \hat{b} + \dfrac{1 - \hat{b}}{f_{min}} \cdot f_j & (f_{min} < f_j \leq 0) \wedge (b_j \geq 0) \quad (e) \\ \hat{b} \cdot \left(1 - \dfrac{1}{f_{max}} \cdot f_j\right) & (0 < f_j \leq f_{max}) \wedge (b_j \geq 0) \quad (f) \end{cases}$$

$$\hat{b} = p_a(0,0) + \frac{1 - p_a(0,0)}{B'} b_j$$

MA_i decide se accettare o rifiutare l'offerta sulla base del confronto tra un valore χ estratto casualmente da una distribuzione uniformemente distribuita tra 0 e 1 ed il valore p_a ossia accetta l'offerta se $\chi \leq p_a$.

Se l'agente MA_i stabilisce di non considerare la richiesta di servizio di un JA_j viene emesso un messaggio di rifiuto (**Refuse**).

Una volta stabilito di considerare una richiesta, gli agenti macchina cercano di rispondere sempre al meglio alla richiesta di un JA_j. Poiché l'obiettivo principale dei MA_i è massimizzare il numero di job serviti, è ipotizzabile una fase di contrattazione tra agente macchina e agente job durante la quale un MAi può decidere di accettare un'offerta (bid) minore per far fronte ad una propria incapacità di servire un JA_j secondo quanto richiesto (in ritardo o anticipo rispetto a s^*_j). Nel sistema di scheduling qui descritto questa possibilità non è ancora prevista, ma lasciata come possibile sviluppo futuro.

Riassumendo, i parametri caratteristici di un agente macchina sono i seguenti: (f_{min}, f_{max}, p_{00}, p_{01}, B').

5.5 Estensione del protocollo: le promesse di servizio

L'euristica di scheduling presentata non possiede alcuna capacità di *backtracking*: una volta che un contratto tra un JA ed un MA è stato stipulato, il JA muore e la decisione presa dal sistema non può essere annullata. Il meccanismo può essere migliorato e reso più dinamico introducendo la possibilità, per gli agenti macchina, di fare *promesse di servizio*: le promesse sono analoghe alle proposte, ma, a differenza di esse, sono caratterizzate da un intervallo di tempo nel quale possono essere *annullate* (sia da JA che da MA).

Per considerare anche le promesse e il loro annullamento, è necessario modificare il protocollo e il comportamento degli agenti (figura 5-8).

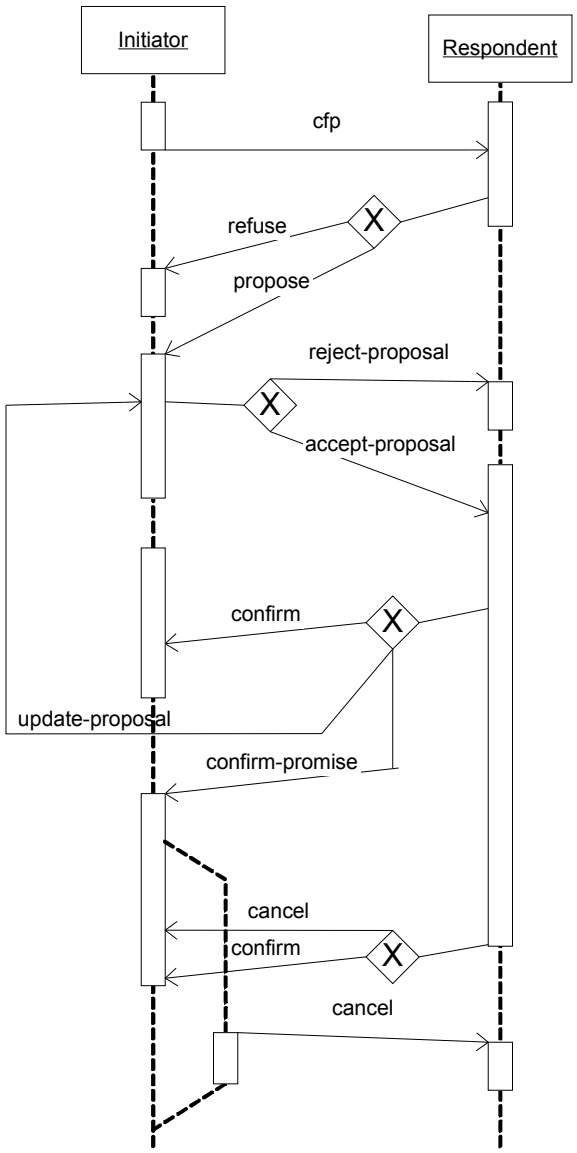


Figura 5-5.8 Diagramma del nuovo protocollo con promesse

La figura 5-9, invece, mostra lo stesso protocollo espresso come diagramma di stato (la notazione è la stessa vista per il protocollo base).

Inizialmente, il protocollo è identico al precedente: un JA_j emette una richiesta, un MA_i rifiuta oppure offre un servizio; nel secondo caso, il JA può rifiutare l'offerta o accettarla. Se il JA accetta e la macchina conferma, si possono verificare due casi, a seconda di quanto il valore del tempo corrente τ si discosta dallo starting time s_j proposto dalla macchina per il job:

- $s_j \leq \tau + \Delta_i$ (dove Δ_i è un parametro dipendente da MA_i). L'offerta è una *proposta* confermabile immediatamente, come nel protocollo base: MA_i invia un messaggio **Confirm**, il job j è schedulato e JA_j muore.

- $s_j > \tau + \Delta_i$. L'offerta è una *promessa di servizio*. MA_i invia un messaggio **Confirm-promise**.

Se viene confermata una *promessa*, l'agente job deve pagare una caparra (*earnest*) pari ad una certa percentuale del *bid* b_j offerto, che viene immediatamente incassata dall'agente macchina. In questo caso, il singolo ciclo di contrattazione termina, ma JA_j non muore: la conversazione con MA_i resta aperta.

Durante i cicli successivi, sia a JA_j che a MA_i è concesso di annullare la promessa (se ad esempio hanno trovato un'offerta migliore) inviando un messaggio **Cancel** alla controparte.

- Se JA_j annulla, *non* riceve indietro la caparra, che resta di proprietà di MA_i.

- Se MA_i annulla, deve restituire al JA_j la caparra, più una penale (*penalty*) corrispondente ad una data percentuale del *bid*.

In entrambi i casi, la conversazione è conclusa con fallimento. Se nessuna delle due controparti annulla, la promessa si conferma automaticamente quando

$$s_j \leq \tau + \Delta_i$$

nel qual caso, MA_i invia un messaggio **Confirm** e la conversazione è conclusa con successo. In questo caso, il JA deve pagare il conguaglio, ovvero la parte di bid che non ha ancora pagato (pari a b_j – *earnest*).

