Indice

Introduzione

La continua crescita tecnologica spinge la ricerca alla sperimentazione ed allo sviluppo di applicazioni in grado di manifestare un comportamento intelligente sempre più simile a quello umano.
Un esempio consiste in applicazioni che eseguono la diagnosi e la riparazione di sistemi complessi, sia hardware che software. Tali applicazioni possono essere costituite da diverse componenti, ogni componente ha una conoscenza specifica della parte di sistema che deve diagnosticare o riparare e coopera con le altre componenti dell'applicazione al fine di mantenere il sistema funzionante.
Un altro possibile esempio è quello di un pilota automatico che assume il controllo dell'aeroplano con lo scopo di condurlo all'aeroporto di destinazione. Il pilota automatico conduce il volo autonomamente senza l'intervento di esseri umani, in modo da perseguire il proprio scopo ed eventualmente generando nuove strategie per reagire agli imprevisti che si manifestano durante il volo, come portare l'aeroplano ad una certa quota per evitare una tempesta oppure modificarne la rotta.
Altre applicazioni di questo tipo potrebbero essere "segretari elettronici" che gestiscono l'agenda degli utenti proponendo nuovi appuntamenti ad altri "segretari elettronici". Un altro infine sono le applicazioni in grado di guidare robots all'interno di particolari ambienti in modo di evitare ostacoli.

Nell'ambito dell'intelligenza artificiale e del software engineering, una direzione di ricerca, si è indirizzata allo sviluppo di sistemi software e/o hardware in grado di operare autonomamente, esibire un comportamento goal-directed, eseguire azioni che scaturiscono da ragionamenti complessi basati sulle conoscenze che il sistema ha del mondo in cui è collocato, cooperare con altre entità che hanno un simile comportamento.
Entità che hanno queste caratteristiche vengono chiamate **agenti**.

In [BKJT95] ed in [SdAMP96] vengono analizzati rispettivamente un esempio reale di sistema ad agenti che monitorizza e diagnostica i guasti di una rete di trasporto dell'energia elettrica in Spagna ed una architettura per agenti specializzati nella diagnosi di sistemi.

Il sistema di monitoraggio della rete per il trasporto di energia elettrica è composto da diversi agenti che cooperano al fine di monitorizare la rete, rilevare i guasti ed eventualmente ripararli. L'agente CSI (Control System Interface) riceve continuamente dati dalla rete, ad esempio messaggi riguardanti il manifestarsi di eventi anomali. In base a questi dati l'agente CSI produce una rappresentazione dello stato della rete e ne effettua una analisi preliminare al fine di rilevare la presenza di eventuali guasti. Il rilevamento di un guasto provoca l'attivazione di altri due agenti: gli agenti AAA (Alarm Analysis Agent) e BAI (Blackout Area Identifier). L'agente BAI determina la zona interessata dal guasto mentre l'agente AAA genera un insieme di ipotesi sulle cause del guasto e tramite l'interazione con gli agenti CSI e BAI raffina tale insieme fino a determinare la causa del guasto. Infine l'agente SRA (Service Restoration Agent) genera un piano di azioni che può essere utilizzato per riparare il guasto.
Ogni agente descritto in [SdAMP96] ha una base delle conoscenze che permette la rappresentazione dello stato del sistema da diagnosticare ed un motore inferenziale per risolvere eventuali anomalie. Il motore inferenziale è in grado di effettuare dei test sul sistema e di rilevare eventuali incoerenze con la rappresentazione mantenuta nella base delle conoscenze dell'agente, quindi di reagire per eliminare tali incoerenze.
In [LLFL+95] viene mostrato un semplice esempio di sistema ad agenti in grado di organizzare riunioni per conto di un utente. Ogni utente possiede un agente "schedule manager" che conosce le attività che impegnano l'utente. Quando l'utente vuole organizzare una riunione attiva un agente "meeting scheduler" che contatta gli agenti "schedule manager" dei partecipanti e prova ad organizzare la riunione per un giorno ed un'ora in accordo con tutti i partecipanti.
Semplici esempi di robots che devono eseguire i propri compiti in ambienti particolari vengono trattati in [Rao96] e [HdBvdHjCM98]. Nel primo caso il robot ha una rappresentazione dell'ambiente in cui è collocato, è in grado di percepire i cambiamenti che si manifestano e di reagire a tali cambiamenti. Il comportamento del robot all'interno dell'ambiente non dipende solo dagli eventi che percepisce ma anche dalla rappresentazione interna degli obiettivi che deve completare.
Nel secondo caso il robot non è in grado di percepire gli eventi che si manifestano all'esterno e per completare i propri obiettivi utilizza solo la rappresentazione interna dell'ambiente.

Definizione di agente

Noonostante le applicazioni descritte siano alquanto diverse tra loro, sia per quanto riguarda il dominio applicativo che per lo scopo finale, gli agenti che le compongono presentano caratteristiche comuni [WJ95]:

- *autonomia*: l'agente è in grado di operare senza l'intervento di entità esterne ed ha il controllo delle azioni che compie e del proprio stato interno;

- *reattività*: l'agente percepisce l'ambiente in cui è collocato e risponde ai cambiamenti che si verificano in esso modificando il proprio stato interno ed eseguendo azioni;

- *proattività*: l'agente non compie azioni e non modifica il proprio stato solo in funzione dei cambiamenti dell'ambiente in cui è collocato, ma pianifica tali azioni e modifica tale stato al fine di perseguire uno scopo a lungo termine; in questo senso l'agente ha un comportamento goal-directed;

- *abilità sociale*: l'agente è in grado di comunicare con altri agenti al fine di cooperare e/o competere con essi per portare a termine il proprio scopo.

Queste proprietà sono descritte informalmente e sono abbastanza deboli da poter applicare il concetto di agente anche a sistemi software e hardware che non corrispondono propriamente ad agenti. Quello che caratterizza il concetto di agente, oltre alle proprietà precedenti, è la complessità del comportamento assunto durante la sua attività: il comportamento dell'agente non consiste solamente nell'esecuzione di azioni ma nell'avere controllo sul proprio stato e nel prendere l'iniziativa per raggiungere i propri obiettivi utilizzando strategie apropriate.

Per alcuni ricercatori che operano nel campo dell'intelligenza artificiale il termine agente ha un significato più specifico che quello caratterizzato nella definizione debole. Da questo punto di vista un agente è una entità che in aggiunta alle caratteristiche menzionate nella definizione debole, cioè autonomia, reattività, proattività e abilità sociale, può essere concettualizzato attraverso qualità mentali generalmente utilizzate per descrivere il comportamento umano. Tali qualità, secondo [WJ95] possono essere divise in due categorie:

- Qualità utilizzate per rappresentare l'ambiente in cui l'agente è collocato, cioè il mondo esterno:
  - credenza;
  - conoscenza.
- Qualità utilizzate per rappresentare il modo in cui l'agente agisce:
  - desiderio;
  - intenzione;

– obbligo;
– bisogno;
– scelta.

Per ottenere una descrizione esaustiva del comportamento dell'agente è necessario utilizzare almeno una qualità per ogni categoria: almeno una qualità mentale per rappresentare l'ambiente in cui l'agente è collocato ed almeno una qualità mentale per rappresentare il modo in cui agisce.
Le due categorie non sono concettualmente separate in quanto ad esempio il modo in cui un agente agisce dipende direttamente dalle proprie credenze e tali credenze sono modificate dagli effetti delle azioni svolte.

Agenti ed altre tecnologie

L'utilizzo delle entità caratterizzate nella sezione per realizzare applicazioni complesse presuppone la necessità di realizzare sistemi formati da componenti indipendenti, ognuna con capacità di iniziativa, ragionamento, comunicazione.
La tecnologia basata sugli agenti nasce dalla sinergia tra diverse tecnologie informatiche non necessariamente appartenenti al campo dell'Intelligenza Artificiale.

Agenti e sistemi distribuiti

Un sistema distribuito è un insieme di calcolatori con sistemi operativi indipendenti, connessi tra loro al fine di rendere condivisibile l'accesso a diverse risorse (software e hardware) in maniera trasparente per tutti gli utenti [Kra94].
Affinchè le risorse condivise dagli utenti possano essere accedute concorrentemente e trasparentemente il sistema deve essere fornito di componenti software indipendenti che non condividono memoria ma sono capaci di interagire tra loro. Ogni componente è provvista di una interfaccia che utilizza durante l'interazione e che contiene informazioni riguardanti i servizi offerti e quelli che può richiedere.
Le componenti possono essere definite come tipi, dai quali vengono create delle istanze allocate ai calcolatori che compongono il sistema; un programma distribuito viene quindi descritto dalla configurazione relativa all'interconnessione tra le istanze delle componenti.
La tecnologia relativa ai sistemi multi-agente è simile a quella relativa ai sistemi distribuiti: ha lo scopo di fornire servizi distribuiti in un particolare ambiente; il sistema è formato da componenti distinte (gli agenti) che interagiscono tra loro attraverso un sistema di comunicazione; il comportamento del sistema può essere specificato da un linguaggio che descrive l'evoluzione dello stato degli agenti. Tuttavia la differenza sostanziale tra il concetto di componente utilizzato nei sistemi

distribuiti ed il concetto di agente consiste nella complessità del programma che li definisce: l'autonomia dell'agente non consiste nella semplice capacità di eseguire azioni, ma in un comportamento complesso basato sul concetto di iniziativa nell'eseguire azioni e nel controllare lo stato interno dell'agente; la proattività non consiste solamente nell'assolvere le richieste ottenute, ma pianificare delle strategie e modificare il proprio comportamento per eseguire tali richieste.

Agenti e oggetti

Un oggetto è un'entità costituita da uno stato e da una collezione di operazioni dette metodi. L'esecuzione di tali operazioni ha come effetto la modifica dello stato dell'oggetto stesso oppure l'invio di opportuni messaggi ad altri oggetti.
Un agente è necessariamente definito da uno stato e da operazioni con il fine di modificare tale stato. Deve inoltre avere la capacità di scambiare messaggi con altri agenti. Da questo punto di vista il concetto di agente sembra avere una relazione molto stretta con quello di oggetto, ma l'agente può avere un comportamento goal-directed, deve essere quindi in grado di pianificare le azioni che permettono di perseguire il proprio obiettivo [Woo97]; lo stato interno è generalmente più complesso di quello di un oggetto: l'agente può essere composto a sua volta da altre componenti ognuna specializzata nell'eseguire compiti oppure nell'immagazzinare dati. Inoltre i servizi offerti da un oggetto vengono richiesti tramite messaggi (chiamate di metodi) specifici per quell'oggetto, in un sistema in cui più agenti interagiscono tra loro, invece, il linguaggio di comunicazione deve essere lo stesso per tutti gli agenti che ne fanno parte.
Queste capacità sembrano non rientrare tra quelle di un oggetto rendendo quindi il concetto di agente una estensione del concetto di oggetto.

Agenti e sistemi esperti

Un sistema esperto è composto essenzialmente da tre componenti: la shell, la base della conoscenza ed il motore inferenziale. La shell è un'interfaccia tra il sistema esperto ed un utente umano. In un primo momento la shell permette l'acquisizione della conoscenza, che viene rappresentata nella base della conoscenza, ed in seguito la sottomissione di una query formulata dall'utente al motore inferenziale. Il motore inferenziale costruisce una risposta alla query formulata e tramite la shell viene visualizzata all'utente.
Il sistema esperto può essere visto come un agente molto primitivo che interagisce solo con l'utente tramite la shell. Il sistema esperto non ha capacità proattiva: infatti non ha la capacità di modificare il proprio stato né di pianificare azioni che non siano quella di fornire la risposta alla query che sta processando. Non ha inoltre la capacita di competere o cooperare con eventuali altri sistemi ma solo

con l'utente umano, inoltre il sistema esperto è un'entità con "disembodied intelligence", cioè non è collocato in un ambiente, è intelligenza senza corpo, mentre l'agente è collocato in un ambiente, lo percepisce ed eventualmente reagisce a tali percezioni.

Gli agenti ereditano dai sistemi esperti le tecniche di rappresentazione della conoscenza e la capacità di eseguire "ragionamenti" complessi basati sulle tali conoscenze, questa capacità definisce il comportamento proattivo dell'agente e quindi l'attitudine a sviluppare opportune strategie per raggiungere i propri obiettivi.

La realizzazione dei sistemi ad agenti

Il processo di sviluppo del software comincia con l'acquisizione dei requisiti che il sistema deve possedere; tali requisiti servono per la realizzazione della specifica formale cioè del modello astratto che definisce le funzionalità del sistema.
Il linguaggio con cui viene realizzata la specifica deve essere in grado di esprimere i requisiti che l'applicazione deve avere, nel caso di sistemi multi-agente i requisiti considerati derivano da ciò che caratterizza la definizione di agente, inoltre le proprietà che un linguaggio di specifica per agenti deve avere sono:

- La struttura del linguaggio deve far sì che la specifica di un sistema sia modulare e facilmente modificabile, in modo che eventuali modifiche del sistema possano essere realizzate con semplicità;
- Il linguaggio di specifica deve avere una semantica "chiara" per poter realizzare un sistema corretto e poter descriverne il comportamento;
- la specifica deve essere eseguibile oppure animabile, in modo da poter effettuare dei test sul prototipo del sistema ed eventualmente effettuare delle modifiche.

Un nuovo linguaggio di specifica per sistemi multi-agente

Un sistema multi-agente è caratterizzato dall'eterogeneità degli agenti che lo compongono: ogni agente ha conoscenze specifiche proprie, strategie di comportamento diverse, assolve compiti diversi.
Dal punto di vista della programmazione la struttura interna dell'agente, cioè l'architettura, consiste in un insieme di strutture dati (componenti), di operazioni su tali strutture dati e di operazioni per la ricezione e la comunicazione del'informazione.
Le operazioni sulle componenti consistono nell'inserimento, nella eliminazione e

nella lettura delle informazioni utilizzate dall'agente durante la sua attività. Le operazioni riguardanti la comunicazione dei dati sono utilizzate per la ricezione e l'invio dei messaggi ad eventuali altri agenti del sistema e per la percezione degli eventi che si manifestano nell'ambiente in cui l'agente è collocato.
Questa tesi ha lo scopo di realizzare un linguaggio di specifica per sistemi multi-agente in grado di descrivere le diversità che caratterizzano gli agenti in modo omogeneo attraverso la realizzazione di un linguaggio astratto, cioè indipendente da un particolare dominio applicativo e da una particolare architettura, con una struttura sintattica modulare che permetta la modifica del sistema in modo incrementale e con una semantica che renda possibile la realizzazione di specifiche corrette ed eventualmente animabili.

Organizzazione della tesi

Il documento è diviso in due parti: la prima parte introduce i concetti di agente, architettura e linguaggio di specifica per agenti, la seconda affronta la realizzazione di un nuovo linguaggio di specifica per sistemi multi-agente.

Il primo capitolo della tesi introduce il concetto di architettura, concetto da cui non si può prescindere nella realizzazione di applicazioni basate su agenti, al fine di fornire una breve panoramica sulle diverse architetture sviluppate nell'ambito della ricerca sugli agenti.
Il secondo capitolo consiste in una rassegna di linguaggi di specifica per agenti. La rassegna focalizza l'attenzione su i diversi formalismi che caratterizzano i linguaggi e sulla relazione che hanno con la definizione debole di agente.
Il terzo capitolo contiene le idee su cui si basa il linguaggio proposto nella tesi.
Nel quarto viene descritta la sintassi del linguaggio, facendo riferimento ad un esempio di applicazione reale.
Nel quinto capitolo viene descritta la semantica operazionale delle istruzioni del linguaggio e viene fornito un sistema di transizioni tra configurazioni del MAS.
La tesi si conclude con due appendici nelle quali vengono descritte, la sintassi formale del linguaggio e l'esempio di applicazione reale a cui vine fatto riferimento nel quarto capitolo.

Parte I

Architetture e linguaggi di specifica per sistemi multi-agente

Capitolo 1

Architetture per agenti

La descrizione dei moduli che compongono un agente, come tali moduli sono realizzati, e come cooperano, costituisce l'architettura di un agente. L'architettura di un agente descrive quindi se il comportamento è più reattivo o proattivo, quali strutture dati sono utilizzate per rappresentare lo stato dell'agente, quali strategie permettono lo sviluppo di ragionamenti complessi e/o l'interpretazione degli eventi dell'ambiente. L'architettura determina inoltre se l'agente è in grado oppure no di comunicare con altri agenti ed infine quale protocollo di comunicazione adotta nel caso in cui avvenga la comunicazione.
Secondo Jorg P. Müller [Mul97b] gli agenti possono essere realizzati attraverso quattro tipi di architetture:

- *Agenti reattivi*: gli agenti realizzati secondo tale architettura manifestano comportamento reattivo ma non proattivo, percepiscono quindi l'ambiente in cui sono immersi e compiono azioni dipendenti dal cambiamento di tale ambiente attraverso semplici regole della forma:

  situazione $\rightarrow$ *azione*.

  Non hanno una rappresentazione interna dell'ambiente, non hanno quindi credenze né conoscenze riguardo all'ambiente né riguardo al proprio stato interno impedendo così pianificazioni complesse di azioni.

- *Agenti deliberativi*: tali agenti hanno una rappresentazione simbolica dell'ambiente in cui sono immersi, possiedono una *componente inferenziale* che permette la pianificazione di azioni sfruttando la rappresentazione interna dell'ambiente, non hanno comportamento reattivo, ma solo proattivo.

- *Agenti interagenti*: Sono agenti in grado di coordinare la loro attività con quella di altri agenti attraverso un qualche tipo di comunicazione.

- *Agenti ibridi*: gli agenti che appartengono a questa classe sono generalmente costituiti da due o più strati gerarchicamente organizzati; l'interazione tra questi strati descrive il comportamento dell'agente. L'agente può essere composto da strati reattivi e da strati deliberativi in grado di cooperare e/o competere tra loro.
  Tuttavia in [KS96] viene mostrata una architettura in cui l'agente, pur non essendo composto da strati, è in grado di manifestare un comportamento ibrido.

  I vantaggi che comporta tale architettura sono:

  1. Differenti funzionalità dell'agente sono separate e collegate da interfacce ben definite.
  2. Il progetto dell'agente è più compatto e robusto, il debug è facilitato.
  3. I differenti strati possono essere eseguiti parallelamente, in questo modo l'agente durante la pianificazione di una sequenza di azioni(per la quale può essere impiegato molto tempo) può tempestivamente reagire a eventuali cambiamenti dell'ambiente.
  4. Poichè al fine di realizzare differenti tipi di funzionalità sono richiesti diversi tipi di conoscenze, ad ogni strato possono essere associate solo le conoscenze necessarie.

Vediamo ora alcuni esempi di architetture per agenti tratti da [Mul97b].

1.1 Architetture per agenti reattivi

1.1.1 Brooks: sussunzione

Gli agenti sono decomposti in strati organizzati gerarchicamente che agiscono indipendentemente e parallelamente. Gli strati che hanno priorità più bassa sono utilizzati per realizzare comportamenti di base (nel caso si tratti di robots, muoversi all'interno di una data area ed evitare ostacoli) mentre quelli con più alta priorità vengono utilizzati per perseguire semplici obiettivi che non necessitano di ragionamenti complessi (tali agenti non hanno capacità proattiva) come cercare e raccogliere oggetti.
Gli strati più ad alto livello di priorità **sussumono** il ruolo degli strati a più basso livello. Tali strati sono in grado di sostituire gli inputs e di rimuovere gli outputs degli strati con priorità più bassa, determinando in questo modo il comportamento dell'agente.

Questa architettura viene utilizzata con successo nella costruzione di robots i cui compiti consistono nella costruzione di mappe oppure nell'esplorazione di ambienti, mentre tali agenti sono incapaci di portare a termine compiti che richiedono una qualche forma di ragionamento deduttivo o di cooperazione.

1.1.2 Maes: azione-selezione dinamica

Il comportamento dell'agente è conseguenza dell'attivazione e della disattivazione dinamica delle azioni che può compiere. Un agente è descritto da un insieme di *moduli di competenza* e ogni modulo è descritto da:

- precondizioni;
- effetti;
- un livello di attivazione.

Tali moduli costituiscono una rete i cui collegamenti possono essere di tre tipi:

- il collegamento *successore* tra i moduli a e b (a $\overrightarrow{s}$ b) denota il fatto che a produce la precondizione per b.
- il collegamento predecessore tra i moduli a e b è definito da : a $\overrightarrow{p}$ b se e solo se b $\overrightarrow{s}$ a.
- il collegamento distruttore a $\overrightarrow{d}$ b descrive l'azione di distruzione della precondizione di b da parte di a.

Questi tre tipi di collegamento definiscono il modo in cui i moduli vengono attivati o disattivati: il successore viene utilizzato per attivare altri moduli, il predecessore per realizzare un meccanismo di backtracking, mentre il distruttore risolve eventuali conflitti tra moduli. Il livello di attivazione viene utilizzato per evitare disabilitazioni cicliche tra moduli: un modulo non ha la possibilità di disattivare moduli con livello di attivazione maggiore, mentre può disattivare moduli con livello di attivazione minore. Alcuni messaggi tra moduli incrementano il *valore di attivazione*, quando questo valore supera il livello di attivazione e le precondizioni del modulo sono soddisfatte, allora tale modulo compie una azione.

1.1.3 Agenti reattivi e comportamento complesso

Per quanto un agente reattivo non abbia la capacità di manifestare un comportamento complesso, a causa della mancanza di proattività ed autonomia tra le sue qualitá, in un sistema composto da più agenti reattivi che interagiscono tra loro può emergere un comportamento “intelligente” tipico dei ragionamenti complessi. Un esempio è lo scenario in cui un robot ha il compito di raccogliere campioni di un certo materiale dislocati in una data area e di riportarli alla base. Tale robot utilizza regole molto semplici per indicare ad altri robots le regioni con maggiore densità di campioni: se un robot carica un campione lascia dei residui di materiale lungo il cammino verso la base, se un robots non ha ancora caricato nulla, e trova dei residui lungo il cammino li carica e li porta alla base. In questo modo vengono individuati dei cammini privilegiati che conducono i robots alle regioni con maggiore densità di campioni.
Il sistema di robots descritto in questo scenario manifesta un comportamento "intelligente" in quanto è in grado di sfruttare i giacimenti con maggior densità di materiale. Se tale compito fosse stato eseguito da un singolo robot le regioni visitate sarebbero state, probabilmente, quelle più vicine alla base e la scoperta delle regioni migliori sarebbe stata casuale.
Il comportamento “intelligente”, in questo caso, è una caratteristica del sistema e non del singolo robot ed è dovuto alle regole, che per quanto semplici, permettono l’interazione e la cooperazione seppur involontaria tra gli agenti.

1.2 Architetture per agenti deliberativi

1.2.1 Bratman et al.: IRMA.

L’agente pianifica le proprie azioni basandosi su una rappresentazione esplicita delle proprie percezioni, credenze, desideri, intenzioni. L’agente realizzato secondo questa architettura è composto da quattro moduli:

- una *struttura intenzionale*, cioè una libreria di piani per compiere azioni;
- un *analizzatore di opportunità* e un *ragionatore* capaci di suggerire opzioni basate sulle credenze e sull’obiettivo dell’agente, per compiere azioni ogniqualvolta tali credenze sono aggiornate da qualche percezione;
- un *processo di filtraggio* che filtra le opzioni ottenute in modo che siano consistenti con il sistema intenzionale;
- un *processo deliberativo* che adotta un piano di azioni dalla *struttura intenzionale* in base alle opzioni che hanno superato il processo di filtraggio.

Un piano di azioni appartenente alla struttura intenzionale rappresenta le credenze che l'agente ha riguardo le intenzioni che possono essere adottate, mentre i piani ottenuti dal *processo deliberativo* definiscono le intenzioni che saranno intraprese.

Le architetture basate sulle qualità mentali *credenze, desideri, intenzioni* sono chiamate **BDI** (belifes, desires, intentions) e permettono la realizzazione di agenti in grado di manifestare un comportamento "intelligente" in cui vengono risolte situazioni complesse. Piuttosto che un modello formale per il modello BDI, IRMA è una struttura di controllo per gli agenti deliberativi realizzati secondo lo stile BDI.

Rao e Georgeff hanno sviluppato un interprete astratto che descrive il controllo di un agente realizzato secondo l'architettura degli agenti BDI di Bratman (figura 1.1):

repeat

> opzioni := generatore-opzioni(coda-eventi);
> opzioni-estese := deliberatore(opzioni);
> aggiorna-intenzioni(opzioni-estese);
> esegui-intenzioni();
> percepisci-evento-esterno();
> elimina-intenzioni-eseguite();
> elimina-intenzioni-impossibili();
> elimina-opzioni-impossibili();

end repeat

Figura 1.1: Interprete dell'architettura BDI.

Ad ogni ciclo l'interprete controlla la coda degli eventi, ed in base a questi genera gli obiettivi (opzioni) che dipendono dallo stato dell'ambiente, tali opzioni vengono poi trasformate in intenzioni e quindi eseguite. Il ciclo termina con la percezione di nuovi eventi dall'ambiente, con l'eliminazione delle intenzioni la cui esecuzione ha avuto successo, quelle la cui esecuzione è impossibile, ed infine eliminando le opzioni (desideri) impossibili.

1.3 Architetture per agenti interagenti

L'intelligenza artificiale distribuita (DAI) riguarda lo studio di agenti intelligenti organizzati in sistemi distribuiti. Da questo punto di vista i meccanismi di coordinamento e di cooperazione tra agenti autonomi sono più importanti delle strutture interne di tali agenti.
Nel seguito sono mostrati quattro approcci a tale architettura.

1.3.1 Fisher: MAGSY

Un agente MAGSY consiste in un insieme di fatti che rappresentano la conoscenza locale dell'agente, un insieme di regole che rappresentano la conoscenza procedurale ed il comportamento, un insieme di servizi che rappresentano l'interfaccia dell'agente; inoltre l'agente MAGSY può comunicare con altri agenti al fine di cooperare. Tramite questa architettura possono essere realizzati sistemi multi-agente definendo eventuali protocolli di comunicazione tra agenti.

1.3.2 Jennings: GRATE*

L'obiettivo di tale architettura è quello di rendere possibile la cooperazione tra sistemi intelligenti indipendenti attraverso un apposito strato che permette la comunicazione tra essi, memorizzando le informazioni degli agenti in specifiche strutture dati. Tale architettura permette la realizzazione di problem solvers interagenti, in quanto la capacità di soluzione di problemi è estesa dalla condivisione di informazione tra gli agenti del sistema.
GRATE* è costituita da *strati di dominio* che garantiscono le funzionalità necessarie al funzionamento di ogni singolo problem solver, e da uno *strato di controllo e cooperazione* che funziona da meta-controllore in modo da coordinare l'attività dei vari *strati di dominio* del sistema. Lo *strato di controllo e cooperazione* è diviso in tre moduli: il modulo di controllo è responsabile della pianificazione e esecuzione degli obiettivi locali, il modulo di cooperazione gestisce la cooperazione e la coordinazione tra agenti, il modulo di accertamento delle situazioni è responsabile della scelta tra procedure locali o coordinate con altri agenti per la soluzione dei problemi.

1.3.3 Steiner et al.: MECCA

Nell'architettura MECCA il comportamento di un agente è descritto da quattro processi che vengono eseguiti parallelamente:

- il *processo di attivazione del goal* che percepisce situazioni rilevanti dall'ambiente, ed in base a queste genera un goal che diverrà input per il processo di pianificazione;

- il *processo di pianificazione* che genera un insieme parzialmente ordinato di possibili sequenze di azioni che l'agente può eseguire;

- il *processo di schedulazione* che serializza l'insieme parzialmente ordinato di azioni ottenuto dal processo pianificatore, associando loro un ordine temporale;

- il *processo di esecuzione* il quale esegue, negli istanti stabiliti, le azioni ottenute dallo schedulatore.

Il comportamento dell'agente può essere coordinato dall'azione di altri agenti che condividono goals, oppure dall'esecuzione sincronizzata dei piani con altri agenti. Al fine di cooperare gli agenti fanno uso di primitive di comunicazione quali *INFORM, PROPOSE, ACCEPT* che sono trattate dal processo pianificatore come azioni, in questo modo il processo pianificatore crea sequenze di azioni che l'agente può eseguire tenendo conto delle comunicazione con altri agenti al fine di perseguire un obiettivo.

1.3.4 Sundermeyer et al.: COSY

Nell'architettura COSY un agente viene descritto da *comportamento*, *risorse* ed *intenzioni*.
Il *comportamento* di un agente può essere classificato in percettivo, cognitivo, comunicativo ed esecutivo, ognuno dei quali è realizzato da uno specifico modulo nell'architettura.
Le *risorse* includono risorse cognitive come credenze e conoscenze, oppure protocolli di comunicazione.
Le intenzioni modellano goals, preferenze, ruoli, responsabilità, tattiche che sono legate alle azioni che verranno intraprese dall'agente.
Oltre ai moduli di esecuzione, comunicazione e percezione, vi sono il modulo delle motivazioni, che implementa le strategie di un agente, ed il modulo cognitivo che valuta la situazione corrente, selezionando ed eseguendo azioni in quella situazione. Le azioni che possono essere intraprese dall'agente appartengono a due tipi di piani: piani di tipo *scripts* che descrivono sequenze stereotipali di azioni con il fine di realizzare goals, e i *protocolli di comunicazione* che definiscono come gli agenti comunicano tra loro.

1.4 Architetture per agenti ibridi

1.4.1 Firby et al.: azioni reattive

Questa architettura descrive agenti composti da tre moduli: un *pianificatore*, un *esecutore*, ed un *controllore*.
Il *pianificatore* produce un piano di azioni approssimativo selezionando un elemento da una libreria di piani in base allo stato dell'ambiente.
L'*esecutore* mette in atto il piano ottenuto dal pianificatore espandendolo a runtime in modo che la sequenza di azioni prodotta sia maggiormente dettagliata di quella ottenuta dal pianificatore. L'interazione di questi due moduli semplifica e rende più efficiente l'operato del pianificatore permettendo all'agente l'esecuzione di azioni in tempo reale. Nel caso in cui la sequenza di azioni prodotta dall'esecutore risultasse scorretta l'esecutore è in grado di percepire il fallimento dell'obiettivo e quindi di selezionare una sequenza di azioni alternativa.
Il *controllore* fornisce due tipi di procedure che sono attivate dall'esecutore ed il cui risultato è acquisito dall'esecutore stesso: una routine di percezione usata per aggiornare lo stato interno che rappresenta l'ambiente e la procedura di comportamento che influisce fisicamente sull'ambiente. Nel caso di un robot potrebbe gestire il movimento evitando collisioni con gli oggetti.

1.4.2 Lyons et al.: pianificazione e reazione

Questa architettura descrive agenti formati da tre moduli: un *pianificatore*, un *reattore*, e l'*ambiente circostante* che viene considerato parte del sistema. In questo tipo di agenti la componente reattiva realizzata dal reattore viene eseguita parallelamente alla componente deliberativa realizzata dal pianificatore, e quest'ultima componente fa sì che il comportamento del reattore sia concorde con l'obiettivo dell'agente.
Il *reattore* consiste di un insieme di processi che percepiscono l'ambiente e da un insieme di processi che compiono azioni associate ad opportuni eventi nell'ambiente. Tali processi agiscono in tempo reale ed parallelamente dal pianificatore.
Il *pianificatore* controlla che il comportamento del reattore sia in accordo con l'obiettivo dell'agente, basandosi sulla rappresentazione interna dell'ambiente, dello stato del reattore e dell'obiettivo che deve perseguire. Nel caso questo non accada il pianificatore acquisisce dati dai processi di percezione del reattore in modo da predire lo stato futuro dell'ambiente ed elimina oppure aggiunge azioni al reattore con il fine di ottenere la coerenza rispetto al goal.

1.4.3 Ferguson: Touring machines

Gli agenti realizzati secondo questa architettura sono costituiti da tre strati: *strato reattivo*, *strato di pianificazione*, *strato di modellazione*. Questi strati operano concorrentemente ed ognuno è collegato sia al sistema di percezione dell'ambiente che al sistema che realizza le azioni dell'agente.
Lo *strato reattivo* compie azioni in funzione degli eventi percepiti dall'ambiente.
Lo *strato di pianificazione* genera ed esegue piani, che sono parzialmente ordinati all'interno di una libreria, al fine di soddisfare gli obiettivi a lungo termine dell'agente.
Lo *strato di modellazione* fornisce la possibilità di modificare i piani ottenuti dallo strato di pianificazione in funzione dei cambiamenti dell'ambiente esterno durante la loro esecuzione. Inoltre lo strato di modellazione fornisce una struttura per la modellazione dell'ambiente, e per la rappresentazione dello stato interno di eventuali altri agenti.
I tre strati interagiscono tramite una *struttura di controllo* che attraverso due tipi di regole definisce il comportamento effettivo dell'agente. Le regole del primo tipo filtrano gli eventi in ingresso dall'ambiente selezionando solo quelli rilevanti, mentre quelle del secondo tipo filtrano le azioni generate dallo strato reattivo e da quello di pianificazione in modo da ottenere un comportamento coerente.