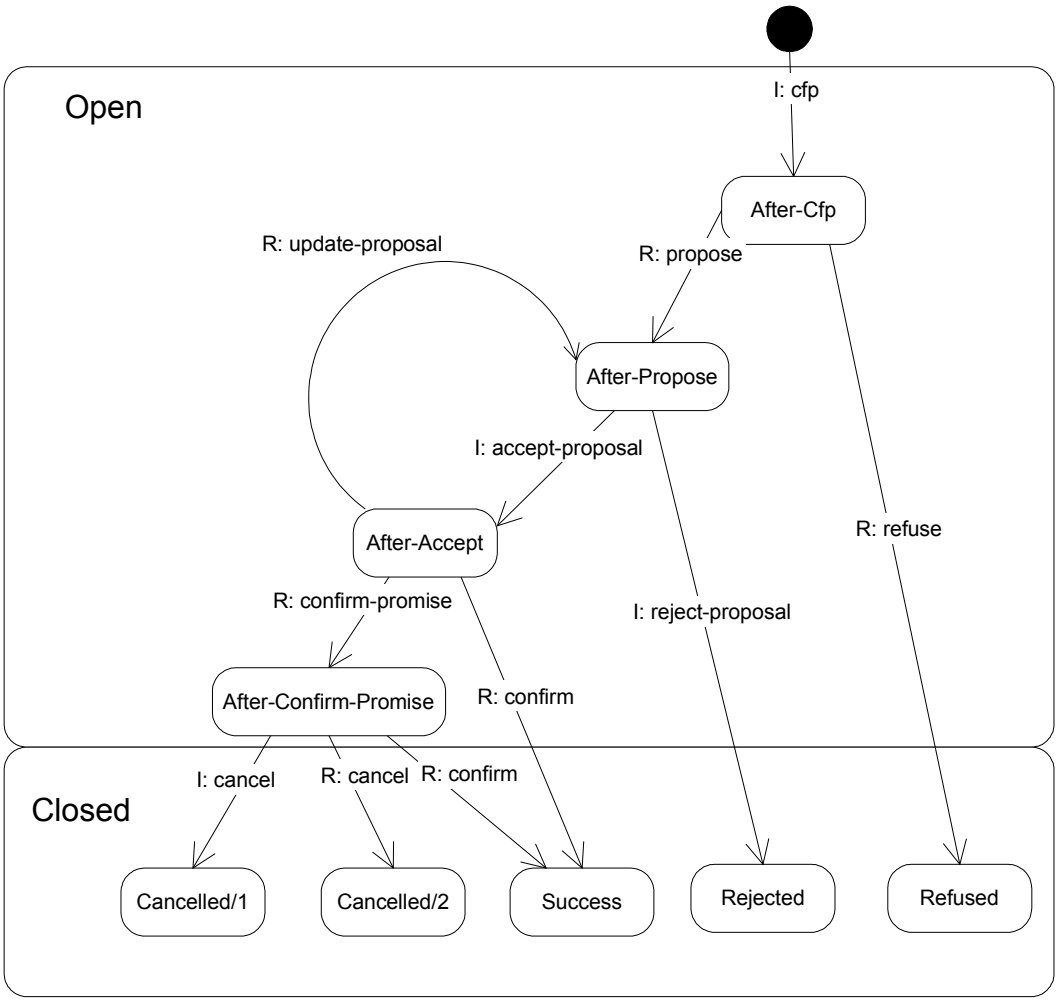


Figura 5-5.9 Statechart del protocollo con promesse

5.5.1 Comportamento dell'agente job

All'inizio di un ciclo di contrattazioni, JA controlla se possiede una promessa attiva, e se è giunta l'ora che venga confermata. Se $s_j \leq \tau + \Delta_i$, il job semplicemente attende di ricevere il **Confirm** dalla macchina.

Se JA non ha una promessa e/o non è giunta l'ora di confermarla, decide se fare altre richieste con il metodo classico. In caso decida di lanciare un nuovo insieme di **Cfp**, se ha già una promessa attiva deve tener conto dei propri debiti e verificare di non rischiare l'insolvenza: in questo caso, infatti, il JA sa che dovrà pagare un conguaglio al MA con cui ha la promessa, ma sa anche che emettere

una richiesta ha un costo fisso RC. Dunque, alle normali condizioni già discusse in §5.4.4, si aggiunge la condizione:

$RC < (B - (b_j - earnest))$

Dunque JA invia **Cfp** a tutti i MA (nel caso abbia una promessa, non invia **Cfp** al MA con cui ha tale promessa). All'arrivo di messaggi **Propose** crea due liste separate, quella delle proposte e delle promesse. Si noti che, per semplificare il protocollo e poter meglio costruire sul vecchio, si è deciso di *non* introdurre un nuovo atto comunicativo "*promise*": per questo motivo, un atto **Propose** può contenere sia una promessa sia una proposta vera e propria – quello che fa la differenza è il valore dello starting time s_j contenuto nel messaggio.

Quando a tutti i **Cfp** è stata data risposta (affermativa o negativa), JA mette a confronto i risultati ottenuti. In particolare è necessario mettere a confronto:

- la migliore delle promesse,
- la migliore delle proposte,
- l'eventuale promessa attiva precedente.

Per "migliore" si intende quella con funzione obiettivo (costo) minore (i criteri decisionali sono trattati nel seguito). Se è stata accettata una delle nuove proposte/promesse, JA invia **Accept-proposal**, e:

- se MA risponde con un **Update-proposal**, si ripete il processo di valutazione e scelta;
- se MA risponde con **Confirm**, JA è schedulato e muore;
- se MA risponde con **Confirm-promise**, JA registra la promessa attiva e paga la caparra;

Sia che riceva **Confirm**, sia che riceva **Confirm-promise**:

- se c'era una vecchia promessa ed è stata annullata, invia **Cancel** al relativo MA;

- invia **Reject-proposal** a tutti i MA le cui proposte/promesse non sono state accettate.

Se invece il JA decide di mantenere la vecchia promessa (nel caso in cui ne possegga una), invia **Reject-proposal** a tutti i MA che avevano fatto proposte/promesse.

Criterio di annullamento delle promesse da parte di JA

In questa sezione si utilizza la notazione seguente:

- P_1 è l'eventuale promessa attiva

- P_2 è la migliore delle promesse

- P_3 è la migliore delle proposte

- C(P) è il valore del costo (funzione obiettivo) della promessa o proposta P.

Il processo decisionale si svolge in due fasi: per prima cosa vengono confrontate P_2 e P_3:

  - Se $C(P_3) <= C(P_2)$ viene scelta P_3.

  - Se $C(P_2) – C(P_3) > C_{th}$ viene scelta P_2.

Nel caso sia presente P_1, deve essere confrontata con la migliore tra P_3 e P_2, nel modo che segue:

- Se era stata scelta P_2 (proposta):

  c) Se $C(P_2) <= C(P_1)$ viene scelta P_2.

  d) Se $C(P_1) – C(P_2) > C_{th}$ viene mantenuta P_1.

- Se era stata scelta P_3 (promessa):

  e) Se $C(P_1) <= C(P_3)$ viene mantenuta P_1.

  f) Se $C(P_3) – C(P_1) > C_{th}$ viene scelta P_3.

C_{th} è un valore di soglia che fa parte dei parametri caratteristici del JA.

La condizione (a) è banale; al contrario, la (b) non è così ovvia: la promessa (P_2) potrebbe essere più vantaggiosa in termini di costo, ma la proposta (P_3) è pur sempre un'assicurazione di servizio sicuro – in pratica, il criterio decisionale è il costo, ma si può dire che alla proposta venga dato un "maggior vantaggio".

Si noti che nei casi (c)/(d) si ha una competizione tra una promessa e una proposta: anche in questo caso viene data priorità alla proposta. Nei casi (e)/(f) si trovano a confronto due promesse: il criterio è analogo, ma viene data la precedenza alla vecchia promessa (per la quale si è già pagata una caparra).

È evidente che nei casi (c) ed (f) deve essere annullata la promessa attiva.

5.5.2 Comportamento dell'agente macchina

All'inizio di un ciclo di contrattazioni, MA controlla se possiede delle promesse attive, e se è giunta l'ora di confermarle. Per ciascun JA_j a cui è stata fatta una promessa, si ha che se $s_j \leq \tau + \Delta_i$, allora MA aggiunge il job j allo schedule, manda **Confirm** al JA_j, incassa il conguaglio e chiude il contratto.

Le promesse non ancora confermabili vengono per il momento mantenute. Dopo questa fase preliminare, il MA si mette in ascolto di nuove richieste.

Ogni volta che l'agente macchina riceve un **Cfp** e decide di considerarlo (con il metodo classico, la funzione p_a di due variabili) emette un messaggio **Propose** contenente lo slot migliore, calcolato *non tenendo conto delle promesse fatte in passato*: in tal modo, lo slot potrebbe andare in conflitto con una o più

promesse attive. Ogni volta che un JA accetta una proposta (con **Accept-proposal**) l'accettazione viene registrata; quando tutte le proposte emesse sono state accettate o rifiutate, MA le valuta seguendo la solita priorità: in ordine di bid.

Per ogni proposta accettata possono verificarsi diverse possibilità:

- Non è schedulabile perché si sovrappone con un'altra proposta che è stata confermata prima perché con bid più alto: come nel protocollo base, calcola un nuovo slot (sempre senza tener conto delle promesse attive) e invia **Update-proposal**.

- Sarebbe schedulabile, ma non lo è perché si sovrappone a una o più promesse attive fatte in cicli precedenti e non ancora confermate. Se MA lo ritiene conveniente (secondo un criterio descritto in seguito) annulla tutte queste promesse (inviando **Cancel** ai vari JA coinvolti) per confermare la nuova proposta. Se invece non lo ritiene opportuno, calcola un nuovo slot *che non si sovrapponga con alcuna promessa* e invia **Update-proposal**.

- È schedulabile, non si sovrappone con promesse, e vale la condizione $s_j \leq \tau + \Delta_i$: si tratta di una proposta confermabile immediatamente, dunque MA schedula il job e invia **Confirm**; il contratto è concluso.

- È schedulabile, non si sovrappone con promesse, e vale la condizione $s_j > \tau + \Delta_i$: questa è una *nuova promessa*: MA la aggiunge alla lista delle promesse attive, incassa la caparra, invia **Confirm-promise** e il contratto resta aperto.

Criterio di annullamento delle promesse da parte di MA

In questa sezione si utilizza la notazione seguente:

- P_x è la nuova proposta (o promessa),

- K è il numero di vecchie promesse attive potenzialmente sovrapposte a P_x
- b_x è il bid totale di P_x
- b_j è il bid totale della promessa fatta al job j
- $earnest_j$ è la caparra pagata dal job j
- $penalty_j$ è la penale da pagare al JA_j nel caso si annulli la promessa.

MA annulla le K promesse che si sovrappongono con P_x se:

$$b_x > \sum_{j=0}^{K} (b_j + penalty_j)$$

Ovviamente deve anche essere vero che:

$$B > \sum_{j=0}^{K} (earnest_j + penalty_j)$$

dove B è il budget corrente dell'agente macchina. Infatti l'agente macchina, se annulla le promesse, deve restituire ad ogni JA_j la caparra versata, più una penale.

5.5.3 Vantaggi dell'approccio basato sulle promesse

Il vantaggio apportato dalle promesse è proprio il fatto che non costituiscono un contratto inscindibile, ovvero possono essere rotte per far posto ad una migliore configurazione di job.

Un caso in cui la presenza di promesse risulta molto utile è mostrato nel seguente esempio: si considerino tre job (j = 1, 2, 3) che devono essere assegnati ad una singola macchina; i dati caratteristici dei job devono rispettare i seguenti vincoli:

- $a_1, a_2 << a_3$ (i primi due job arrivano molto prima)

- $d_1 < d_3 < d_2$ (la *due date* del terzo cade in mezzo alle altre due)
- $d_1 > d_3 - p_3$ (all'ottimo, i job 1 e 3 si sovrapporrebbero)
- $d_3 > d_2 - p_2$ (all'ottimo, i job 3 e 2 si sovrapporrebbero)
- $d_1 < d_2 - p_2$ (all'ottimo, i job 1 e 2 non si sovrapporrebbero)

Con tali vincoli, è evidente che i tre job non possono essere tutti schedulati con costo zero: la situazione è rappresentata nella figura seguente.

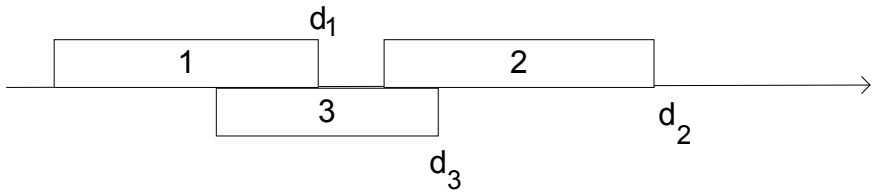


Figura 5-5.10 Tre job di esempio

Dal momento che a_3 è molto maggiore di a_1 e a_2, inizialmente nel sistema sono presenti soltanto gli agenti JA_1 e JA_2: i rispettivi job non sono in competizione e dunque vengono entrambi schedulati Just-In-Time (ovvero in modo che vengano completati esattamente alla *due date*) dall'agente macchina.

Figura 5-5.11 I primi due job schedulati a costo zero

Successivamente all'istante a_3, nel sistema viene introdotto l'agente JA_3, tuttavia l'intervallo di tempo rimasto tra i job 1 e 2 è minore di p_3. A questo punto occorre distinguere due casi:

2. nel sistema base – senza promesse – il job 3 verrebbe schedulato early o tardy; per semplicità, supponiamo che venga schedulato tardy: la situazione è quella descritta in figura 5.12;

3. nel sistema con le promesse gli agenti JA_1 e JA_2, avendo fatto richiesta molto in anticipo rispetto all'istante di *starting time* ottimo, ricevono dall'agente macchina una *promessa di servizio*. Se la richiesta avanzata da JA_3 è più conveniente (*bid* molto alto) l'agente macchina, applicando i criteri decisionali visti in §5.5.2, può decidere di annullare le promesse fatte in precedenza. Nei cicli successivi, gli agenti JA_1 e JA_2 inviano nuovi **Cfp** all'agente macchina, che decide di schedularli uno early e uno tardy. Lo schedule in questo caso è quello illustrato in figura 5.13.

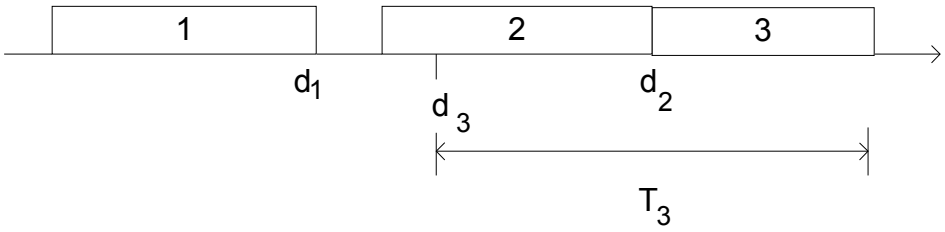


Figura 5-5.12 Schedule finale dei tre job nel caso a).

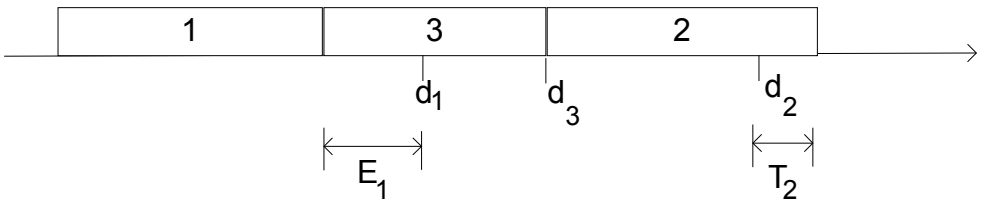


Figura 5-5.13 Schedule finale dei tre job nel caso b).

Siano C_a e C_b i valori che la funzione obiettivo assume rispettivamente nei casi a) e b).

- $C_a = wt_3 \ (d_2 + p_3 - d_3)$
- $C_b = we_1 \ (d_1 - d_3 + p_3) + wt_2 \ (d_3 + p_2 - d_2)$

Si vuole dimostrare che, nel caso b), in seguito all'annullamento delle promesse alcuni job vengono schedulati in modo peggiore, ma complessivamente l'obiettivo migliora. Supponiamo per semplicità che i valori dei pesi di earliness e tardiness siano gli stessi per tutti i job: $we_1 = wt_2 = wt_3$.

In questo caso, si ha che $C_a > C_b$ quando:

$$d_2 + p_3 – d_3 > d_1 – d_3 + p_3 + d_3 + p_2 – d_2$$

ovvero

$$2\ d_2 – d_3 > d_1 + p_2$$

Poiché, per ipotesi, $d_1 < d_2 - p_2$, si può operare una maggiorazione sostituendo $(d_2 - p_2)$ a d_1:

$$2\ d_2 – d_3 > d_2 - p_2 + p_2 > d_1 + p_2$$

ovvero

$$d_2 > d_3$$

che è vero per ipotesi. Si è dunque dimostrato che la presenza di promesse di servizio è conveniente anche in un caso molto semplice e con pesi uguali per tutti i job (anche se questo non implica che l'utilizzo delle promesse sia vantaggioso in qualsiasi caso). Appare evidente che la convenienza è ancora maggiore nel caso in cui il job 3 sia più prioritario, in quanto ciò implica:

- $wt_3 >> we_1, wt_2$ (è più conveniente schedulare il job 3 Just-In-Time);
- budget assegnato a JA_3 molto elevato (il *bid* sarà più alto e la macchina sarà più propensa ad annullare altre promesse).