1.4.4 Muller et al.: INTERRAP

Oltre alle tre componenti descritte nella precedente architettura INTERRAP estende la struttura di controllo aggiungendo uno *strato di pianificazione cooperativa*, che fornisce protocolli e strategie di comunicazione tra agenti, ed uno *strato di pianificazione locale*. Tale estensione fornisce agli agenti INTERRAP la capacità di realizzare la pianificazione di azioni tramite l'interazione con altri agenti. In questo modo un agente può decidere se cominciare una cooperazione con altri agenti per perseguire un obiettivo oppure di perseguire tale obiettivo sfruttando solo le proprie capacità.
Il comportamento cooperativo si manifesta quando gli strati più a basso livello della struttura di un agente non sono in grado di risolvere una data situazione, quindi viene innescato un passaggio di messaggi da uno strato di livello inferiore a quello di livello superiore fino al raggiungimento dello *strato di pianificazione cooperativa* il quale attraverso l'interazione con altri agenti produce un piano costituito da azioni base e da comandi di sincronizzazione. Tale piano viene quindi passato allo *strato di pianificazione locale* il quale esegue le azioni descritte nel piano interagendo e sincronizzandosi con gli altri agenti.

Capitolo 2

Linguaggi di specifica per agenti

I linguaggi di specifica per agenti che hanno le caratteristiche descritte nella sezione sono molti ed ognuno caratterizzato da strutture formali diverse, dalla descrizione degli agenti con metodologie orientate agli oggetti alla descrizione del loro comportamento tramite qualità mentali.
In questo capitolo viene presentata una rassegna di linguaggi di specifica per agenti.

2.1 Programmazione orientata ad agenti (AOP)

Un linguaggio di specifica in cui l'agente è descritto tramite le proprie qualità mentali è **AGENT 0** realizzato da Yoav Shoham [Sho93] .
Shoham non ha proposto solo un linguaggio di programmazione ma una struttura per la programmazione orientata all'agente composta essenzialmente da tre punti:

1. un linguaggio formale con una sintassi semplice ed una semantica chiara in grado di descrivere lo stato mentale dell'agente;
2. un linguaggio interpretato in cui viene descritto il comportamento dell'agente, e la cui semantica è in accordo con la semantica della descrizione dello stato mentale.
3. un "agentificatore" che converte sistemi hardware e software in agenti programmabili.

Nell'articolo intitolato *Agent-oriented programming* Shoham sorvola la descrizione dell'agentificatore, mentre si concentra sui linguaggi per la descrizione dello stato mentale dell'agente e del suo comportamento.
Il primo passo per la descrizione dello stato mentale dell'agente riguarda le categorie mentali che vengono considerate.

In ogni istante il futuro è determinato dalla storia passata e dalle decisioni che vengono adottate nell'istante presente. Le decisioni adottate dipendono essenzialmente dalle credenze che l'agente ha sul passato, presente e futuro dell'ambiente in cui è collocato. In base a questa osservazione le categorie mentali utilizzate per descrivere il comportamento dell'agente sono *credenze, obbligazioni* e *capacità di compiere azioni*, dove per obbligazione viene inteso l'obbligo dell'agente di assumere necessariamente un certo comportamento nei confronti di un altro agente. In questo contesto la capacità di un agente di prendere decisioni non è altro che una obbligazione verso se stesso.

2.1.1 Un linguaggio per credenze, obbligazioni e capacità

L'utilizzo del concetto di *tempo*, per Shoham, sembra essere essenziale al fine di realizzare un linguaggio che descriva il comportamento dell'agente basandosi sulle categorie mentali appena elencate. Un agente, in un certo istante ha credenze su eventi che riguardano altri istanti di tempo, inoltre compirà azioni ad un preciso istante di tempo nel futuro, oppure le avrà compiute in un particolare istante del passato.
Nella programmazione orientata agli agenti viene utilizzato un linguaggio temporale discreto in cui un'istruzione ha la forma seguente:

$$predicato(arg_1 \ldots arg_n)^t$$

Ad esempio l'istruzione $solleva(robot, braccio)^t$ significa che il robot, al tempo t, solleva il braccio. In tale linguaggio le *azioni* sono rappresentate da fatti nel senso della logica del primo ordine: ad esempio il robot non intraprende l'azione solleva il braccio al tempo t ma piuttosto il fatto $solleva(robot, braccio)^t$ è reso vero al tempo t.

Nell'istante di tempo t$'$ l'agente *crede* che si manifestano eventi che riguardano l'istante di tempo t$''$. Per formalizzare questo concetto viene introdotto l'operatore modale B (belief), così l'espressione "l'agente a al tempo t crede φ " viene rappresentata come:

$$B_a^t \ \varphi$$

dove φ può essere definita ricorsivamente in modo da contenere ancora l'operatore B. Ad esempio l'espressione $B_a^3 B_b^{10} like(c, d)^7$ significa che al tempo 3 l'agente a crede che al tempo 10 l'agente b crederà che al tempo 7 c era come d.

Per descrivere il comportamento che deriva dalla categoria mentale *obbligazione* viene introdotto l'operatore OBL:

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

Un'espressione di questo tipo significa che l'agente a al tempo t è obbligato (oppure impegnato) verso l'agente b ad intraprendere l'azione φ, ma siccome non vi è distinzione tra fatti ed azioni questo significa che il fatto φ deve essere reso vero.

La possibilità da parte dell'agente di prendere una *decisione* viene rappresentata come un'obbligazione verso se stesso:

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

Questo significa che l'agente a è obbligato verso se stesso a fare in modo che il fatto φ sia reso vero. Con questa interpretazione il concetto di decisione ha strettamente a che fare con il concetto di scelta: per l'agente decidere un fatto è vero, significa scegliere che il fatto debba essere necessariamente reso vero.

Strettamente legata alla categoria mentale dell'obbligazione, e quindi della decisione, vi è la *capacità di compiere azioni*: un robot non può decidere di sollevare un braccio se non è capace di farlo, questo significa che un agente non *deciderà* di compiere azioni che non *crederà* di poter intraprendere. La capacità di compiere azioni non è necessariamente una categoria mentale però può essere descritta da tali categorie:

l'agente a è capace di $\varphi \iff$ (a decide $\varphi \implies \varphi$ può essere vero)

La capacità dell'agente a di compiere l'azione φ al tempo t è descritta dall'operatore:

$$CAN^t_a\ \varphi$$

Tale capacità è riferita ad uno specifico istante di tempo, infatti $CAN^5_{robot} open(door)^8$ significa che all'istante 5 il robot è capace di aprire la porta nell'istante 8, ma questa capacità può venir meno già nell'istante 6. Affinchè questo non accada devono essere definite delle condizioni sul concetto di capacità in modo che queste permangono fino a che l'agente ne abbia bisogno.

Sia $time(\varphi\ )$ una funzione il cui risultato è il massimo istante di tempo all'interno dell'espressione φ , ad esempio $time(open(door)^t)\ =\ t$.
Attraverso la funzione *time* può essere definita l'istruzione ABLE:

$$ABLE_a\ \varphi\ =_{def}\ CAN_a^{time(\varphi\ )}\varphi$$

ad esempio:

$$ABLE_{robot}\ open(door)^5 \ \equiv\ CAN^5_{robot}\ open(door)^5$$

l'istruzione ABLE è la versione "immediata" dell'istruzione CAN.

Per definite le categorie mentali e le espressioni sintattiche per rappresentarle è necessario assumere che queste soddisfino delle proprità al fine di rendere consistente il comportamento dell'agente. Le proprietà devono inoltre essere abbastanza "deboli" in modo che il linguaggio possa essere utilizzato per il maggior numero di applicazioni.
Sia le credenze che le obbligazioni devono essere internamente consistenti:

- per ogni t, a se $\psi \in \{\varphi\ :\ B^t_a\varphi\}$ allora $\neg\psi \notin \{\varphi\ :\ B^t_a\varphi\}$.
- per ogni t, a se $\psi \in \{\varphi\ :\ OBL^t_{a,b}\ \varphi\ \text{ per qualche } b\}$ allora $\neg\psi \notin \{\varphi\ :\ OBL^t_{a,b}\ \varphi\ \text{ per qualche } b\}$.

Avere definito l'operatore modale B in relazione ad un istante di tempo significa che se l'agente ha una credenza in un dato istante di tempo, nell'istante immediatamente successivo l'agente non ha più tale credenza. Questa caratteristica limita le capacità di un agente, quindi nell'AOP viene assunto che l'agente abbia memoria delle proprie credenze e mantenga tale insieme consistente eliminando le credenze in contraddizione con quella appresa più di recente. Inoltre l'agente non crede ad un fatto fino a quando non lo apprende (questo non significa che mentre non ha ancora appreso un dato fatto crede alla negazione di quel fatto, ma semplicemente che non ha nessun tipo di conoscenza riguardo a quel fatto).

L'agente è obbligato ad assumere un dato comportamento nei confronti di un altro agente solo se è possibile assumere quell'atteggiamento e se l'agente è capace di assumerlo:

- per ogni t,a,b,φ : $OBL^t_{a,b}\ \varphi\ \supset\ B^t_a((ABLE_a\ \varphi\ ) \wedge \varphi\ )$.

L'agente ha capacità di introspezione, cioè è a conoscenza delle proprie obbligazioni:

- per ogni t,a,b,φ : $OBL^t_{a,b}\ \varphi\ \equiv\ B^t_a OBL^t_{a,b}\ \varphi$;
- per ogni t,a,b,φ : $\neg OBL^t_{a,b}\ \varphi\ \equiv\ B^t_a \neg OBL^t_{a,b}\ \varphi$.

Un'ultima proprietà riguarda la capacità di compiere azioni: se un agente in un certo istante è capace di compiere un'azione allora tale capacità permane nel tempo, quello che cambia sono le credenze riguardanti tali capacità. Se l'agente è capace di compiere un'azione ed esiste un istante di tempo in cui *crede* di essere capace di eseguire tale azione allora la eseguirà.

2.1.2 Un interprete per un agente definito in AOP

Nella sezione precedente sono state definite le qualità mentali (e le proprietà di tali qualità) che caratterizzano il comportamento di un agente AOP. Tale comportamento viene ottenuto dall'esecuzione delle due azioni che compongono il ciclo di funzionamento dell'interprete per agenti AOP:

1. Lettura dei messaggi, aggiornamento dello stato mentale (conoscenza, obbligazioni e capacità).

2. Esecuzione delle obbligazioni riferite all'istante di tempo corrente.

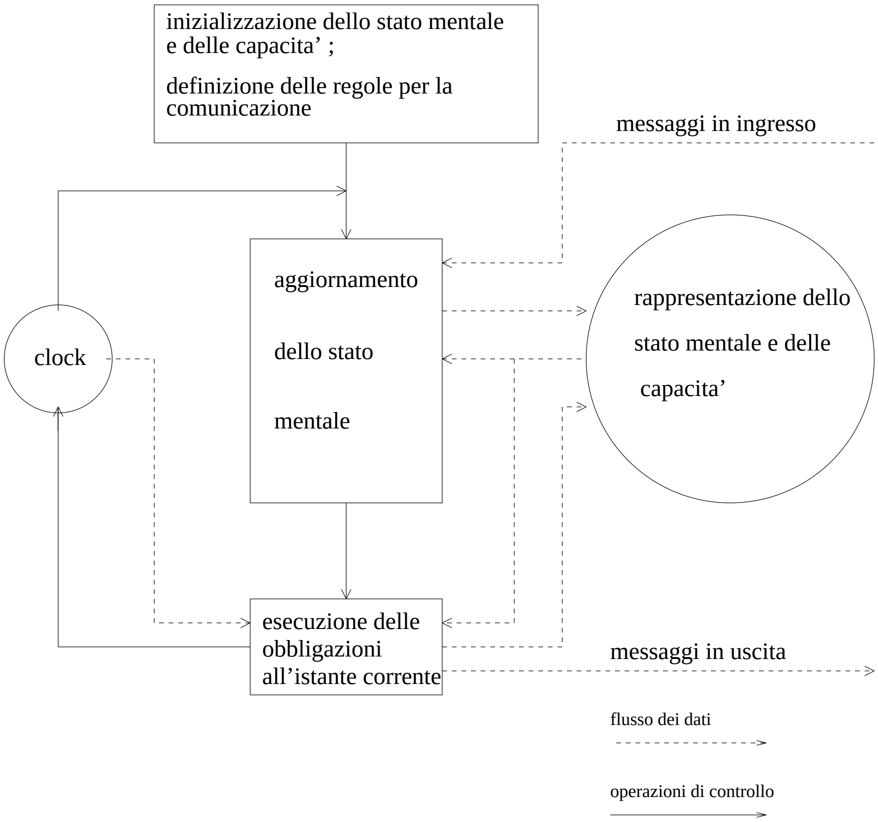


Figura 2.1: Diagramma di flusso dell'interprete per agenti AOP.

Le obbligazioni che vengono eseguite al passo 2 riguardano lo scambio di messaggi tra agenti e l'esecuzione di azioni "private", cioè obbligazioni verso l'agente stesso oppure azioni che hanno lo scopo di modificare lo stato mentale dell'agente, e che quindi verranno effettivamente eseguite alla successiva iterazione del ciclo al

passo 1.
La figura 2.1 rappresenta il diagramma di flusso del funzionamento di tale interprete. La componente fondamentale per il funzionamento dell'interprete è il clock il cui ruolo fondamentale è quello di effettuare le iterazioni del ciclo ad intervalli regolari. Nella componente clock viene inizializzata la variabile *timegrain* con il valore della lunghezza dell'intervallo di tempo necessario per eseguire le due azioni dell'interprete, ed inoltre ad ogni ciclo di esecuzione viene assegnato alla variabile *now* il valore dell'istante di tempo corrente.

2.1.3 AGENT 0 ed il suo interprete.

AGENT 0 è un semplice linguaggio di programmazione per agenti realizzato nello stile AOP, cioè le cui istruzioni caratterizzano il comportamento dell'agente attraverso le qualità mentali descritte precedentemente. Il linguaggio con cui vengono descritte tali qualità fa riferimento al primo punto della struttura per la programmazione orientata ad agent della sezione 2.1, mentre AGENT 0 fa riferimento al secondo punto.

AGENT 0 non vuole essere un linguaggio applicativo in grado di realizzare sistemi multi-agenti che possono essere utilizzati per fini commerciali, ma solo un prototipo sul quale possono essere attuati miglioramenti in modo da realizzare linguaggi in stile AOP adatti a risolvere problemi reali. Pertanto anche l'interprete di AGENT 0 è solamente una semplice istanza dell'interprete descritto nella sezione 2.1.2.

In questo semplice linguaggio di programmazione possono essere definiti azioni credenze ed obbligazioni. Le obbligazioni riguardano solamente azioni primitive, cioè azioni che possono essere direttamente eseguite dall'agente e che non necessitano di nessun tipo di pianificazione.
Di seguito è descritta la sintassi di AGENT 0 e quindi la definizione informale delle categorie sintattiche più interessanti:

Dalla B.N.F. di AGENT 0, figura 2.2, si evince che un programma è una sequenza di obbligazioni preceduta dalla definizione delle credenze iniziali, da quella delle capacitàe dall'inizializzazione della variabile *timegrain*.