5.6 L'implementazione

Nell'ambito di questa tesi, sono state realizzate due diverse implementazioni del sistema di scheduling qui proposto.

5.6.1 Implementazione di test

Questa prima implementazione è stata realizzata per testare le capacità del sistema di scheduling: non si tratta di un vero sistema ad agenti, ma di una normale applicazione Java a singolo thread. Un insieme di classi simulano il comportamento degli agenti job e macchine, e una classe `Coordinator` gestisce completamente al proprio interno l'evoluzione del sistema.

Quest'implementazione non è molto flessibile, e su di essa non è stata sperimentata l'estensione del protocollo basata sulle promesse di servizio.

5.6.2 Implementazione ad agenti

Questa seconda implementazione è stata realizzata utilizzando il framework ad agenti FIPA-OS (vedi §4.3.2): per prima cosa sono stati definiti i protocolli di interazione utilizzati dagli agenti JA e MA.

Per definire un protocollo in FIPA-OS si crea una classe contenente un *grafo orientato* che rappresenta i passi del protocollo che si vuole implementare. Il grafo si ricava senza difficoltà se i protocolli sono stati formalizzati utilizzando i diagrammi di stato AUML: si vedano le figure 5-3 e 5-8.

Ogni stato dei diagrammi (compreso lo stato iniziale, ma esclusi gli stati finali) viene rappresentato con un array di oggetti: gli elementi di tali array descrivono le transizioni tra gli stati. Ogni transizione è caratterizzata da una stringa che indica il *performative* (il tipo di messaggio inviato), da un indicatore che specifica come l'agente destinatario deve interpretare il messaggio (richiede un'azione, non richiede un'azione o è uno stato finale), e da un secondo indicatore che specifica il mittente del messaggio (0 = initiator, 1 = participant); inoltre ogni array può contenere puntatori ad altri array per indicare a quali stati puntano le transizioni. L'array che rappresenta lo stato iniziale deve chiamarsi `__protocol` e deve essere una variabile di classe (vedi §3.5.1), mentre gli altri possono avere nomi qualsiasi; la convenzione vuole che il nome di un array – cioè il nome di uno

stato del protocollo – sia composto dalla parola "after" ("dopo") seguita dal *performative* che ha portato in quello stato.

Ad esempio, quella che segue è la rappresentazione "ad array" del protocollo base di figura 5-3 (il protocollo con le promesse è analogo):

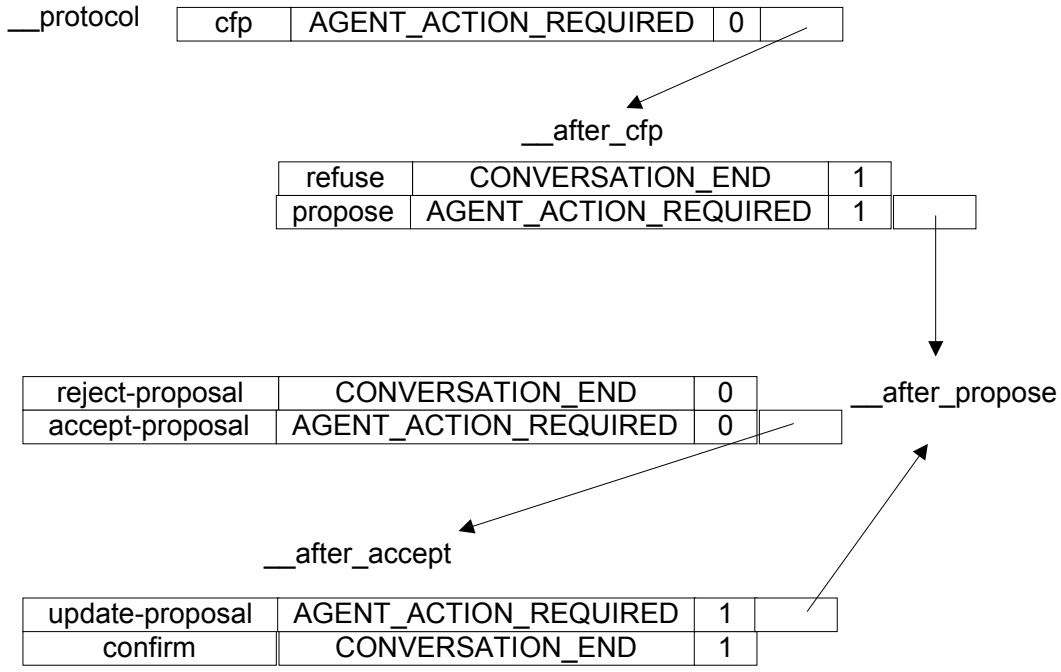


Figura 5-5.14 Grafo del protocollo di scheduling base

La classe contenente il grafo così definito ha una doppia utilità:

- Consente a FIPA-OS di controllare la validità di una conversazione a run-time: se a causa di un errore un agente invia un messaggio non consentito dalla sequenza (o anche se invia lo stesso messaggio due volte), viene sollevata un'eccezione.

- Consente allo strumento `TaskGenerator` di creare automaticamente il codice per semplificare l'implementazione degli agenti Initiator e Respondent.

Come si è detto nella descrizione di FIPA-OS (§4.3.2), in questo framework ogni agente può essere costituito da più *task*, ognuno dei quali

caratterizzato da un compito specifico. In particolare, il `TaskGenerator` crea due task che hanno il compito di gestire la conversazione descritta dal protocollo che si desidera implementare, uno per l'agente che inizia la conversazione e uno per l'agente che risponde.

Per comprendere l'utilità dei task generati automaticamente, si osservi lo schema seguente, che rappresenta in modo astratto la struttura interna dell'agente job (l'agente macchina è analogo).

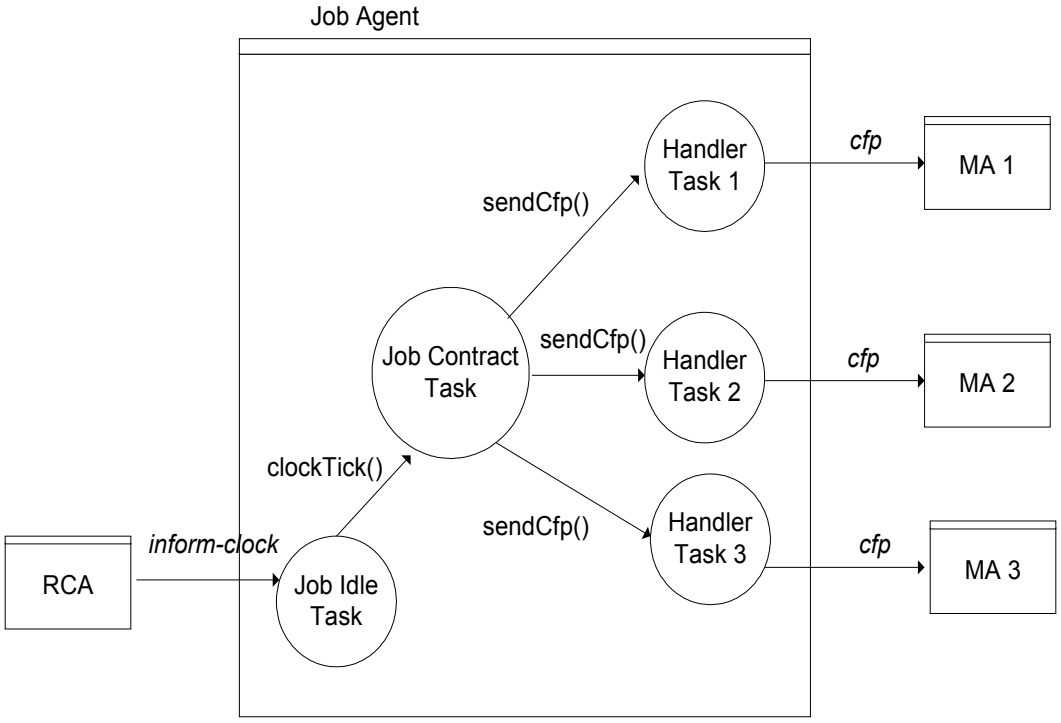


Figura 5-5.15 Schema dell'agente job

Gli agenti sono rappresentati con il simbolo visto in §3.6.5; per la rappresentazione dei task sono stati utilizzati dei cerchi in quanto l'AUML non possiede un simbolo dedicato (i task indicati come "handler" sono stati generati automaticamente dal `TaskGenerator`). Quando un agente job riceve un atto comunicativo **Inform-clock**, esso viene intercettato dal task ascoltatore (*Idle Task*) in quanto non facente parte di conversazioni già iniziate: tale task estrae dal messaggio le informazioni sul tempo corrente e le trasmette al task centrale dell'agente (*Contract Task*) invocando un metodo apposito. Il Contract Task