Un costrutto sintattico fondamentale per AGENT 0 sono i *fatti* (*fact*). Tale costrutto è utilizzato per la definizione delle azioni e delle condizioni necessarie alla loro esecuzione. I *fatti* sono sentenze atomiche, quindi non sono *fatti* le congiunzioni né le disgiunzioni di sentenze atomiche e non lo sono neppure sentenze a cui sono applicati operatori modali.

```
<program> ::= timegrain := <time>
              CAPABILITIES := (<action> <mntlcond>)*
              INITIAL BELIEFS := <fact>*
              COMMITMENT RULES := <commitrule>*

<commitrule> ::= (COMMIT <msgcond> <mntlcond> (<agent> <action>)*)

<msgcond> ::= <msgconj> | (OR <msgconj>*)

<msgconj> ::= <msgpattern> | (AND <msgpattern>*)

<msgpattern> ::= <agent> INFORM <fact> |
                 <agent> REQUEST <action> |
                 (NOT <msgpattern>)

<mntlcond> ::= <mntlconj> | (OR <mntlconj>*)

<mntlconj> ::= <mntlpattern> | (AND <mntlpattern>*)

<mntlpattern> ::= (B <fact>) | ((CMT <agent>) <action>) |
                  (NOT <mntlpattern>)

<action> ::= (DO <time> <privateaction>) |
             (INFORM <time> <agent> <fact>) |
             (REQUEST <time> <agent> <action>) |
             (UNREQUEST <time> <agent> <action>) |
             (REFRAIN <action>) |
             (IF <mntlcond> <action>)

<fact> ::= (<time> (<predicate> <arg>*))

<time> ::= <integer> | now | <time constant> |
           (+ <time> <time>) | <variable>

<time constant> ::= m | h | d | y

<agent> ::= <alphanumeric string> | <variable>

<predicate> ::= <alphanumeric string>

<arg> ::= <alphanumeric string> | <variable>

<variable> ::= ?<alphanumeric string> | ?!<alphanumeric string>
```

Figura 2.2: Sintassi formale di AGENT 0.

Un esempio di *fatto* è : (t (solleva braccio sinistro)).
Quando un agente è soggetto ad una obbligazione l'*azione* da eseguire (*action*) può essere *privata* oppure *comunicativa* ed ognuno di questi due tipi di *azioni* può essere sia condizionale che non condizionale.

La sintassi per le *azioni private* è :

(DO t p-action)

dove t è un istante di tempo e p-action una *azione privata*. Il nome di una azione privata è conosciuto solo dall'agente che la esegue e può coinvolgere oppure no l'input-output dell'agente che la esegue. A differenza delle azioni private le azioni comunicative coinvolgono sempre l'input-output dell'agente e i nomi di tali azioni sono comuni ad ogni agente del sistema multi-agente.
In AGENT 0 vi sono tre tipi di azioni comunicative: azioni che *informano* un agente di un fatto, azioni che *richiedono* ad un agente di eseguire una azione ed azioni che *cancellano* una richiesta di azione precedentemente fatta ad un agente.

La sintassi per una *azione comunicativa di informazione* è :

(INFORM t a fact)

dove t è un istante di tempo, a il nome dell'agente che viene informato, e fact il fatto rappresentante l'informazione. È interessante notare che t è l'istante di tempo in cui l'azione comunicativa viene eseguita e che a sua volta fact è una sentenza che contiene altre informazioni temporali.

La sintassi per una *azione comunicativa di richiesta* di una azione ad un agente è la seguente:

(REQUEST t a action)

dove t è l'istante di tempo in cui viene eseguita l'azione comunicativa, a il nome dell'agente a cui viene richiesta l'azione ed action un'azione eventualmente definita ricorsivamente, ad esempio: (REQUEST 1 a (REQUEST 5 b (INFORM 10 c fact))). L'agente che esegue questa azione nell'istante 1 chiede all'agente a l'esecuzione di una azione a sua volta comunicativa che deve essere eseguita all'istante 5 dall'agente b e che consiste nell'informare, all'istante 10, l'agente c del fatto $fact$.

L'*azione comunicativa di cancellazione* di una richiesta all'agente a all'istante t è definita sintatticamente come:

(UNREQUEST t *a* action).

Esiste un'ultima azione che impedisce ad altri agenti di richiedere una azione all'agente che la esegue:

(REFRAIN action).

Le azioni viste fino a questo punto sono azioni non condizionali.
La condizione sulla quale si basano le azioni condizionali è definita in funzione dello stato mentale dell'agente. Quando deve essere eseguita una azione condizionale lo stato mentale dell'agente viene esaminato e se tale stato mentale soddisfa la condizione l'azione viene eseguita.
Una condizione mentale (*mntlcond*) è definita come congiunzione, disgiunzione oppure negazione di *mental patterns*, dove un *mental pattern* è definito come:

(B fact)

oppure

((CMT *a*) action)

dove fact è un fatto, *a* è il nome di un agente, action è una azione. L'operatore B all'interno del *mental pattern* viene utilizzato per definire la condizione mentale sulle credenze dell'agente, mentre l'operatore CMT è utilizzato per definire la condizione mentale sulle obbligazioni che impegnano l'agente: *a* è l'agente destinatario dell'obbligazione e action l'azione che deve essere intrapresa verso l'agente destinatario. Un esempio di *mental pattern* è : (B (t (braccio sinistro sollevato))).
Data la sintassi della condizione mentale la sintassi dell'azione condizionale è :

(IF mntlcond action)

dove mntlcond è una condizione mentale e action è una azione eventualmente condizionale.
Un esempio di azione comunicativa condizionale è :

(REQUEST t b (IF (B fact) (INFORM t+1 *a* fact)))

In AGENT 0 le variabili sono denotate dal prefisso "?" e possono sostituire nomi di agenti, fatti o azioni. Una azione condizionale con variabili potrebbe essere:

(IF (NOT ((CMT ?x) (REFRAIN canta))) canta) .

Le variabili sono considerate quantificate esistenzialmente quindi il loro scope si espande fino al primo NOT oppure per tutta l'istruzione se non contiene NOT. Le variabili di AGENT 0 possono essere quantificate anche universalmente, tali variabili sono precedute dal prefisso "?!" ed il loro scope è tutta l'istruzione.

A differenza dell'obbligazione definita nel linguaggio che descrive lo stato mentale dell'agente (sezione 2.1.1) l'obbligazione utilizzata nel linguaggio per la descrizione del comportamento dell'agente viene eseguita se sono soddisfatte sia una condizione sullo stato mentale dell'agente sia una condizione sui messaggi scambiati con gli altri agenti del sistema.
La condizione sullo stato mentale dell'agente è la *mntlcond* definita precedentemente, mentre quella sui messaggi è definita come congiunzione, disgiunzione oppure negazione di *message patterns*, dove un *message pattern* è definito come una tripla:

(mittente tipo contenuto)

in cui *mittente* è un nome di agente, *tipo* può assumenre i valori INFORM, REQUEST, UNREQUEST, *contenuto* è un fatto oppure una azione.
A questo punto può essere definito il costrutto sintattico più importante in AGENT 0 cioè l'obbligazione. Il concetto di obbligazione in AGENT 0 è rappresentato sintatticamente dalla *commitrule*:

(COMMIT msgcond mntlcond (agent action)$*$)

dove *msgcond* e *mntlcond* sono rispettivamente la condizione sui messaggi e la condizione mentale, *agent* è un nome di agente, *action* una azione e $*$ significa la ripetizione di zero o più volte della coppia (agent action).
Un esempio di obbligazione è :

(COMMIT (?a REQUEST ?action) (B (now (myfriend ?a))) (?a ?action))

dove (?a REQUEST ?action) è la condizione sui messaggi, (B (now (myfriend ?a))) è la condizione mentale e (?a ?action)) è l'azione che l'agente deve intraprendere nei confronti dell'agente mittente. L'obbligazione descritta nell'esempio significa: se esiste un'agente, il cui nome è rappresentato dal valore con cui è istanziata la variabile a, chiede che venga eseguita l'azione *action* e tale agente è "mio amico" allora esegui l'azione *action*.

L'interprete di AGENT 0 è una istanza dell'interprete definito nella sezione 2.1.2, quindi anch'esso costituito da un ciclo che itera due istruzioni ad intervalli regolari la cui dimensione è determinata dal valore associato alla variabile *timegrain*:

1. Aggiornamento delle credenze e delle obbligazioni.

2. Esecuzione delle obbligazioni riferite all'istante di tempo corrente.

Poiché negli agenti AGENT 0 la capacità di compiere azioni non cambia durante l'attività dell'agente, il primo passo dell'interprete generale viene specializzato nell'aggiornare solo le credenze e le obbligazioni mentre il secondo passo rimane immutato.

In AGENT 0 sia le credenze che le obbligazioni sono rappresentate in basi di dati.
La base di dati delle credenze è aggiornata sia attraverso la ricezione di messaggi che come conseguenza dell'esecuzione di una azione privata. Il problema dell'assimilare credenze (cioè l'aggiornamento della base di dati) in modo che l'insieme di tali credenze sia consistente è stato risolto semplificando la sintassi delle espressioni che rappresentano le credenze, permettendo cioè soltanto l'uso della negazione logica e non degli altri connettivi logici. Questo rende la ricerca della consistenza lineare nella dimensione della base di dati. Quando un agente viene informato di un fatto tale fatto viene aggiunto alla base di dati, nel caso in cui la base risultasse inconsistente viene eliminato l'insieme minimale di vecchi fatti (fatti presenti nella base di dati prima che venisse inserito il nuovo fatto) che la rendono inconsistente.
La base di dati delle obbligazioni è composta da coppie della forma (agente, azione), dove agente è l'agente "destinatario" dell'obbligazione e azione è l'azione oggetto dell'obbligazione.
Anche le capacità di azione sono contenute in una base di dati i cui elementi sono della forma (azione privata, condizione mentale) dove la condizione mentale determina il contesto in cui tale azione può essere eseguita.
Per poter eseguire l'azione oggetto di un'obbligazione l'agente deve essere *capace* di eseguire tale azione ed inoltre deve *credere* di poter eseguire tale azione. Da questo punto di vista la capacità di eseguire azioni è strettamente legata alla conoscenza dell'agente e quindi all'aggiornamento di tali conoscenze: ogniqualvolta le credenze dell'agente vengono aggiornate l'interprete di AGENT 0 esamina le obbligazioni correnti il cui oggetto è un'azione privata ed elimina quelle le cui "precondizioni" sono violate.
Le obbligazioni possono essere rimosse anche attraverso la ricezione del messaggio UNREQUEST. In questo caso l'interprete elimina dalla base di dati relativa alle obbligazioni l'elemento corrispondente, se esiste, altrimenti non accade nulla.
È interessante notare che la rimozione delle obbligazioni non dipende dal programma, mentre l'aggiunta di tali obbligazioni è strettamente dipendente dal programma.

L'algoritmo di aggiunta delle obbligazioni per l'interprete di AGENT 0 è :

Per ogni istruzione della forma (COMMIT msgcond mntlcond (a_i $action_i$)$*$) se:

- la condizione sui messaggi msgcond vale per il messaggio ricevuto,
- la condizione mentale mntlcond vale per il corrente stato mentale,
- per ogni i l'agente è correntemente capace di eseguire l'azione $action_i$,
- per ogni i l'agente non ha ricevuto messaggi con contenuto REFRAIN $action_i$, l'azione $action_i$ a sua volta non è della forma REFRAIN $action_i^{'}$ e l'agente non è obbligato ad eseguire l'azione $action_i^{'}$,

allora per ogni i l'agente è obbligato verso l'agente a_i ad eseguire l'azione $action_i$; nel caso una di queste condizioni non valga per qualche azione dell'obbligazione, tale azione non viene eseguita mentre le altre si.

Inoltre un agente è capace di eseguire una azione in accordo alle seguenti condizioni:

- Un agente può inviare messaggi di REQUEST e UNREQUEST a qualsiasi altro agente.
- Un agente può informare chiunque (compreso se stesso) di un fatto in cui crede.
- Un agente è capace di una azione se la condizione mentale è in accordo con lo stato mentale nell'istante in cui deve essere eseguita l'azione.
- Un agente può rifiutare qualsiasi azione per la quale non sia già stato obbligato.
- Un agente può eseguire un'azione condizionale (IF mntlcond action) se può eseguire l'azione action in accordo con la condizione mentale mntlcond.

Nel secondo passo del ciclo che definisce l'interprete di AGENT 0 vengono eseguite le obbligazioni relative all'istante corrente. L'interprete esegue tutte le azioni che devono essere eseguite nell'intervallo (*now - timegrain, now*], l'effetto dell'esecuzione dipende dal tipo di azione:

- INFORM, REQUEST, UNREQUEST: inviano l'appropriato messaggio.
- REFRAIN: ha l'unico effetto di rifiutare obbligazioni che hanno come soggetto l'azione argomento di REFRAIN.

- DO: consultare la base di dati delle credenze e quella delle obbligazioni, quindi esaminare la condizione mentale associata all'azione primitiva argomento di DO, e se questa vale eseguire tale azione.

- IF: consultare la base di dati delle credenze e quella delle obbligazioni, quindi esaminare la condizione mentale, se questa vale, eseguire l'azione.

2.1.4 Osservazioni

Come già osservato AGENT 0 non vuole essere un linguaggio applicativo ma un linguaggio che può essere sperimentato su alcuni domini applicativi col fine di evidenziare pregi e difetti dei linguaggi utilizzati per realizzare agenti in stile AOP. AGENT 0 ha un buon set di istruzioni per la comunicazione mentre le istruzioni per eseguire azioni, cioè obbligazioni e azioni condizionali, non sono sufficienti per permettere all'agente di pianificare autonomamente una sequenza di azioni in grado di risolvere situazioni complesse, rendendo così difficile la programmazione di agenti che sono in grado di affrontare domini applicativi particolarmente complicati.

2.2 dMARS

Nella sezione 1.2.1 del capitolo 1 è stata descritta l'architettura che da origine ad agenti **BDI**. Un linguaggio di specifica per agenti di tale architettura è **dMARS** (distribuited Multi-Agent Reasoning System) [dKLW98].

L'agente *BDI* è essenzialmente composto dalle quattro componenti illustrate nella figura 2.3: le *credenze*, cioè la rappresentazione delle informazioni che l'agente ha del mondo in cui è collocato, i *desideri* cioè gli obiettivi che l'agente deve perseguire, le *intenzioni* cioè le azioni da compiere per soddisfare un proprio obiettivo, la *libreria di piani* che è un insieme di piani che l'agente utilizza per realizzare le proprie intenzioni. La libreria di piani di un agente rappresenta la conoscenza procedurale dell'agente, cioè la conoscenza necessaria per elaborare soluzioni di problemi complessi.
I piani sono costituiti da diverse componenti:

- La *condizione sull'invocazione* (trigger): specifica le circostanze sotto le quali il piano viene selezionato per realizzare un desiderio, ad esempio il piano "fai il tè " viene selezionato se l'agente è assetato.

- Il *contesto*: specifica le circostanze sotto le quali l'esecuzione del piano può cominciare, in seguito all'esempio precedente il piano "fai il tè " viene eseguito se l'agente è in possesso della bustina di tè e di una tazza .

- Le *condizioni di mantenimento* del piano, cioè le circostanze che devono essere mantenute vere durante l'esecuzione del piano.

- Il *corpo* del piano cioè la struttura che rappresenta il possibile metodo di soluzione, può contenere sia sotto obiettivi che azioni primitive. In riferimento all'esempio "fai il tè " il corpo dell piano potrebbe essere una sequenza di azioni della forma: "prendi dell'acqua bollente", "metti la bustina del tè nell'acqua", "metti l'acqua nella tazza". In questo esempio il compito "prendi dell'acqua bollente" potrebbe essere un sotto obiettivo, mentre il resto della sequenza potrebbero essere azioni primitive, cioè azioni che l'agente è capace di eseguire senza completare ulteriori sotto obiettivi.

L'agente dMARS osserva sia l'ambiente in cui è situato sia il proprio stato interno, e gli eventi così percepiti vengono inseriti in una *coda degli eventi* (event queue). Questa operazione, così come quella di portare a termine un obiettivo, è realizzata dall'interprete dMARS che esegue continuamente il seguente ciclo:

1. Osserva il mondo esterno e lo stato interno aggiornando la coda degli eventi con gli eventi osservati.

2. Genera nuovi possibili desideri selezionando piani la cui condizione sull'invocazione è in accordo con un evento nella coda degli eventi.