incapsula la logica necessaria per condurre un singolo ciclo di contrattazioni: se decide di avanzare una richiesta, effettua una ricerca presso il *Directory Facilitator* (non mostrato in figura, vedi §3.4.5) per scoprire quali agenti macchina sono presenti nel sistema; quindi crea tanti *Handler Task* quanti sono i MA. Ogni Handler Task gestisce una conversazione con un singolo MA, semplificando molto l'implementazione del Contract Task, che può così concentrarsi sulla *businnes logic* del problema senza dover tener conto del fatto che sta comunicando con altri agenti (ad es. non deve gestire direttamente gli AID e i messaggi ACL, perché è compito dei vari Handler Task): se ad esempio il Contract Task vuole fare una richiesta, invoca semplicemente `sendCfp()` su tutti gli Handler Task; a loro volta gli Handler Task invocheranno appositi metodi (non mostrati in figura) per informare il Contract Task dell'avvenuta ricezione di messaggi da parte degli agenti macchina (ad es. **Refuse**, **Propose**, ecc.).

Capitolo 6: Risultati sperimentali

6.1 La procedura per l'analisi delle prestazioni

Per analizzare le prestazioni del sistema multi-agente sviluppato, è stato generato un insieme di problemi di test e si sono confrontati i valori della funzione obiettivo ottenuti:

- con il MASS sviluppato
- con la soluzione ottima ricavata risolvendo esattamente le versioni off-line degli stessi problemi

Tale analisi statistica può essere considerata un primo passo verso un'accurata *analisi di competitività* (vedi §2.5) che farà parte degli sviluppi futuri di questa ricerca.

Data un'istanza I di un problema, e indicando con S_{opt} lo schedule ottimo ottenuto risolvendo la versione off-line di I, è possibile valutare la seguente espressione:

$$E\left[\frac{C(S,I)}{C(S_{opt},I)}\right] \leq k \qquad (1)$$

dove la media è calcolata su un insieme di problemi generati casualmente (secondo la procedura descritta nel seguito), e k è un valore costante che rappresenta il rapporto medio di competitività (ACR, Average Competitive Ratio). Questo tipo di analisi è necessaria per diverse ragioni:

- a quanto si conosce, non esistono in letteratura benchmark appropriati per la versione off-line del problema
- in letteratura vi sono inoltre scarsi risultati riguardanti problemi di scheduling E/T on-line
- non è semplice effettuare un'analisi del caso peggiore a causa del comportamento non deterministico degli agenti (vedi §5.4.4 e §5.4.5).
- non è semplice valutare l'influenza dei parametri del sistema nel processo di scheduling

6.1.1 La procedura per la generazione dei problemi

I problemi di test sono sequenze di job generate off-line tramite la seguente procedura. Nel seguito verrà utilizzata la notazione

$$x = U[min, max]$$

per indicare che il valore x è generato casualmente con una distribuzione uniforme tra un valore minimo ed uno massimo.

Ogni istanza è caratterizzata da M macchine e N job. Per ogni job j si ha che:

$$p_j = U[p_{min}, p_{max}]$$

$$r_j = U[0, \alpha \cdot \bar{p} \cdot N]$$

$a_j = \max[0, r_j - q_j]$ dove $q_j = U[0, \beta \cdot \bar{p}]$

$d_j = r_j + p_j + \gamma \cdot \bar{p}$

dove

- p_{min}, p_{max} e $\bar{p}$ sono rispettivamente i valori minimo, massimo e medio della durata (processing time) dei job
- α è una costante positiva utilizzata per influenzare il livello di conflitto tra le richieste di servizio dei job.
- β è un valore intero positivo
- γ è un valore intero positivo, e il valore $\gamma \cdot \bar{p}$ (chiamato *slack*) è utilizzato per influenzare la distribuzione dei job nel tempo, e quindi determinare la difficoltà del problema.

6.2 Formulazione a numeri interi del problema off-line

È necessario a questo punto confrontare il problema on-line definito nel capitolo precedente con il suo equivalente off-line, formalmente definito come "*off-line parallel machine JIT scheduling problem*".

Siano dati N job indipendenti, j=1,...,N da schedulare su M macchine identiche parallele. Le caratteristiche di job e macchine sono le stesse del problema on-line, con l'unica differenza che in questo caso il problema è risolubile off-line in quanto tutti i dati relativi ai job sono noti a priori.

Per ogni job j, le definizioni di p_j, r_j, d_j, we_j, wt_j, S_j, C_j, E_j e T_j sono le stesse viste in §5.3 (si noti che in questo caso non esiste a_j).

Scopo del problema è trovare uno schedule S ammissibile tale da minimizzare il costo, dato da:

$$C(S) = \sum_{j=1,\ldots N} (we_j \cdot E_j + wt_j \cdot T_j) \quad (2)$$

La funzione obiettivo (2) non è regolare (considera anche l'earliness oltre alla tardiness), e dunque la soluzione ottima potrebbe non appartenere alla classe degli schedule semi-attivi (vedi §2.2.3). La notazione di Graham per questo problema è **P/r$_j$/ET**.

Il problema è NP-hard (ovvero non è possibile trovare la soluzione ottima in un tempo polinomiale) in quanto il seguente caso particolare:

- una sola macchina
- d_j diversa per ogni job j
- we_j e wt_j diversi per ogni job j

è NP-hard, come dimostrato in [Zhu00]. In questa sezione viene illustrata una formulazione matematica del problema P/r_j/ET visto come un problema di *mixed integer programming* (MIP): il problema è molto simile a quelli proposti in [Zhu00] e [Balak99]. In tale formulazione, le variabili C_j, E_j e T_j, per ogni job j=1,...,N, sono considerate continue. Inoltre vengono definiti due insiemi di variabili binarie (denominate anche "variabili 0/1" in quanto possono assumere solo i valori 0 e 1):

- variabili di assegnazione Y_{ij}, per i=1,...,M e j=1,...,N, tali che Y_{ij}=1 se il job j è assegnato alla macchina i, e Y_{ij}=0 altrimenti;

- variabili di sequenza "deboli" $X_{jk,}$ per j=1,...,N-1 e k=1,...,N, k>j, tali che X_{jk}=1 se il job j precede il job k nel caso siano entrambi assegnati alla stessa macchina, e X_{jk}=0 se il job k precede il job j sempre nel caso siano entrambi assegnati alla stessa macchina.

Le variabili X_{jk} sono chiamate deboli (*weak*) in quanto non implicano alcun tipo di precedenza tra due job assegnati a macchine diverse; in questo caso il valore assunto da tali variabili è irrilevante ai fini del problema.

La formulazione MIP per il problema P/r_j/ET è la seguente:

$$\min C(S) = \sum_{j=1,...N} (we_j \cdot E_j + wt_j \cdot T_j) \tag{3}$$

con i seguenti vincoli:

$C_j \geq r_j + p_j$	j=1,...,N	(4)
$C_j + E_j - T_j = d_j$	j=1,...,N	(5)
$\sum_{i=1,...,M} Y_{ij} = 1$	j=1,...,N	(6)
$C_j - C_k + B \cdot (3 - X_{jk} - Y_{ij} - Y_{ik}) \geq p_j$	i=1,...,M, j=1,...,N-1 k=j+1,...,N	(7)
$C_k - C_j + B \cdot (2 + X_{jk} - Y_{ij} - Y_{ik}) \geq p_k$	i=1,...,M, j=1,...,N-1 k=j+1,...,N	(8)
$C_j, E_j, T_j \in \Re^+$	j=1,...,N	(9)
$Y_{ij} \in \{0,1\}$	i=1,...,M j=1,...,N	(10)
$X_{jk} \in \{0,1\}$	j=1,...,N-1 K=j+1,...,N	(11)

Tabella 6-6.I Vincoli del problema

Il vincolo (4) indica semplicemente che la soluzione deve essere ammissibile. Il (5) corrisponde alla definizione dell'earliness e della tardiness dei

job; si noti che tali vincoli, insieme alla funzione obiettivo da minimizzare, garantiscono che la relazione complementare $E_j \cdot T_j = 0$ sia sempre soddisfatta. Il vincolo (6) obbliga ogni job ad essere assegnato ad una delle macchine disponibili. I vincoli (7) e (8) corrispondono ai ben noti vincoli disgiuntivi della sequenziazione, ovvero impongono che, se due job j e k sono assegnati alla stessa macchina, o j precede k o viceversa. La quantità B è una costante arbitraria tale che sia molto maggiore di tutte le altre costanti del problema, ovvero:

$$B >> \max_{j=1,\ldots,N} r_j + \sum_{j=1,\ldots,N} p_j$$

Si noti che i vincoli (7) e (8) consentono l'uso delle variabili "deboli", in quanto il valore di X_{jk} è irrilevante quando sia Y_{ij} che Y_{ik} sono diversi da 1 per una data macchina i. Infine, (9), (10) e (11) definiscono semplicemente la natura delle variabili del problema.