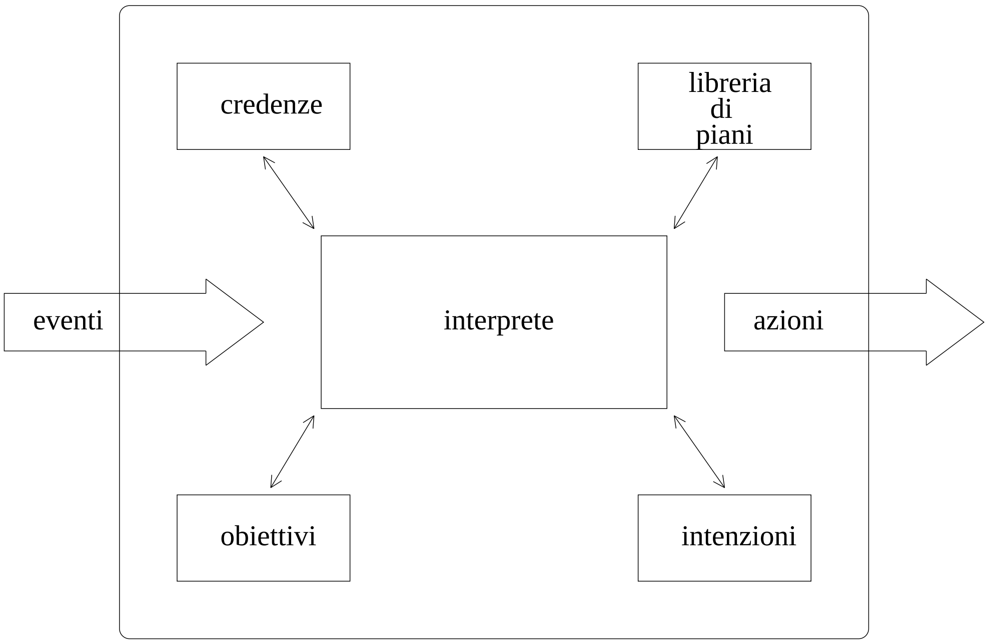


Figura 2.3: Rappresentazione dell'architettura BDI.

3. Scegli un piano tra quelli selezionati al passo precedente.

4. Metti il piano scelto in una pila delle intenzioni già esistente se l'evento in accordo con la condizione al passo 2 è un evento relativo ad un sotto obiettivo, altrimenti, se l'evento è relativo ad un obiettivo, genera una nuova pila delle intenzioni e metti al suo interno il piano.

5. Seleziona una pila delle intenzioni, prendi il primo elemento della pila (un piano), esegui il prossimo passo di tale elemento; se il passo consiste in una azione allora eseguila, se è un sotto obiettivo allora genera un evento relativo a tale sotto obiettivo e mettilo nella coda degli eventi.

In questo modo l'interprete descrive il comportamento dell'agente dalla scelta di un piano per soddisfare un dato desiderio fino alla completa esecuzione di tale piano attraverso azioni primitive e sotto obiettivi.

2.2.1 Credenze, obiettivi e azioni

Le credenze in dMARS sono rappresentate come letterali ground della logica classica, per darne una descrizione formale verranno utilizzati gli insiemi Var, FunSym, PredSym, rispettivamente come insieme delle variabili, insieme dei simboli di funzione, insieme dei simboli di predicato.

L'insieme dei termini Term è definito induttivamente da:

- se $v \in \mathrm{Var}$ allora $v \in \mathrm{Term}$;
- se $f \in \mathrm{FunSym}$ è un simbolo di funzione con zero argomenti allora f è una costante ed appartiene a Term;
- se $t_1, \ldots, t_n \in \mathrm{Term}$, $n \geq 1$, e $f \in FunSym$ è un simbolo di funzione n-aria allora $f(t_1, \ldots, t_n) \in \mathrm{Term}$;

Un atomo è un simbolo di predicato applicato ad una sequenza di termini:

Atom ::= PredSym(Term$*$)

dove Term$*$ indica una sequenza di termini.

L'espressione rappresentante una credenza (belief formula) è un atomo oppure la negazione di un atomo:

BeliefFormula ::= Atom | $\neg$ Atom

L'insieme delle credenze dell'agente è quindi l'insieme delle formule ground rappresentanti le credenze (l'insieme delle BeliefFormula ground).

Gli obiettivi (goal) vengono rappresentati tramite l'uso dei due operatori modali ! e ?. Quindi un agente con obiettivo la formula:

$!\phi$

deve produrre una sequenza (eventualmente vuota) di azioni la cui esecuzione rende la formula ϕ vera.
L'agente con obiettivo la formula:

$?\phi$

deve produrre una sequenza (eventualmente vuota) di azioni la cui esecuzione rende possibile all'agente verificare se la formula ϕ è vera oppure no.
La sintassi per gli obiettivi è :

Goal ::= !BeliefFormula | ?SituationFormula

dove SituationFormula è una formula basata sulle credenze dell'agente che può essere valutata vera oppure falsa:

SituationfFormula ::=

BeliefFormula |
(SituationFormula $\wedge$ SituationFormula) |
(SituationFormula $\vee$ SituationFormula) |
true | false

Le azioni che un agente dMARS è capace di compiere sono di due tipi: azioni interne ed azioni esterne.
Le azioni esterne manifestano il loro effetto all'esterno dell'agente, cioè nell'ambiente in cui è collocato. Tali azioni sono invocate come chiamate di procedura il cui nome appartiene all'insieme ActionSym ed è seguito da una sequenza di termini:

ExtAction ::= ActionSym(Term$*$)

Le azioni interne servono per aggiornare lo stato interno dell'agente, aggiungendo e rimuovendo credenze dall'insieme delle credenze dell'agente. La sintassi delle azioni interne è :

IntAction ::= add(BeliefFormula) | remove(BeliefFormula)

2.2.2 Definizione dei piani

La sintassi di un piano è rappresentata da:

Plan ::=

invocation(TriggerEvent),
context(SituationFormula),
body(Body),
maintenance(SituationFormula),
succes(IntAction $*$),
failure(IntAction $*$)

dove *invocation* è l'evento che seleziona il piano, *context* è il contesto relativo al piano, *body* è il corpo del piano, *maintenance* è una formula che deve rimanere vera durante l'esecuzione del piano, *success* e *failure* sono una sequenza di azioni che l'agente esegue rispettivamente se il piano termina con successo oppure se termina con il fallimento.

Un piano viene selezionato quando è soddisfatta la condizione sull'invocazione: tale condizione può riguardare l'aggiornamento della conoscenza, la percezione di un messaggio, oppure l'acquisizione di un nuovo obiettivo. La sintassi della condizione sull'invocazione è :

TriggerEvent ::=

addbelevent(Belief) |
rembelevent(Belief) |
toldevent(Atom) |
goalevent(Goal)

Nell'agente dMARS il corpo del piano è un albero in cui ogni nodo rappresenta una scelta tra possibili azioni e la cui etichetta appartiene ad un insieme State. Un arco all'interno di un piano può essere etichettato con azioni interne, esterne, oppure con un sotto obiettivo:

Branch ::=

extaction(ExtAction) |
intaction(IntAction) |
subgoal(Goal)

Il nodo di un grafo rappresentante un piano è una foglia etichettata con la parola End oppure una diramazione:

Body ::= End | Fork((State, Branch, Body)$*$)

dove (State, Branch, Body)$*$ rappresenta la sequenza di rami che escono dal nodo.

Una volta selezionato un piano ne viene creata una istanza, ovvero una copia del piano con alcune informazioni addizionali riguardanti l'esecuzione del piano:

- l'ambiente, cioè l'insieme dei valori associati alle variabili durante l'esecuzione del piano;
- lo stato corrente raggiunto durante l'esecuzione, inizialmente la radice dell'albero;
- l'insieme degli archi che possono essere attraversati dallo stato corrente;
- l'arco attualmente in esecuzione;
- un valore che identifica univocamente l'istanza del piano;
- lo stato del piano. Tale stato può assumere i valori: attivo, quando il piano fa parte dell'intenzione corrente; inattivo, quando il piano è momentaneamente sospeso.

Se un arco del grafo non può essere attraversato, ad esempio perchè un sotto obiettivo fallisce, anche tale arco fallisce e viene rimosso dall'insieme degli archi possibili da attraversare.
L'esecuzione di un piano ha successo quando viene raggiunto lo stato finale End, altrimenti fallisce se l'esecuzione non ha raggiunge lo stato finale e l'insieme degli archi possibili da attraversare è vuoto.

Un'intenzione è una sequenza di istanze di piani: in risposta ad un evento esterno viene generata una intenzione consistente in una istanza di un piano. Se l'esecuzione di questa istanza genera un evento interno ed a tale evento l'agente reagisce generando un altro piano, quest'ultimo piano viene concatenato al piano precedente. In questo modo si crea una pila di piani (la pila delle intenzioni) in cui il piano che sta in cima alla pila è il primo ad essere eseguito.

2.2.3 Semantica operazionale degli agenti dMARS

Dal punto di vista semantico un agente dMARS è formato da una *libreria di piani*, una *funzione di selezione delle intenzioni*, una *funzione di selezione degli eventi*, una *funzione di selezione dei piani*, una *funzione di selezione delle sostituzioni* per scegliere tra i possibili assegnamenti di valori alle variabili durante l'esecuzione di un piano, ed una *funzione di selezione delle diramazioni* all'interno di un piano.

Lo stato è descritto dalle componenti dell'agente che vengono modificate durante la sua attività. Tali componenti sono: le credenze, le intenzioni, gli eventi. Lo stato dell'agente è inizializzato con credenze, eventi ed intenzioni utilizzate per avviare l'attività dell'agente. Gli eventi iniziali, e quelli che verranno percepiti in seguito vengono inseriti in una coda degli eventi.

Un piano è *rilevante* se esiste un unificatore più generale (mgu) che lega la condizione sull'invocazione del piano con l'evento percepito. Questo comportamento è specificato dalla funzione *genrelplan* che ha come argomenti un evento e, un insieme di piani ps e ritorna una coppia (p, σ) dove p è un piano rilevante di ps per l'evento e, e σ è l'unificatore più generale per p.

Un piano è *applicabile* se il suo contesto è conseguenza logica delle credenze dell'agente. Quindi viene definito il predicato *dMARSLogCons* che ha come argomenti una SituationFormula e l'insieme delle credenze dell'agente. Tale predicato è vero se la SituationFormula è conseguenza logica delle credenze dell'agente. Sfruttando questo predicato viene definita la funzione *genapplplans* che ha come argomenti un insieme di piani rilevanti con i relativi unificatori più generali e le credenze correnti dell'agente; tale funzione ritorna i piani applicabili con le relative sostituzioni aggiornate.

Se la coda degli eventi non è vuota viene selezionato un evento, viene creato un insieme di piani rilevanti, quindi tale insieme viene "sfoltito" trasformandolo nell'insieme dei piani applicabili, quindi un piano applicabile viene selezionato e viene creata l'istanza per quel piano.
Se l'evento è di tipo esterno viene creata un'intenzione (una pila di istanze di piani) contenente l'istanza del piano come unico elemento.
Se l'evento è interno l'istanza viene inserita nella pila dell'intenzioni che ha generato quell'evento interno.
Se la coda degli eventi è vuota viene selezionata una intenzione e quindi viene eseguita. Il primo passo dell'esecuzione consiste nel scegliere un ramo della prima istanza di piano nell'intenzione (che è una pila di istanze di piani). L'effetto di tale esecuzione dipende da quale tipo di azione è associata alla diramazione dell'istanza di piano eseguita:

- Se l'arco è etichettato con una azione esterna, tale azione viene immediatamente eseguita.
  Se l'esecuzione termina con successo l'arco viene attraversato per raggiun-

gere lo stato successivo e l'ambiente associato all'istanza del piano viene aggiornato con i valori delle variabili ottenuti durante l'esecuzione.
Se viene rilevato un fallimento non vi è cambiamento di stato e viene provato un altro arco dello stato corrente.

- Se l'azione è interna vengono generati gli eventi corrispondenti corrispondenti all'aggiornamento dell'insieme delle credenze in accordo con tale azione.
- Se l'azione associata all'arco è un sotto obiettivo della forma $?\phi$ viene tentata l'unificazione di ϕ , a cui è stato applicato l'ambiente delle sostituzioni, con l'insieme delle credenze. Se l'unificazione ha successo tale unificatore viene composto con l'ambiente delle sostituzioni e l'esecuzione passa allo stato successivo del piano. Se l'unificazione non ha successo allora l'esecuzione del piano fallisce.
- Infine se l'azione è un sotto obiettivo del tipo $!\phi$ viene tentata una unificazione come nel caso precedente. Se l'unificazione ha successo viene aggiornato l'ambiente delle sostituzioni e si passa allo stato successivo; se l'unificazione fallisce l'esecuzione del piano viene sospesa e viene inserito nella coda degli eventi un evento relativo all'obiettivo $!\phi$, il quale quando selezionato, se in accordo con le credenze dell'agente, aggiungerà l'istanza di un piano all'intenzione relativa. Quando l'istanza di piano sarà terminata con successo l'esecuzione del piano verrà riattivata.

Concludendo, se l'esecuzione del piano raggiunge lo stato finale il piano ha successo e l'ambiente ottenuto viene utilizzato per l'esecuzione del piano successivo nella pila delle intenzioni. Se in seguito al successo del piano la pila rimane vuota allora l'intenzione ha successo e quindi viene rimossa così come viene rimosso dalla coda degli eventi l'evento esterno che l'ha generata. L'intenzione fallisce quando almeno un piano della pila fallisce e non vi sono piani alternativi per l'evento esterno che ha generato l'intenzione.

2.2.4 Osservazioni

La possibilità di usufruire di una libreria di piani rende l'agente dMARS utilizzabile in svariate situazioni, inoltre tale libreria rende l'agente autonomo nel pianificare sequenze di azioni che hanno il fine di soddisfare l'obiettivo corrente.

In [dKLW98] non viene esplicitamente menzionato un metodo di comunicazione tra agenti facenti parte di un sistema multi-agente, ma tale comunicazione può essere realizzata attraverso l'impiego di azioni esterne il cui effetto è quello di inviare messaggi che vengono percepiti dagli agenti del sistema come eventi dell'ambiente, inseriti nella coda degli eventi, e quindi processati.

2.3 AgentSpeak(L)

AgentSpeak(L) [Rao96] è un linguaggio di specifica per agenti ispirato a dMARS (sezione 2.2). Con AgentSpeak(L) vengono specificati agenti BDI: vengono specificate le credenze, i desideri, le intenzioni ed i piani per realizzarle. Come in dMARS i piani vengono selezionati da eventi e le intenzioni sono organizzate in pile di istanze di piani.

Il linguaggio è basato su eventi ed azioni che attraverso il concetto di stato interno dell'agente ne descrivono il comportamento.
Lo stato comprende le credenze che l'agente ha sull'ambiente e sugli altri agenti situati in tale ambiente. Gli stati che l'agente vuole raggiungere possono essere visti come desideri mentre le "procedure" che utilizza per raggiungerli possono essere viste come intenzioni.

La specifica di un agente attraverso questo linguaggio consiste nella definizione di un insieme di credenze di base (rappresentate come fatti della logica del primo ordine) e di un insieme di piani, sensibili al contesto, attivati dagli eventi e dai desideri dell'agente.

2.3.1 Definizione del linguaggio

Per definire credenze, desideri e intenzioni degli agenti AgentSpeak(L) vengono utilizzati insiemi di variabili, costanti, simboli di funzione, simboli di predicato, simboli di azioni oltre ai connettivi della logica classica, agli operatori modali "!" e "?" ed al simbolo di concatenazione di istruzioni ";".
Una *credenza* è definita come congiunzione oppure negazione di *atomi credenza*, dove un *atomo credenza* è un simbolo di predicato che viene applicato ad una sequenza di termini:

Atomo credenza:

> se b è un simbolo di predicato, $t_1 \ldots t_n$ sono termini allora $b(t_1, \ldots ,t_n)$, in breve $b(t)$, è un atomo credenza. Un atomo credenza oppure la negazione di un atomo credenza è un *letterale credenza*.

Credenza:

> se $b(t)$ e $c(s)$ sono atomi credenza allora $b(t) \wedge c(s)$ e $\neg b(t)$ sono *credenze*.

Come per dMARS vi sono due tipi di *obiettivi* rappresentati rispettivamente dagli operatori modali ! e ? (sezione 2.2.1):

Obiettivi:

se b è un simbolo di predicato, $t_1 \ldots t_n$ sono termini allora $!b(t_1, \ldots, t_n)$, in breve $!b(t)$, e $?b(t_1, \ldots, t_n)$, in breve $?b(t)$ sono obiettivi.