La formulazione MIP appena definita appare molto compatta, poiché il numero di variabili 0/1 definite, pari a $N \cdot M + N \cdot (N-1)/2$, è minore rispetto ad altri problemi reperibili in letteratura. Tuttavia, gli esperimenti di complessità computazionale eseguiti in [Balak99] su un problema molto simile (nel quale vengono considerati anche i tempi di set-up) indicano che le istanze di problemi che possiamo aspettarci di risolvere in tempi ragionevoli sono caratterizzate *al massimo* da 12 job e 3 macchine.

Inoltre, occorre notare che per ottenere questi risultati [Balak99] usa un particolare approccio chiamato "decomposizione di Bender": si tratta di una tecnica di decomposizione iterativa che ad ogni ciclo risolve una coppia di

problemi, dei quali il primo (con variabili binarie 0/1) si occupa dell'assegnazione e sequenziazione dei job alle macchine, mentre l'altro (senza variabili binarie) si occupa del time-tabling dei job (la decomposizione di Bender è trattata in [Nemhauser]).

Da queste considerazioni segue che questa formulazione è adatta a problemi caratterizzati da un numero ridotto di job e macchine, ma non è da considerare come un approccio generale per la risoluzione di problemi $P/r_j/ET$.

6.2.1 Il sofwtare LINGO

La formulazione appena descritta è stata utilizzata per trovare le soluzioni ottime dei problemi di scheduling generati casualmente: per questo scopo è stato utilizzato *LINGO*.

Sviluppato dalla LINDO Systems, LINGO è uno strumento sofwtare in grado di definire e risolvere problemi di ottimizzazione lineare, non lineare e a numeri interi in modo semplice ed efficiente: comprende un ambiente grafico, uno specifico linguaggio per la descrizione dei problemi (la formulazione dei problemi con LINGO è simbolica e consente di utilizzare equazioni come quelle definite nella sezione precedente) e alcuni *solvers*, cioè programmi di ottimizzazione già pronti ed integrati con LINGO.

Per risolvere i problemi di test è stato utilizzato un algoritmo basato sulla tecnica di *branch and bound*: tuttavia, a causa dell'elevata complessità di alcuni dei problemi considerati, è stato fissato un limite superiore al tempo necessario per la ricerca della soluzione. In questo modo, per i problemi più difficili lo schedule individuato da LINGO non sarà mai esattamente ottimo, ma si avvicinerà alla soluzione ottima abbastanza da consentire una buona analisi di competitività.

6.3 Valutazione della difficoltà dei problemi

Sono stati definiti tre indici per misurare a priori la possibile difficoltà nel trovare una soluzione a costo zero per le versioni off-line delle istanze dei problemi. Il primo indice rappresenta le dimensioni dell'istanza e corrisponde al rapporto N/M.

Il secondo indice è la massima sovrapposizione delle richieste (*Maximum Request Overlap* o *MRO*), corrispondente al minimo numero di macchine necessarie per schedulare i job senza alcuna penalità.

Il terzo indice è la sovrapposizione media tra le richieste (*Average Request Overlap* o *ARO*), valutata come segue:

$$ARO(I) = \frac{1}{N(N-1)} \sum_{j=1}^{N} \frac{\sum_{h=1;h\neq j}^{N} \delta_{jh}}{p_j} \qquad (4)$$

dove δ_{jh} è la parte di processing time del job j che può sovrapporsi con il job h nel caso in cui i due job siano assegnati alla stessa macchina e terminino esattamente alla due date.

Apparentemente, dovrebbe essere più semplice risolvere istanze di problemi con valori bassi dei tre indici; in particolare, il MRO e l'ARO forniscono una sorta di misura della difficoltà in quanto tengono conto di come le richieste dei job sono distribuite nel tempo nonché dei possibili conflitti.

6.4 Risultati delle prove sperimentali

Il primo insieme di problemi di test ha lo scopo di valutare la competitività del MASS che utilizza il protocollo di scheduling base (senza promesse); le istanze sono stati generate con i seguenti valori dei parametri:

- $N \in \{5, 10, 15, 20, 25, 30, 40, 50\}$
- $M \in \{1, 2, 3, 4, 5\}$
- $p_{min} = 1$, $p_{max} = 4$
- $\alpha \in \{0.25, 0.5, 1, 2\}$, $\beta = 2$, $\gamma = 4$.

Tali istanze sono caratterizzate dai seguenti insiemi di valori dei tre indici: $N/M \in [1.25, 50]$, $MRO \in [2, 11]$, $ARO \in [0.014, 0.20]$. Le istanze per le quali $MRO \leq M$ non sono state considerate poiché possono essere risolte banalmente, dal momento che i job non sono veramente in conflitto. Analogamente, anche le istante per le quali $ARO > 0.20$ sono state scartate, poiché è praticamente impossibile risolverle in modo ottimo entro i limiti di tempo imposti al software LINGO.

I risultati ottenuti sono riportati nella tabella seguente:

	Istanze	**Risolte in modo ottimo**	**ACR**
Insieme A	164	103 (62,8%)	1,50
Insieme B	53	32 (60,3%)	1,57
Totale	217	135 (62,2%)	1,52

Tabella 6-6.II Risultati dell'analisi di competitività

La tabella mostra il numero di istanze generate e quello delle istanze di cui LINGO ha trovato la soluzione ottima entro il tempo limite. L'ACR è calcolato sia per l'intero insieme di istanze, sia per i due sottoinsiemi A e B, dove B è caratterizzato da valori più alti dei tre indici: dunque A è un insieme di problemi "più facili" e B un insime di problemi "più difficili".

I risultati evidenziano come il MASS non sembra peggiorare le proprie prestazioni nella risoluzione di istanze classificate a priori come "più difficili" (in base ai valori dei tre indici).

Nella generazione dell'insieme di problemi indicati in precedenza, i pesi di earliness e tardiness sono stati posti tutti a 1 per tutti i job: successivamente, per

verificare l'efficienza del sistema di scheduling basato sulle promesse, sono stati generati nuovi problemi con pesi diversi per i vari job. I pesi sono stati generati casualmente tra un valore minimo compreso tra 1 e 5 ed un valore massimo compreso tra 5 e 10.

Per valutare meglio l'impatto dell'introduzione delle promesse, i problemi generati sono caratterizzati da una difficoltà maggiore rispetto a quelli precedenti (valori dei tre indici più alti).

ACR_1	**ACR_2**	**ACR_3**
2,69	1,89	1,74

I tre ACR indicano:

- ACR_1: rapporto di competitività del sistema di scheduling base
- ACR_2: rapporto di competitività del sistema di scheduling con le promesse, dove il criterio di annullamento è basato soltanto sui bid.
- ACR_3: come il precedente, ma con il criterio di annullamento basato anche sulle due date, oltre che sui bid.

Capitolo 7: Conclusioni e sviluppi futuri

7.1 Conclusioni

Introdotti nell'Intelligenza Artificiale Distribuita, gli agenti ed i sistemi multi-agente sono al centro di numerose ricerche nell'ambito *dell'information technology*, in particolare per quanto riguarda i problemi decisionali.

Una qualunque applicazione distribuita può essere efficacemente rappresentata come un insieme di agenti software cooperanti e un modello multi-agente può rappresentare un approccio euristico ai problemi di ottimizzazione, nel quale dal comportamento autonomo di singole entità decisionali deriva l'ottimizzazione di un obiettivo globale.

In questa tesi è stato applicato questo concetto sviluppando un modello MAS per lo scheduling di job in un contesto manifatturiero dinamico.

In definitiva è stato realizzato uno studio per l'utilizzo di architetture multi-agente nella risoluzione di problemi di controllo distribuito. Una volta definiti tutti gli agenti coinvolti nel sistema secondo uno standard comune (in questo caso lo standard FIPA), la logica decisionale è stata implementata attraverso il

comportamento dei singoli agenti, che rappresentano entità concrete (job e macchine) e comunicano tra loro tramite due possibili protocolli di negoziazione.

Allo stato attuale, il sistema di scheduling ad agenti fornisce buone prestazioni: ulteriori sviluppi futuri comprendono estensioni dell'algoritmo e introduzione di capacità di apprendimento negli agenti.

7.2 Sviluppi futuri

7.2.1 Miglioramento dell'algoritmo di scheduling

Una possibilità per migliorare il sistema di scheduling consiste nel modificare il comportamento degli agenti macchina.

Attualmente i MA offrono sempre ai job il miglior slot possibile, e iniziano ad emettere le conferme a partire dal miglior agente job offerente, anche se ciò può condurre a perdere la contrattazione con altri JA. È possibile introdurre nei MA una nuova regola: se gli slot proposti a due job sono sovrapposti "di poco" (ovvero per un certo periodo di tempo minore di una data soglia) allora l'agente macchina propone due soluzioni sub-ottime non sovrapposte.

Questa soluzione è migliore rispetto al normale comportamento dei MA, che porterebbe alla generazione di una soluzione ottima per un JA e una soluzione molto più lontana dall'ottimo per un altro JA.

Un'altra possibilità (valida soltanto per il protocollo esteso) consiste nel modificare i criteri di annullamento delle promesse da parte dell'agente macchina, in modo da tenere in considerazione anche la funzione obiettivo e il numero di job serviti oltre all'offerta in denaro.

7.2.2 Parametri e prestazioni

L'euristica di scheduling presentata è influenzata da una certa quantità di parametri, cioè quei valori che influenzano il comportamento dei job e delle macchine: il numero di questi parametri potrebbe sembrare a prima vista eccessivo. Tuttavia molti di essi possono essere fissati in maniera relativamente semplice e raramente modificati in seguito; inoltre tali parametri definiscono il comportamento di entità relativamente semplici (gli agenti) le cui azioni dovrebbero essere tracciabili ed analizzabili con semplicità, anche in una fase di prima sperimentazione, facendo riferimento all'interpretazione intuitiva (antropomorfa) di tale comportamento (per questo, parlando dei parametri, si parla di ansiosità, fretta, ecc.). Maggiore è l'autonomia, la proattività e la capacità di interagire che si vuole dare agli agenti, maggiori saranno le sfaccettature del loro carattere e quindi il numero di parametri che li caratterizzeranno.

Ad ogni modo, è evidente che dovrebbe essere possibile ottenere migliori prestazioni (ovvero schedule migliori) con un'appropriata fase di *tuning* di tali parametri. Le possibilità prese in esame sono due:

- sviluppare di una procedura di addestramento off-line
- incorporare nel MASS un sistema di self-tuning, ovvero dotare gli agenti di capacità di apprendimento.

Nel primo caso, il sistema di scheduling interagisce con un algoritmo di ottimizzazione basato su una *local search* non-lineare che tenta di minimizzare la funzione obiettivo modificando i valori dei diversi parametri.

Nel secondo scenario, il concetto di *apprendimento* è applicato al sistema di scheduling nella sua globalità piuttosto che ai suoi singoli elementi: non sono infatti i singoli agenti che imparano, ma la loro collettività o meglio la loro *specie* impara. Per questo forse si potrebbe parlare di *evoluzione* più che di apprendimento, e i parametri caratteristici potrebbero essere visti come una sorta di "codice genetico" degli agenti.

Tra le generazioni successive di agenti potrebbe dunque esistere un meccanismo di apprendimento in base al quale migliorare le prestazioni del sistema di scheduling risultante: gli agenti generatori (JGA e MGA) creerebbero ogni volta agenti con leggere variazioni dei parametri e regolarmente trasmetterebbero alle nuove generazioni di agenti i parametri che hanno ottenuto prestazioni migliori.

A loro volta, anche gli agenti generatori possono essere dotati di parametri caratteristici: ad esempio, un agente JGA può avere un parametro che rappresenta la probabilità di far nascere agenti JA con un comportamento "tranquillo" o "ansioso".

L'evoluzione (ossia la sequenza di variazione dei parametri) può essere:

- sincrona: i dati relativi al comportamento degli agenti di una specie possono essere conservati in database statici ed analizzati periodicamente (ad esempio utilizzando tecniche di *data mining*).
- asincrona: sono gli agenti generatori ad istanziarla quando lo desiderano
- continua: alla morte di una agente viene subito analizzato il suo comportamento in vita.

Quest'ultima possibilità dovrebbe essere messa in pratica in modo molto prudente per evitare variazioni troppo brusche nel comportamento di una specie (variazioni che renderebbero inutile l'analisi delle prestazioni, ovvero lo stesso meccanismo evolutivo): l'evoluzione, rispetto al periodo di vita degli agenti, dovrebbe impiegare tempi molto più lunghi (il comportamento di una specie dovrebbe essere modificalo solo sulla base di statistiche ben assestate).

Uno svantaggio di questo approccio potrebbe essere dato dal rischio di conservare a lungo un insieme di parametri che fornisce pessime prestazioni: questo inconveniente può essere risolto facendo evolvere il sistema degli agenti in maniera fittizia durante un periodo di tuning preliminare (simile in questo senso alla procedura di addestramento off-line descritta in precedenza) durante il quale

viene sottoposta al sistema di scheduling una sequenza di problemi di prova (test fuori linea), lasciando evolvere il sistema sino ad assestare un insieme di parametri con prestazioni accettabili.

L'evidente vantaggio di questa architettura è dato dalla sua elevata dinamicità, in grado di evolvere per adattarsi ai mutamenti della domanda del mercato – condizione essenziale per le moderne imprese manifatturiere.

Bibliografia

[Achliop00] D.Achlioptas, M.Chrobak, J.Noga "Competitive analysis of randomized paging algorithms", in Theoretical Computer Science 234 (2000) 203-218.

[Adiga93] S.Adiga, "Object-oriented software for Manufacturing Systems", Chapman & Hall, London, 1993.

[Agha86] G. Agha, "Actors: a model of concurrent computation in distributed systems", MIT Press, Cambridge, Mass., 1986

[Agnetis00] A. Agnetis, "Introduzione ai problemi di Scheduling", Dipartimento di Ingegneria dell'Informazione – Università di Siena, 2000.

[Baker90] Baker K.H. and Scudder G.D., "Sequencing with earliness and tardiness penalties: a review", Operations Research, 30 (1990) 22-36.

[Balak99] Balakrishnan N., Kanet J.J., Sridharan V., "Early/Tardy scheduling with sequence dependent setups on uniform parallel machines", Computers and Operations Research, 26, 2 (1999) 127-141

[Bauer00] B.Bauer, J.P.Müller, J.Odell, "Agent UML: A Formalism for Specifying Multiagent Interaction"
http://www.auml.org/auml/working/Bauer-AOSE2000.pdf

[Bauer01] B.Bauer, "UML Class Diagrams: Revisited in the Context of Agent-Based Systems", in "Proc. of Agent-Oriented Software Engineering (AOSE) 2001", Agents 2001, Montreal
http://www.auml.org/auml/working/Bauer-AOSE2001.pdf

[Bellman57] R.Bellman, "Dynamic Programming", Princeton University Press, Princeton, New Jersey (1957).

[Blazew96] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, J. Weglarz "Scheduling of Processor and Manufacturing Systems", Springer-Verlag, 1996

[Booch99] G.Booch, J.Rumbaugh, I.Jacobson, "The Unified Language User Guide", Addison-Wesley, Reading, MA, 1999.