In generale il comportamento di un agente AgentSpeak(L) consiste nell'osservare l'ambiente in cui è collocato, percepirne i cambiamenti ed acquisire nuovi obiettivi; in base agli obiettivi acquisiti ed agli eventi percepiti l'agente può compiere azioni in grado di modificare sia l'ambiente che il proprio stato interno.
Gli eventi corrispondenti all'aggiunta e rimuozione di obiettivi e credenze vengono chiamati *eventi trigger*. L'aggiunta di un obiettivo oppure di una credenza viene rappresentata dall'operatore "+", mentre l'eliminazione di un obiettivo oppure di una credenza viene rappresentata dall'operatore "$-$":

Eventi trigger:

se $b(t)$ è un atomo credenza, $!g(t)$ e $?g(t)$ sono obiettivi, allora $+b(t)$, $-b(t)$, $+!g(t)$, $-!g(t)$, $+?g(t)$, $-?g(t)$ sono *eventi trigger*.

Azioni:

se a è un simbolo di azione, $t_1 \ldots t_n$ sono termini allora $a(t_1, \ldots, t_n)$, in breve $a(t)$, è una azione.

Come in dMARS, per soddisfare un desiderio l'agente AgentSpeak(L) utilizza dei piani costituiti da una *testa* e da un *corpo*. La *testa* di un piano consiste in un evento trigger e in un contesto separati dal simbolo ":". L'evento trigger specifica perché il piano è stato selezionato, il contesto specifica invece le credenze che devono essere vere quando viene selezionato il piano. Il corpo del piano, in AgentSpeak(L) a differenza di dMARS, consiste in una sequenza di obiettivi e di azioni.

Piani:

se e è un evento trigger, $b_1 \ldots b_n$ sono letterali credenza e $h_1; \ldots; h_n$ sono azioni oppure obiettivi allora:
$e : b_1 \wedge \ldots \wedge b_n \leftarrow h_1; \ldots; h_n.$
è un piano.
La parte del piano a sinistra di "$\leftarrow$" è detta *testa*, mentre la parte a destra di "$\leftarrow$" è detta *corpo*. La parte a destra dei ":" nella *testa* del piano è detta *contesto*.

Consideriamo l'esempio in cui un'ape robot si nutre dai fiori di un prato. L'ape può succhiare il nettare di un fiore se è sbocciato e non è occupato da un altra ape. La conoscenza dell'ape riguardo al prato consiste nella configurazione dei fiori, posizione delle altre api e propria posizione, quindi con credenze del tipo $adiacente(X, Y)$, $posizione(ape, X)$, $posizione(altreapi, X)$. In seguito allo sbocciare di un fiore l'ape può avere l'obiettivo di raggiungere quel fiore e, se non è occupato, di nutrirsi; in questo caso il piano per raggiungere il fiore sarebbe:

$$+!posizione(ape, X) : posizione(ape, X) \leftarrow true$$

$$+!posizione(ape, X) : posizione(ape, Y) \wedge (not(X = Y)) \wedge adiacente(Y, Z) \wedge (not(posizione(altreapi, Z))) \leftarrow muovi(Y, Z); !posizione(ape, X)$$

In questo esempio il piano, che realizza la capacità dell'ape di muoversi nel prato, è definito da due istruzioni che hanno lo stesso evento trigger: la prima fa terminare il piano con successo se l'ape è già nella posizione richiesta, la seconda istruzione ha effetto se l'ape non si trova nella posizione richiesta. In questo caso l'istruzione cerca la posizione di un fiore adiacente, controlla che il fiore non sia già occupato, sposta l'ape sul fiore trovato e richiama ricorsivamente il piano.
Per come è definito AgentSpeak(L) la descrizione dell'agente sembra essere simile ad una specifica in un linguaggio della programmazione logica, ma le clausole della programmazione logica e i piani di AgentSpeak(L) differiscono per diversi motivi:

- In programmazione logica la testa di una clausola è un goal nello stesso senso in cui sono goals quelli del corpo. In AgentSpeak(L) la testa di un piano contiene invece un evento trigger, il che rende l'invocazione dei piani guidata sia dai dati percepiti che dagli obiettivi dell'agente.

- I piani di AgentSpeak(L) sono sensibili al contesto: la loro invocazione non dipende solo dall'evento trigger, ma attraverso il contesto anche dalle credenze dell'agente.

- In programmazione logica il successo di una clausola genera una sostituzione delle variabili non legate, mentre l'esecuzione dei piani genera, oltre alla sostituzione, una sequenza di azioni che modificano l'ambiente.

- In programmazione logica l'esecuzione di un goal non può essere esplicitamente interrotta mentre in AgentSpeak(L) l'esecuzione di un piano può essere interrotta.

2.3.2 Semantica operazionale

Un agente AgentSpeak(L) consiste in un insieme di credenze B, un insieme di piani P, un insieme di eventi E, un insieme di azioni A, un insieme di intenzioni

I, tre funzioni di selezione S_ε, S_O e S_I.
Quando un agente percepisce un cambiamento nell'ambiente, oppure un utente esterno chiede all'agente di adottare un obiettivo, viene generato un evento trigger che viene inserito nell'insieme degli eventi E. La funzione di selezione S_ε seleziona e rimuove un elemento dall'insieme E. Viene tentata l'unificazione dell'evento selezionato con l'evento trigger dei piani appartenenti all'insieme P; i piani il cui evento trigger unifica con l'evento selezionato vengono chiamati *piani rilevanti* e l'unificatore viene chiamato *unificatore rilevante*. L'*unificatore rilevante* viene poi applicato alle credenze che costituiscono il contesto del piano, se il contesto così ottenuto è conseguenza logica delle credenze B viene generata una *sostituzione corretta di risposta*. Tale piano viene chiamato *piano applicabile* oppure *opzione* e l'unificatore rilevante composto con la sostituzione corretta di risposta viene chiamato *unificatore applicabile*.
Per ogni evento possono esistere più piani applicabili; la funzione di selezione S_O ne sceglie uno. L'applicazione dell'unificatore applicabile all'opzione scelta dalla funzione S_O produce la sequenza di azioni ed obiettivi risposta all'evento trigger che ha scatenato il procedimento.
Ogni intenzione è uno stack di piani parzialmente istanziati. L'evento che genera una nuova intenzione nell'insieme I e che crea un nuovo stack, viene detto *evento esterno*, mentre l'evento generato dall'esecuzione di un obiettivo all'interno di un piano viene detto *evento interno*. Quest'ultimo scatena la selezione di un piano la cui istanza viene inserita come top dello stack contenente il piano che ha generato l'evento interno.
La funzione S_I seleziona all'interno di I un'intenzione da eseguire. Selezionata un'intenzione l'agente esegue il primo obiettivo oppure la prima azione dell'elemento top dello stack (un piano). L'esecuzione di un obiettivo del tipo $!\phi$ genera un evento che aggiunge un piano all'intenzione corrente. L'esecuzione di un obiettivo del tipo $?\phi$ comporta la ricerca di una sostituzione che applicata all'obiettivo lo renda conseguenza logica dell'insieme delle credenze B; se tale sostituzione esiste allora l'obiettivo viene rimosso e la sostituzione viene applicata ai rimanenti piani dell'intenzione. L'esecuzione di una azione comporta l'aggiunta di tale azione all'insieme delle azioni A e la rimozione dell'azione dal piano.
Questo procedimento continua fino a che l'insieme degli eventi E risulta essere vuoto e non vi sono più intenzioni eseguibili.

La sintassi simile a quella dei linguaggi della programmazione logica fa si che il funzionamento dell'agente AgentSpeak(L) abbia una descrizione formale semplice e completa.
Un agente AgentSpeak(L) è descritto da una tupla:

$\langle$ E, B, P, I, A,S_ε,S_O,$S_I\rangle$

dove B, P, A sono gli insiemi delle credenze, piani e azioni già descritti in modo informale, mentre gli insiemi E, I e le funzioni S_ε, S_O, S_I saranno descritti più formalmente.
L'insieme I è un insieme di intenzioni ed ogni intenzione è definita come uno stack di piani parzialmente istanziati. Un intenzione appartenente ad I ha la seguente forma:

$$[p_1 \ddagger, \ldots, \ddagger\, p_z]$$

dove p_1 è il bottom dello stack, p_z è il top e l'operatore $\ddagger$ è il simbolo di separazione degli elementi dello stack. L'espressione $[+!true : true \leftarrow true]$ viene definita *intenzione banale* ed è rappresentata dal simbolo T.
L'insieme E è composto da elementi della forma:

$$\langle e, i\rangle$$

dove e è un evento trigger ed i è un'intenzione. Se e è un evento esterno i è istanziata con l'*intenzione banale*, altrimenti se e è un evento interno i è istanziata con l'intenzione che ha generato l'evento.

Definiti gli insiemi I ed E possono essere definiti i concetti di piani ed unificatori rilevanti e applicabili:

> Sia $S_\varepsilon(E) = \varepsilon = \langle d, i\rangle$, e sia p il piano $e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots ; h_n$. Il piano p è rilevante rispetto all'evento ε sse esiste almeno un mgu σ tale che $d\sigma = e\sigma$. L'unificatore σ è quindi chiamato *unificatore rilevante* per ε.
>
> Un piano p descritto da $e : b_1 \wedge \cdots \wedge b_m \leftarrow h_1; \ldots ; h_n$ è definito applicabile rispetto all'evento ε sse esiste un unificatore rilevante σ per ε ed esiste una sostituzione θ tale che $\forall(b_1, \ldots , b_m)\sigma\theta$ è una conseguenza logica di B. La composizione $\sigma\theta$ è detta *unificatore* applicabile per ε e θ è chiamata *sostituzione corretta di risposta*.

Il procedimento con cui viene creata una nuova intenzione oppure viene aggiunta l'istanza di un piano ad un intenzione già esistente dipende dal tipo di evento. Nel caso sia di tipo esterno viene selezionato un piano applicabile rispetto all'evento e quindi viene applicato l'unificatore applicabile al corpo del piano. L'istanza così ottenuta è utilizzata per creare una nuova intenzione che viene aggiunta all'insieme I:

> Sia $S_O(O_\varepsilon) = p$, dove O_ε è l'insieme di piani applicabili (opzioni) per l'evento $\varepsilon = \langle d, i\rangle$ e p è $e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots ; h_n$.

Il piano p diventa una nuova intenzione rispetto ad ε dove i è l'intenzione banale sse esiste un unificatore applicabile σ tale che:
$[+!true : true \leftarrow true \ddagger (e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots ; h_n)\sigma] \in$ I.

Nel caso di un evento interno, al corpo del piano applicabile selezionato, viene applicato l'unificatore applicabile; l'istanza così ottenuta viene aggiunta allo stack che contiene il piano che ha generato l'evento interno:

Sia $S_O(O_\varepsilon) = p$, dove O_ε è l'insieme di piani applicabili (opzioni) per l'evento $\varepsilon = \langle d, [p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow !g(t); h_2; \ldots ; h_n]\rangle$ e p è $+!g(s) : b_1 \wedge \ldots \wedge b_m \leftarrow k_1; \ldots ; k_j$. Il piano p è aggiunto allo stack la cui intenzione ha generato l'evento interno ε sse esiste un unificatore applicabile σ tale che:
$[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow !g(t); h_2; \ldots ; h_n \ddagger (+!g(s) : b_1 \wedge \ldots \wedge b_m)\sigma \leftarrow (k_1; \ldots ; k_j)\sigma] \in$ I.

Quando una intenzione viene selezionata la prima formula del corpo del piano che deve essere eseguita è del tipo $!\phi$, o del tipo $?\phi$, oppure è una azione o l'espressione *true*. Nel primo caso il sistema la esegue generando un evento, nel secondo cerca un mgu che unifichi l'obiettivo con l'insieme delle credenze, e nel caso in cui tale mgu esista viene applicato al resto dell'intenzione. Nel terzo caso l'azione viene aggiunta all'insieme A e nell'ultimo caso l'obiettivo $!\phi$ viene soddisfatto ed eliminato dallo stack; in questo caso la sostituzione viene applicata al resto dell'intenzione.

Sia $S_I(I) = i$, dove i è $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow !g(t); h_2; \ldots ; h_n]$.
L'intenzione i viene detta eseguita sse $\langle +!g(t), i\rangle \in$ E.

Sia $S_I(I) = i$, dove i è $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow ?g(t); h_2; \ldots ; h_n]$.
L'intenzione i viene detta eseguita sse esiste una sostituzione θ tale che $\forall g(t)\theta$ è una conseguenza logica di B e i è rimpiazzata da $[p_1 \ddagger \ldots \ddagger (f : c_1 \wedge \ldots \wedge c_y)\theta \leftarrow h_2\theta; \ldots ; h_n\theta]$.

Sia $S_I(I) = i$, dove i è $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow a(t); h_2; \ldots ; h_n]$.
L'intenzione i viene detta eseguita sse $a(t) \in$ A, e i è rimpiazzata da $[p_1 \ddagger \ldots \ddagger f : c_1 \wedge \ldots \wedge c_y \leftarrow h_2; \ldots ; h_n]$.

Sia $S_I(I) = i$, dove i è $[p_1 \ddagger \ldots \ddagger p_{z-1} \ddagger !g(t) : c_1 \wedge \ldots \wedge c_y \leftarrow true]$, dove p_{z-1} è $e : b_1 \wedge \ldots \wedge b_x \leftarrow !g(s); h_2; \ldots ; h_n$. L'intenzione viene detta eseguita sse esiste una sostituzione θ tale che $g(t)\theta = g(s)\theta$ e i è rimpiazzata da $[p_1 \ddagger \ldots \ddagger p_{z-2} \ddagger (e : b_1 \wedge \ldots \wedge b_x)\theta \leftarrow (h_2; \ldots ; h_n)\theta]$

Dalla descrizione formale della semantica operazionale segue l'interprete AgentSpeak(L) di figura 2.4, dove la funzione *top* ritorna il top dello stack di un'intenzione; la funzione *head* ritorna l'intestazione di un piano; la funzione *body* ritorna il corpo di un piano; le funzioni *first* e *rest* ritornano rispettivamente il primo elemento di una sequenza e una sequenza senza primo elemento; la funzione *push* ha come argomenti un'istanza di piano ed una intenzione con l'effetto di aggiungere l'istanza del piano all'intenzione; la funzione *pop* ha come argomento un'intenzione ed ha l'effetto di decurtare lo stack del top. Nell'istruzione "$x = pop(S_I(\mathrm{I}))$;" x viene istanziata con il top dell'intenzione selezionata da $S_I(\mathrm{I})$ e dopo l'esecuzione di *pop* tale intenzione rimane senza top.

algorithm interprete()

while E $\neq \phi$ do

> $\varepsilon = \langle d, i\rangle = S_\varepsilon(\mathrm{E})$;
> E = E/ε;
> $O_\varepsilon = \{p\theta \mid \theta$ unificatore applicabile per ε e p$\}$
> if evento-esterno(ε) then I = I $\bigcup$ $[S_O(O_\varepsilon)\sigma]$, σ unificatore applicabile per ε;
> else $push(S_O(O_\varepsilon)\sigma$,i), σ unificatore applicabile per ε;
> case $first(body(top(S_I(\mathrm{I})))) =$ true
>> $x = pop(S_I(\mathrm{I}))$;
>> $push(head(top(S_I(\mathrm{I})))\theta \leftarrow rest(body(top(S_I(\mathrm{I}))))\theta,\ S_I(\mathrm{I}))$
>>> dove θ è un mgu tale che $head(x)\theta = head(top(S_I(\mathrm{I})))\theta$;
>
> case $first(body(top(S_I(\mathrm{I})))) =$!g(t)
>> E = E $\bigcup$ $\langle$+!g(t),$S_I(\mathrm{I})\rangle$
>
> case $first(body(top(S_I(\mathrm{I})))) =$?g(t)
>> pop($S_I(\mathrm{I})$);
>> $push(head(top(S_I(\mathrm{I}))))\theta \ \leftarrow rest(body(top(S_I(\mathrm{I}))))\theta,\ S_I(\mathrm{I}))$
>>> dove θ è una sostituzione tale che B $\models$ g(t)θ.
>
> case $first(body(top(S_I(\mathrm{I})))) = a(t)$
>> $pop(S_I(\mathrm{I}))$;
>> $push(head(top(S_I(\mathrm{I}))))\leftarrow rest(body(top(S_I(\mathrm{I})))),\ S_I(\mathrm{I}))$
>> A = A $\bigcup$ $\{a(t)\}$

endwhile

Figura 2.4: Interprete AgentSpeak(L).

2.3.3 Una teoria per la dimostrazione delle proprietà di un agente AgentSpeak(L)

Al fine di dimostrare le proprietà di un agente AgentSpeak(L) viene mostrato un sistema di transizione etichettato che descrive il funzionamento dell'agente attraverso la definizione di configurazione per agenti BDI.