[Cammarata] M. Cammarata, “Pubblica amministrazione e open source - Forse è incominciata una rivoluzione.” (2000) InterLex
http://www.interlex.it/pa/rivoluz.htm

[Cybele] “OpenCybele Agent Infrastructure - Users Guide”.
http://www.opencybele.org/docs/Users.pdf

[CybSurv] “A survey of agent infrastructures”, vedi [Cybele] sez. 8 pag. 73.

[Demaz00] Y.Demazeau, P.M. Ricordel, “From Analysis to Deployment: a Multi-Agent Platform Survey”, LEIBNIZ laboratory, 2000.
http://www-lia.deis.unibo.it/confs/ESAW00/pdf/ESAW07.pdf

[Deng00] X.Deng, N.Gu, T.Brecht, K.Lu “Preemptive Scheduling of Parallel Jobs on Multiprocessors”, Society for Industrial and Applied Mathematics (2000).

[DevOli] G.Devoto, G.C.Oli, “Il dizionario della lingua italiana”, (1990) Casa Editrice Felice Le Monnier S.p.A., Firenze

[Durfee91] E.H. Durfee, V.R. Lesser, “Partial Global Planning: A Coordination Framework for Distributed Hypothesis Formation” (1991) Dept. of EE and CS Dept. of Comp. and Info.
IEEE Transactions on Systems, Man, and Cybernetics

[Feigenbaum] E.A.Feigenbaum, "The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering," Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI 77), pp. 1014-29. 1977

[FIPA01] “FIPA Abstract Architecture Specification”
http://www.fipa.org/specs/fipa00001/

[FIPA25] “FIPA Interaction Protocol Library Specification”.
http://www.fipa.org/specs/fipa00025/

[FIPA37] “FIPA Communicative Act Library Specification”
http://www.fipa.org/specs/fipa00037/

[Fons01] S.Fonseca, M.Griss, R.Letsinger, “Agent Behaviour Architectures - A MAS Framework Comparison”. HPL Technical Report 2001-332.
http://hpl.hp.com/org/stl/maas/docs/HPL-2001-332.pdf

[Franklin96] S.Franklin, A.Graesser, “Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents”, Proceedings of the Third International

Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[French82] S. French. “Sequencing and Scheduling: an introduction to the mathematics of the job-shop”. Ellis Horwood Limited, 1982.

[Garey79] M. R. Garey and D. S. Johnson, “Computers and Intractability”. W.H. Freeman, New York, 1979.

[Garey98] Garey M.R., Tarjan R.E., Wilfong G.T., “One-processor scheduling with symmetric earliness and tardiness penalties”, Mathematics of Operations Research, 13 (1988) 330-348.

[Gou98] L.Gou, P.B.Luh, Y.Kyoya, “Holonic manufacturing scheduling: architecture, cooperation mechanism, and implementation”, Computers in Industry 37 (1998) 213-231

[Graham79] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, “Optimization and approximation in deterministic sequencing and scheduling: a survey”, Annals of Discrete Mathematics 4, 287-326.

[Hall86] Hall N.G., “Single and multiple-processor models for minimizing completion time variance”, Naval Research Logistics Quarterly, 33 (1986) 49-54.

[Hall91] Hall N.G., Kubiak W., Sethi S.P., “Earliness and tardiness scheduling problems, II: deviation of completion time about a restrictive common due date”, Operations Research, 39 (1991) 847-856.

[Hewitt77] C. Hewitt, “Viewing control structures as patterns of passing messages”, Journal of Artificial Intelligence, pp. 323-364, 8-3, June 1977

[Huhns99] Huhns, Stephens, “Multiagent Systems and Societies of Agents”, Multiagent Systems, 1999

[Jenn98] N.R.Jennings, M.Wooldridge, “Applications of Intelligent Agents”, In “Agent Technology: Foundations, Applications, and Markets”, N.R. Jennings and M.J Wooldridge (Eds.), Springer. (1998).

[JivComp] A.K.Galan, “Comparison of Existing Design Tools with JiVE”, in “JiVE: JAFMAS integrated Visual Environment”. http://www.ececs.uc.edu/~abaker/JiVE/Chapter2.html

[Kanet81] Kanet J.J., “Minimizing the average deviation of job completion times about a common due date”, Naval Research Logistics Quarterly, 28 (1981) 643-651.

[Maes96] P.Maes, A. Wexelblat, CHI 96 Tutorial, "Software Agents", 1996
Ulteriori documenti sulle attività di ricerca di Pattie Maes si trovano presso:
http://pattie.www.media.mit.edu/people/pattie/

[McCarthy] J.McCarthy, “History of LISP”. ACM SIGPLAN Notices, Vol. 13 (1978), No. 8, p.217-223.
http://www-formal.stanford.edu/jmc/history/lisp/lisp.html

[Moscardi] S.Moscardi, “Caratteristiche e dinamiche del sistema distributivo giapponese”, Facoltà di Economia, Università degli Studi di Firenze, A.A.1999-2000

[Nemhauser] Nemhauser G.L., Wolsey L.A., “Integer and combinatorial optimization”, John Wiley, Chichester, 1988.

[Nwana96] H.S.Nwana, “Software Agents: an Overview”, Knowledge Engineering Review, Vol. 11, No 3, pp.1-40, Sept 1996. Cambridge University Press, 1996

[Odell00] J.Odell, H. Van Dyke Parunak, B.Bauer, “Extending UML for Agents”, AOIS Workshop at AAAI 2000.
http://www.jamesodell.com/ExtendingUML.pdf

[Oliveira] E.Oliveira, K.Fischer, O.Stepankova, “Multi-agent systems: which research for which applications”, Robotics and Autonomous Systems 27 (1999) 91-106.

[Petrovic02] S.Petrovic, “Introduction to scheduling” in Automated Scheduling, School of Computer Science and IT, University of Nottingham, 2002.

[Russel95] S.Russell, P.Norvig. “Artificial Intelligence: A Modern Approach”, Englewood Cliffs: Prentice-Hall, Inc. 1995.
Ulteriori informazioni su Russel e Norvig possono essere trovate presso:
http://http.cs.berkeley.edu/~russell/aima.html

[Satapathy] G.Satapathy, S. Kumara, “The object oriented design based agent modeling”, in the Proceedings of the Fourth Applications of Agents and Multiagents, London, UK, 1999.

[Schmul99] J.Schmuller, “Teach Yourself UML in 24 Hours”, Sams Publishing, 1999

[Sgall98] Sgall J., “On-line scheduling – a survey”, in On-line Algorithms, Fiat A., Woeginger G.J. (Eds.), Springer-Verlag, 1998.

[Shen99] W.Shen, D.H. Norrie, “Agent-Based Systems for Intelligent Manufacturing: A State-of-the-Art Survey”, Knowledge and Information Systems, an Int. Jour. 1, 2, (1999) 129-156.

[Shmoys95] Shmoys D.B., Wein J., Williamson, “Scheduling parallel machines on-line”, SIAM Jornal on Computing, 24, 6 (1995) 1312-1331.

[Shoham] Y.Shoham, “Agent-oriented programming”, Artificial Intelligence, 60(1), Pages 51--92, 1993

[Sivrikaya99] Sivrikaya-Serifoglu F., and Ulusoy G., “Parallel machine scheduling with earliness and tardiness penalties”, Computers and Operations Research, 26 (1999) 773-787.

[Sleator85] D. Sleator, R.E. Tarjan, “Amortized efficiency of list update and paging rules, Communications of the ACM, 28 (1985) 202-208.

[Wool95] M.Wooldridge, N.R. Jennings, “Intelligent agents: Theory and practise”, The Knowledge Engineering Review 10, 2 (1995) 115-152.

[Wool99] M.Wooldridge, “Intelligent Agents”, in “Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence”, G. Weiss (Ed.), MIT Press, Cambridge, MA. (1999).

[Yeh95] T.Yeh, C.Kuo, C.Lei, and H.Yen “Competitive Analysis of On-Line Disk Scheduling”, Department of Electrical Engineering, National Taiwan University (1995).

[Zhu00] Zhu Z., Heady R.B., “Minimizing the sum of earliness/tardiness in multi-machine scheduling: a mixed integer programming approach”, Computers & Industrial Engineering, 38 (2000) 297-305.

Indice