> Un sistema di transizione etichettato BDI è una coppia $\langle \Gamma, \vdash \rangle$ dove Γ è un insieme di configurazioni e $\vdash \subseteq \Gamma \times \Gamma$.

una configurazione è :

> una tupla $\langle E_i, B_i, I_i, A_i, i \rangle$, dove $E_i \subseteq$ E,
> $B_i \subseteq$ B, $I_i \subseteq$ I, $A_i \subseteq$ A ed i è l'etichetta della transizione.

Nella definizione di configurazione non compare l'insieme dei piani P, in quanto l'insieme viene considerato costante mentre l'agente passa da una configurazione all'altra.
Il sistema di transizione è costituito da un insieme di meta-regole che descrivono come l'agente passa da una configurazione all'altra.

L'aggiunta di una intenzione all'insieme I viene descritta dalla seguente meta-regola:

$$\frac{\langle \{\dots, \langle +!g(t), T \rangle, \dots\},\ B_i,\ I_i,\ A_i,\ i \rangle}{\langle \{\dots\},\ B_i,\ I_i \bigcup \{[p\sigma\theta]\},\ A_i,\ i+1 \rangle}$$

dove $p = +!g(s) : b_1 \wedge \dots \wedge b_m \leftarrow h_1; \dots ; h_n = S_O(O_\varepsilon)$, $S_\varepsilon(\mathrm{E}) = \langle\ +!g(t), T \rangle$, $g(t)\sigma = g(s)\sigma$, $\forall (b_1 \wedge \dots \wedge b_m)\theta$ è una conseguenza logica di B_ie T è l'intenzione banale.

L'aggiunta di un piano come top di una intenzione già esistente viene descritta dalla seguente meta-regola:

$$\frac{\langle \{\dots, \langle +!g(t), j \rangle, \dots\},\ B_i,\ \{\dots, [p_1 \ddagger \dots \ddagger p_z], \dots\},\ A_i,\ i \rangle}{\langle \{\dots\},\ B_i,\ \{\dots, [p_1 \ddagger \dots \ddagger p_z \ddagger p\sigma\theta], \dots\},\ A_i,\ i+1 \rangle}$$

dove $p_z = f : c_1 \wedge \dots \wedge c_y \leftarrow !g(t); h_2; \dots ; h_n$, $p = +!g(s) : b_1 \wedge \dots \wedge b_m \leftarrow k_1 \wedge \dots \wedge k_x$, $S_\varepsilon(\mathrm{E}) = \langle\ +!g(t), j \rangle$, j è $[p_1 \ddagger \dots \ddagger p_z]$, $g(t)\sigma = g(s)\sigma$ e $\forall (b_1 \wedge \dots \wedge b_m)\theta$ è una conseguenza logica di B_i.

L'esecuzione di un obiettivo della forma $!\phi$ è descritta da:

$$\frac{\langle E_i,\ B_i,\ \{\dots, [p_1 \ddagger \dots \ddagger p_z], \dots\},\ A_i,\ i \rangle}{\langle E_i \cup \{\langle +!g(t), j \rangle\},\ B_i,\ \{\dots, [p_1 \ddagger \dots \ddagger p_z], \dots\},\ A_i,\ i+1 \rangle}$$

dove $S_I(I_i) = j = [p_1 \ddagger \ldots \ddagger p_z]$ e $p_z = f : c_1 \wedge \ldots \wedge c_y \leftarrow !g(t); h_2; \ldots ; h_n$.

Le meta-regole che descrivono l'esecuzione dell'obiettivo $?\phi$, delle azioni e del piano con il corpo costituito dall'istruzione true vengono tralasciate, così come sono tralasciate le meta-regole riguardanti l'aggiunta di credenze, l'eliminazione degli obiettivi oppure delle credenze.

Con il sistema di meta-regole descritto possono essere definiti i concetti di derivazione quindi di refutazione per programmi AgentSpeak(L):

> una derivazione BDI è una sequenza finita o infinita di configurazioni $\gamma_0, \ldots, \gamma_n, \ldots$

La nozione di refutazione in AgentSpeak(L) è relativa ad una particolare intenzione, cioè una refutazione ha inizio quando un'intenzione viene adottata e finisce quando lo stack associato a quell'intenzione diviene vuoto.

2.3.4 Osservazioni

Attraverso l'uso dei piani, AgentSpeak(L) come dMARS offre una notevole capacità proattiva. Inoltre la definizione di una semantica formale definita attraverso l'uso di un sistema di transizione, fornisce la possibilità di dimostrare le proprietà comportamentali dell'agente e rende sicuro e modulare l'ampliamento del linguaggio: ad esempio l'aggiunta di eventi interni riguardanti l'eliminazione delle intenzioni, oppure di eventi riguardanti il successo ed il fallimento di azioni, piani ed obiettivi.

Il fiunzionamento dell'interprete AgentSpeak(L) differisce da quello dMARS nei momenti in cui vengono processati eventi ed intenzioni: dMARS processa le intenzioni solo mentre l'insieme degli eventi è vuoto; l'interprete AgentSpeak(L) è costituito, invece, da un ciclo le cui operazioni consistono nel sceglie un evento dall'insieme degli eventi, a seconda che sia esterno o interno creare oppure aggiornare l'intenzione corrispondente, quindi selezionare una intenzione ed eseguire la prima istruzione del top dell'intenzione.

Nella semantica di AgentSpeak(L)l'insieme degli eventi viene aggiornato solo dagli eventi interni, mentre in [Rao96] viene esplicitamente asserito che gli agenti percepiscono eventi manifestati sia da utenti che da cambiamenti nell'ambiente.

2.4 Concurrent MetateM

Concurrent MetateM è un linguaggio per modellare sistemi distribuiti le cui componenti interagiscono concorrentemente e autonomamente [MTF97].
Tali componenti vengono chiamati oggetti e comunicano con gli altri oggetti del sistema attraverso un protocollo di comunicazione broadcasting: nel messaggio trasmesso viene rappresentato l'identificatore del destinatario. Il messaggio viene ricevuto da tutti gli oggetti del sistema ma solo il destinatario lo legge.

Gli oggetti Concurrent MetateM sono definiti da una *componente interfaccia* e da una *componente interna*. La componente interfaccia di un oggetto contiene informazioni sul nome dell'oggetto, sui messaggi che può ricevere e su quelli che può inviare. Ad esempio, un'interfaccia per l'oggetto stack potrebbe essere della forma:

$$\text{stack}(pop, push)\ [popped, full]$$

dove $\{pop, push\}$ è l'insieme dei messaggi che l'oggetto può ricevere, mentre $\{popped, full\}$ è l'insieme dei messaggi che l'oggetto può inviare agli altri oggetti del sistema.

La componente interna consiste nel meccanismo in grado di ricevere messaggi, eseguire azioni in accordo con il contenuto dei messaggi ricevuti, di creare ed inviare nuovi messaggi. Il meccanismo di calcolo della componente interna è basato su MetateM, un linguaggio che utilizza formule della logica temporale.
La logica temporale utilizzata in Concurrent MetateM viene chiamata "logica MetateM del primo ordine" (FML). È basata su di un modello lineare e discreto del tempo: il tempo è modellato come una sequenza infinita e discreta di stati.

2.4.1 MetateM

MetateM [Fis93] è un linguaggio di programmazione basato sulla logica temporale costituito da regole della forma:

$$p \Rightarrow f$$

dove p è una formula temporale riferita al passato, mentre f è una formula temporale riferita al presente e/o al futuro.
Alcuni operatori della logica temporale che verranno usati nel seguito sono:

- **start**: rappresenta l'istante iniziale del tempo;
- $\Box$ (always): la formula temporale $\Box\varphi$ è soddisfatta nell'istante di tempo corrente se la formula φ verrà soddisfatta **sempre** in futuro.

- $\Diamond$ (sometime): la formula $\Diamond\varphi$ è soddisfatta nell'istante corrente se **esiste un istante di tempo futuro** in cui φ verrà soddisfatta.

- $\odot$ (strong last-time): la formula $\odot\varphi$ è soddisfatta se esiste l'**istante di tempo precedente a quello corrente** ed in tale istante di tempo φ era soddisfatta.

- $\bigoplus$ (last-time): la formula $\bigoplus\varphi$ è soddisfatta se $\odot\varphi$ è soddisfatta oppure se **start** è soddisfatta.

- $\mathcal{W}$ (unless): la formula $\varphi\mathcal{W}\psi$ è soddisfatta nell'istante corrente se φ è soddisfatta dall'istante corrente fino all'istante in cui ψ è soddisfatta.

- $\bigcirc$ (immediate future): la formula $\bigcirc\varphi$ è soddisfatta nello stato corrente se φ è soddisfatta nello stato successivo a quello corrente.

L'esecuzione di un insieme di formule $\{p_i \Rightarrow f_i \mid 1 \leq i \leq n\}$ consiste nel costruire un modello per:

$$\Box \bigwedge_{i=1}^{n} (p_i \Rightarrow f_i)$$

Ogni regola dell'insieme viene confrontata con lo stato corrente che rappresenta la storia passata del sistema. Se la parte sinistra della formula (p_i) è in accordo con lo stato corrente, allora la parte destra (f_i) viene soddisfatta, e lo stato viene aggiornato tenendo conto delle formule appena soddisfatte.
L'esecuzione di un oggetto consiste nel leggere i messaggi ricevuti, creare la congiunzione delle parti destre (f_i) delle regole le cui parti sinistre (p_i) sono soddisfatte dallo stato corrente, quindi scegliere un elemento di tale congiunzione, renderlo vero e sceglierne un altro fino a che la congiunzione rimane vuota.

Un esempio di programma MetateM è definito dalle seguenti regole:

$$\begin{aligned}&\textbf{start} \Rightarrow \mathit{popped}\\ &\odot \mathit{pop} \Rightarrow \Diamond \mathit{popped}\\ &\odot \mathit{push} \Rightarrow \mathit{full} \lor \mathit{popped}\end{aligned}$$

Nella prima regola, *popped* viene soddisfatta nell'istante iniziale del tempo (**start**). La seconda regola asserisce che se *pop* era vero nell'istante di tempo precedente a quello corrente allora deve esistere un istante di tempo futuro in cui *popped* sarà soddisfatta. Infine la terza regola asserisce che se *push* era vera nell'istante di tempo precedente a quello corrente allora $full$ oppure *popped* devono essere soddisfatti nell'istante corrente di tempo.

2.4.2 Semantica di Concurrent MetateM

I predicati utilizzati in Concurrent MetateM sono divisi in tre categorie:

- *Predicati* ambiente che rappresentano messaggi in ingresso.
  Un predicato ambiente è vero sse è stato ricevuto il messaggio corrispondente. Ad esempio il predicato ambiente *push* viene soddisfatto se è stato ricevuto il messaggio push.

- *Predicati componente* che rappresentano messaggi inviati dall'oggetto.
  Quando un predicato componente viene soddisfatto, come effetto collaterale viene inviato all'ambiente tramite broadcasting il corrispondente messaggio. Per esempio se la formula *popped* viene soddisfatta e se *popped* è un predicato componente allora il messaggio popped viene inviato.

- *Predicati interni* che non hanno effetti all'esterno dell'oggetto.
  Sono utilizzati in formule che definiscono il comportamento interno dell'oggetto e non sono utilizzati né per l'invio né per il ricevimento dei messaggi.

Per definire la semantica di Concurrent MetateM gli oggetti verranno considerati come strutture i cui campi sono:

- *regole*: l'insieme delle regole che costituiscono l'oggetto;

- *amb*: l'insieme dei predicati ambiente;

- *comp*: l'insieme dei predicati componente;

- *int*: l'insieme dei predicati interni;

Per esempio l'insieme dei predicati componente dell'oggetto obj viene rappresentato dall'espressione obj.*comp*.
Per evitare la sovrapposizione dei nomi dei predicati negli oggetti, questi vengono ridenominati in modo da contenere anche il nome dell'oggetto che contiene il predicato. Per esempio il predicato p utilizzato nell'oggetto O viene ridenominato p-O.
La funzione semantica per il linguaggio proposizionale Concurrent MetateM utilizzata in [Fis93] è $\mathcal{M}$; associa alla categoria sintattica di cui si vuole il significato (ad esempio regole, oggetti, l'intero sistema, ...) una formula logica temporale.

Inizialmente verrà considerato un sistema Concurrent MetateM semplificato in cui: gli oggetti vengono eseguiti sincronamente; i messaggi vengono ricevuti istantaneamente; i predicati ambiente possono comparire soltanto nella parte sinistra delle regole; gli oggetti non possono cambiare dinamicamente la loro interfaccia. La semantica di un sistema Concurrent MetateM S, consistente in $O_1, \ldots, O_n$ oggetti, con i vincoli appena considerati è data da:

$$\begin{aligned}\mathcal{M}_{\text{sistema}}(S) &= \bigwedge_{i=1}^{n} \mathcal{M}_{\text{oggetti}}(rinom(O_i))\\ \mathcal{M}_{\text{oggetti}}(O) &= \mathcal{M}^{O}_{\text{regole}} \ \wedge \ \text{comm}(O)\end{aligned}$$

dove $rinom(O_i)$ rinomina i predicati nell'oggetto O_i, mentre il predicato $comm(O)$ genera dei vincoli per i messaggi inviati e ricevuti dall'oggetto O:

$$\begin{aligned}comm(O) \ \equiv \ & \bigwedge_{p-O \ \in \ O.amb} \Box(p \ \Rightarrow \ p\text{-}O) \wedge\\ & \bigwedge_{q-O \ \in \ O.comp} \Box(q\text{-}O \Rightarrow \ q)\end{aligned}$$

Intuitivamente il predicato $comm(O)$ fa sì che quando viene ricevuto un messaggio, corrispondente al predicato ambiente p, la versione ridenominata del predicato viene resa vera. Similmente quando un predicato componente viene reso vero allora il messaggio corrispondente viene inviato con il nome originale.

Eliminando il vincolo sull'esecuzione sincrona, ogni oggetto può progredire alla propria velocità e può utilizzare il proprio clock locale. Di conseguenza ad ogni oggetto O è associato un clock rappresentato dal predicato $tick(O)$. In questo modo ad ogni stato temporale discreto dell'esecuzione di un oggetto viene associato un intervallo di tempo determinato da *tick*. Se un predicato è soddisfatto in un particolare stato durante l'esecuzione di un particolare oggetto, il predicato verrà considerato soddisfatto nell'intervallo di tempo corrispondente a quello stato.
Siccome gli oggetti vengono eseguiti asincronamente, gli intervalli del clock di diversi oggetti possono sovrapporsi. Per rappresentare questa cratteristica in [Fis93] viene utilizzata una logica temporale basata su di un modello denso del tempo. La logica utilizzata è la "Temporal Logic of the Reals" (TLR). In questa logica non esiste il concetto di intervallo di tempo tra un istante e quello successivo: tra due istanti di tempo successivi vi è un infinito numero di istanti di tempo. Nella logica TLR gli unici operatori temporali riferiti ad intervalli di tempo sono:

- $A\,\mathcal{U}^{+}B$, dove A è vera in qualsiasi istante nell'intervallo di tempo tra l'istante corrente e l'istante in cui B diventerà vera.

- $C\ \mathcal{S}^{-}D$, in cui C è vera in qualsiasi istante nell'intervallo di tempo tra l'istante in cui D diventa vera e l'istante di tempo corrente.

L'unico vincolo della TLR che viene utilizzato per poter modellare la sovrapposizione degli intervalli del clock è quello della *finita variabilità dei predicati*: tra due istanti di tempo un predicato può variare il proprio valore di verità un numero finito di volte.
Definiti questi due operatori della TLR e dato il vincolo della variabilità finita dei predicati, il predicato *tick* può essere modellato come segue:

$$\begin{aligned}&(\textbf{start} \Rightarrow \ tick) \wedge\\ &\Box(\neg tick \ \Rightarrow \ (\neg tick)\,\mathcal{U}^{+}tick) \wedge\\ &\Box(tick \ \Rightarrow \ tick\,\mathcal{U}^{+}(\neg tick))\end{aligned}$$

dove **start** e l'operatore temporale $\Box$ sono estesi nel modo ovvio per la logica TLR.
I periodi in cui *tick* vale si alternano infinitamente spesso con periodi in cui vale $\neg tick$, e ogni periodo in cui vale *tick* oppure $\neg tick$ è di lunghezza arbitraria ma finita. In questo modo l'intero dominio temporale del sistema di oggetti viene "coperto" da periodi $tick/\neg tick$, con la definizione del vincolo di variabilità finita in ogni intervallo finito del dominio, il valore di verità del predicato *tick* non può alternarsi infinite volte.
Con la definizione del predicato *tick* può essere modellato il clock di ogni oggetto del sistema. La valutazione del predicato p nello stato s è modellata come valutazione del predicato p-O nell'intervallo di clock (rappresentato dal predicato *tick*) associato ad s.

Avendo modellato il concetto di clock può essere definita la semantica delle regole che compongono un oggetto all'interno del sistema. La valutazione semantica di una regola è rappresentata da una formula della logica TLR; formula poi valutata all'interno dell'appropriato intervallo di clock. Per fare ciò devono essere definiti tre predicati ausiliari:

- $next\_tick(O, X)$: soddisfatto se la formula X è soddisfatta nell'intervallo di clock dell'oggetto O successivo a quello corrente.
- $last\_tick(O, X)$: soddisfatto se la formula X è soddisfatta nell'intervallo di clock dell'oggetto O precedente a quello corrente.
- $during\_tick(O, X)$: soddisfatto se la formula X è soddisfatta durante l'intervallo di clock corrente dell'oggetto O.

Utilizzando questi predicati, la semantica dell'insieme di regole che compongono la componente interna di un oggetto Concurrent MetateM può essere definita come segue:

$$\begin{aligned}\mathcal{M}_{regole}(\mathrm{O}) \;=\; & \textstyle\bigwedge_{R\,\in\,O.regole}\, next\_tick(O, \mathcal{M}^{O}_{regola}(R)) \wedge \\ & next\_tick(O, \mathcal{M}_{regole}(O))\end{aligned}$$

$\mathcal{M}_{regole}$ è definita ricorsivamente in modo da ottenere una valutazione semantica dell'insieme di regole in ogni istante di esecuzione dell'oggetto.
Siccome qualsiasi formula della TLR può essere ridotta in una delle seguenti forme normali [Fis92] in cui k_i, m_j, l sono letterali:

$$\mathbf{start} \;\Rightarrow\; \textstyle\bigvee_{j=1}^{r} m_j$$

$$\mathbf{start} \;\Rightarrow\; \Diamond l$$

$\odot \bigwedge_{i=1}^{q} k_i \;\Rightarrow\; \bigvee_{j=1}^{r} m_j$

$\odot \bigwedge_{i=1}^{q} k_i \;\Rightarrow\; \Diamond l$

La semantica delle regole della componente interna è data dalle seguenti valutazioni:

$$\mathcal{M}^{O}_{regola}(\mathbf{start} \Rightarrow D) = [(\neg last\_tick(O, \mathbf{true})) \Rightarrow during\_tick(O, D)]$$

$$\mathcal{M}^{O}_{regola}(\mathbf{start} \Rightarrow \Diamond l) = [(\neg last\_tick(O, \mathbf{true})) \Rightarrow \Diamond(tick(O) \wedge during\_tick(O, l))]$$

$$\mathcal{M}^{O}_{regola}(\bigodot C \Rightarrow \; D) = [(last\_tick(O, C)) \Rightarrow during\_tick(O, D)]$$

$$\mathcal{M}^{O}_{regola}(\bigodot C \Rightarrow \; \Diamond l) = [(last\_tick(O, C)) \Rightarrow \Diamond(tick(O) \wedge during\_tick(O, l))]$$

dove C è una congiunzione di letterali, D una disgiunzione di letterali e l un letterale.
Inoltre viene introdotto un vincolo che assicura che, se un predicato è vero in un istante di tempo interno all'intervallo del clock, tale predicato rimane vero per tutta la durata dell'intervallo:

$$(p \;\wedge\; tick(O)) \;\Rightarrow\; p\,\mathcal{U}^{+}\neg tick(O)$$

Eliminando anche il vincolo riguardante la ricezione istantanea dei messaggi, bisogna modellare il fatto che, quando viene inviato un messaggio, esiste un istante di tempo futuro in cui il messaggio verrà ricevuto dall'oggetto destinatario. Tale modellazione si ottiene modificando il predicato $comm(O)$ definito precedentemente:

$$\bigwedge_{p-O \;\in\; O.amb} \Box(p \;\Rightarrow\; \Diamond p\text{-}O) \;\wedge\; \bigwedge_{q-O \;\in\; O.comp} \Box(q\text{-}O \Rightarrow \; \Diamond q)$$

dove *p-o* e *q-o* sono rispettivamente un predicato ambiente e un predicato componente dell'oggetto O. L'unico problema relativo a questa definizione si ha se il messaggio viene ricevuto quando l'oggetto non è attivo, cioè quando il predicato *tick* è falso. Per risolvere questo problema il predicato $comm(O)$ viene ridefinito in modo che il messaggio venga letto solo quando il predicato tick vale:

$$\bigwedge_{p-O \;\in\; O.amb} \Box(p \;\Rightarrow\; \Diamond(tick(O) \;\wedge\; p\text{-}O)) \;\wedge$$
$$\bigwedge_{q-O \;\in\; O.comp} \Box(q\text{-}O \Rightarrow \; \Diamond q).$$

Per ottenere una sincronizzazione tra gli oggetti del sistema bisogna eliminare il vincolo riguardante la non presenza di predicati ambiente nella parte destra delle regole. L'effetto sincronizzazione si ottiene se un predicato ambiente compare nella parte destra di una regola; in questo modo affinché il predicato ambiente sia soddisfatto l'oggetto deve attendere l'arrivo del messaggio appropriato. Ad esempio nelle regole:

$$\begin{array}{l} \ldots \Rightarrow ask \\ \bigodot answer \Rightarrow do\_it \\ \bigodot ask \Rightarrow answer \end{array}$$

ask è un predicato componente, ed *answer* è un predicato ambiente. Se un messaggio *ask* è stato inviato l'unico modo di soddisfare l'ultima regola è quello di ricevere il messaggio *answer*, fino a quando tale messaggio non arriva l'esecuzione dell'oggetto è sospesa.
La sincronizzazione è assicurata in quanto un oggetto non può definire autonomamente il valore di verità di un predicato ambiente: il soddisfacimento di un predicato ambiente consiste nella ricezione del corrispondente messaggio. Per fare si che un oggetto non possa autonomamente modificare il valore di verità di un predicato ambiente viene introdotto il seguente vincolo:

$$[(\neg p)\mathcal{S}^{-}(tick(O) \ \wedge \ p - O) \ \vee \ \bigotimes \neg p - O] \Rightarrow \ \neg p - O$$

il predicato ambiente p deve essere valutato falso se non è stato ricevuto un nuovo messaggio in seguito all'ultimo messaggio letto (l'operatore $\bigotimes$ significa "sempre nel passato" e può essere definito tramite l'operatore $\mathcal{S}^{-}$).

L'ultimo vincolo da rimuovere per ottenere Concurrent MetateM riguarda la definizione dinamica delle interfacce degli oggetti. Michael Fisher in [Fis93] non affronta questo problema formalmente, ma si limita ad enfatizzare il fatto che per definire tale capacità l'esecuzione di una regola deve cambiare la rappresentazione sintattica dell'oggetto. Ad esempio deve cambiare i campi *amb* e *comp* della struttura che definisce l'oggetto. Per fare questo la funzione semantica $\mathcal{M}^{O}_{regola}$ deve essere ridefinita in modo che ritorni non solo una formula TLR ma anche la versione aggiornata della struttura associata all'oggetto.

2.4.3 Un agente BDI tramite Concurrent MetateM

La struttura di Concurrent MetateM permette una facile descrizione di agenti con architetture diverse. La realizzazione di agenti reattivi risulta semplice, ma

anche agenti più complessi come quelli BDI (vedi sezioni 1.2.1, 2.2) possono essere descritti in Concurrent MetateM. In [MTF97] viene affrontato questo aspetto.
Per definire un agente BDI tramite l'uso di Concurrent MetateM viene utilizzata un'estensione della logica FML chiamata MML (Meta-MetateM logic). La logica MML viene utilizzata per definire un livello meta in cui vengono rappresentati le credenze, i desideri, le intenzioni ed i piani dell'agente BDI.
Un agente Concurrent MetateM, come visto nella sezione 2.4, è composto da una componente interfaccia e da una componente interna.
La componente interfaccia dell'agente BDI è :

BDI-agent(*percept*, *goal*)[*action*]

L'agente riceve dall'ambiente due tipi di messaggi: *percept(p)* e *goal(intrinsec,G)*. Messaggi della forma *percept(p)* rappresentano il modo in cui l'agente percepisce l'ambiente. Con tale messaggio l'agente acquisisce le proprie credenze. Con messaggi della forma *goal(intrinsec,G)* l'agente acquisisce dall'ambiente i propri obiettivi. Se l'agente appartiene ad un contesto multi-agente può acquisire i propri obiettivi tramite comunicazione con altri agenti; in questo caso nei messaggi il termine *intrinsec* viene sostituito dal nome dell'agente mittente. L'agente può produrre il messaggio *action(A)* con il quale modifica l'ambiente in cui è collocato.

La componente interna dell'agente consiste nella rappresentazione e nel mantenimento delle credenze, degli obiettivi, delle intenzioni e dei piani.
Le credenze di un agente sono rappresentate da un insieme di atomi ground della forma:

belief(B)

dove B è un letterale. Ad esempio una credenza potrebbe essere *belief(¬piove)*.
L'agente acquisisce credenze come asserzione oppure come percezione. Il predicato *assert(B)* viene utilizzato nel corpo di un piano e fa sì che venga asserita la credenza B. Inoltre la credenza B può anche essere percepita tramite la ricezione del messaggio *percept(B)*.
Le regole con le quali viene definito il mantenimento delle credenze sono:

$$(assert(B) \ \vee \ percept(B)) \ \Rightarrow \ belief(B)$$

$$\neg(assert(\neg B) \ \vee \ percept(\neg B)) \ \mathcal{S} \ belief(B) \ \Rightarrow \ belief(B)$$

La prima regola dice che se il letterale B è asserito oppure percepito allora diventa una credenza: per ogni stato in cui i predicati *assert(B)* o *percept(B)* valgono il predicato *belief(B)* vale. La seconda regola dice che una credenza vale finchè non viene asserita la sua negazione ($\mathcal{S}$ è l'operatore temporale *since*, $a \ \mathcal{S} \ b$ è vera se b

vale fino a quando a vale) senza questa regola le credenze non sarebbero ricordate tra stati consecutivi.
In questa descrizione dell'agente BDI vengono considerati due tipi di obiettivi: obiettivi intrinseci, obiettivi operazionali (sotto obiettivi). Un obiettivo intrinseco viene generato all'esterno dell'agente, ad esempio da un altro agente. Se l'agente ha dei piani che possono essere evocati da tale obiettivo viene generata una nuova intenzione per soddisfare quell'obiettivo.
L'obiettivo operazionale G viene adottato se nel corpo del piano che viene eseguito vi è il sotto obiettivo G; in questo caso, se il piano fa parte dell'intenzione I, G diventa un sotto obiettivo di I.
Sia gli obiettivi intrinseci che quelli operazionali sono rappresentati dall'atomo:

$$goal(I, G)$$

dove G è il nome dell'obiettivo, I viene istanziata con la costante *intrinsec* per un obiettivo intrinseco, e viene istanziata con l'intenzione che ha generato G per un obiettivo operazionale.
Il comportamento di un agente nei confronti dei propri obiettivi è specificato dalle seguenti tre regole:

$$\bigoplus goal(I, G) \;\Rightarrow\; \Diamond(achieved(goal(G)) \vee failed(goal(G)));$$

$$\neg achieved(purpose(goal(G)))\ \mathcal{S}\ goal(I, G) \;\Rightarrow\; \neg achieved(goal(G));$$

$$\neg failed(plan(N))\ \mathcal{S}\ option(N, I, goal(G), \varphi) \;\Rightarrow\; \neg failed(goal(G))$$

La prima regola asserisce che se il predicato $goal(I, G)$ è stato soddisfatto nell'istante precedente a quello corrente, allora esiste un istante futuro in cui i predicati $achieved(goal(G))$ oppure $failed(goal(G))$ saranno soddisfatti; la seconda regola asserisce che, se dall'istante in cui è stato adottato l'obiettivo fino all'istante corrente, l'esecuzione del piano che ha come utilizzo il raggiungimento dell'obiettivo scelto non è terminata con successo ($\neg achieved(purpose(goal(G)))$), il predicato $achieved(goal(G))$ è falso; la terza regola asserisce che se iniziata l'esecuzione del piano scelto per soddisfare l'obiettivo e tale piano non è fallito, il predicato $failed(goal(G))$ nell'istante corrente è falso.

Per rappresentare i piani viene utilizzato il predicato:

$$plan(Nome, Utilizzo, \phi_{contesto}, \phi_{corpo})$$

Ogni piano deve avere un unico $Nome$, un $Utilizzo$ indicante l'evento che seleziona il piano, un contesto, cioè un'espressione riguardante le credenze che devono essere vere quando il piano viene selezionato ed un *corpo* che consiste in

una formula Concurrent MetateM.
Sono definiti due tipi di piani: *piani reattivi* e *piani diretti dagli obiettivi*. I *piani reattivi* sono attivati da un cambiamento nelle credenze dell'agente e sono selezionati da eventi della forma $react(\phi_{credenze})$, dove $\phi_{credenze}$ è una espressione riguardante le credenze dell'agente. I *piani diretti dagli obiettivi* sono attivati da un obiettivo dell'agente, ed il loro scopo è quello di soddisfare l'obiettivo, sono selezionati da eventi della forma $goal(G)$. Ad esempio se un piano ha come istanza della variabile $Utilizzo$ il predicato $goal(muovi\_avanti)$, tale piano verrà selezionato nello stato in cui il predicato $goal(I, muovi\_avanti)$ sarà vero.
Nel modello Concurrent MetateM dell'architettura BDI il corpo di un piano è rappresentato da una formula della logica temporale. Formula che non viene eseguita direttamente da Concurrent MetateM ma da un *meta-interprete* utilizzato con il fine di controllare l'esecuzione del piano, ad esempio, sospendere l'esecuzione di un piano per eseguirne uno più urgente.
Il *meta-interprete* viene evocato tramite il predicato:

$$exec\_plan(Nome, Intenzione, Utilizzo, \phi_{corpo})$$

i cui argomenti sono rispettivamente il nome del piano, l'intenzione per la quale il piano viene eseguito, l'utilizzo del piano cioè l'evento che causa la selezione del piano ed infine la formula temporale che definisce il corpo del piano.
Le regole che descrivono il *meta-interprete* sono di due tipi: regole temporali e regole di stato. Le regole temporali interpretano gli operatori temporali del corpo del piano e fanno si che il piano prosegua l'esecuzione attraverso stati temporali successivi. Le regole di stato interpretano le congiunzioni all'interno della formula temporale che caratterizza il corpo del piano. In [MTF97] viene solamente fatto riferimento alla regola temporale riguardante l'interpretazione dell'operatore temporale $\bigcirc$, per una descrizione completa di tali regole viene fatto riferimento a [Mul97a].

$$\bigoplus(exec\_plan(N, I, U, \bigcirc\phi) \wedge \neg failed(plan(N))) \Rightarrow \Diamond exec\_plan(N, I, U, \phi)$$

Questa regola asserisce che: se nello stato precedente a quello corrente il piano N non è fallito ed in quello successivo in cui il piano ha cominciato la propria esecuzione, ϕ doveva essere eseguita, allora esisterà un istante futuro in cui ϕ verrà eseguita.

Le nuove intenzioni sono create asserendo il predicato

$$new\_intention(I)$$

Il comportamento dell'agente nei confronti delle proprie intenzioni è caratterizzato dalle seguenti regole:

$$\neg done(intention(I))\ \mathcal{S}\ new\_intention(I)\ \Rightarrow\ current\_intention(I);$$

$$exec\_plan(name(N), intention(I), goal(G), \phi_{corpo})\ \Rightarrow\ active\_intention(I);$$

$$current(intention(I)) \wedge \neg active\_intention(I)\ \Rightarrow\ suspended\_intention(I).$$

L'esecuzione di una intenzione può terminare con un successo oppure con un fallimento, in entrambi i casi il predicato $done(intention(I))$ è reso vero. Il predicato $new\_intention(I)$ è vero solo nello stato in cui viene adottata l'intenzione, mentre durante gli stati successivi è vera la formula $\Diamond(done(intention(I)))$. Se una intenzione è stata adottata ma non è ancora terminata, il predicato $current\_intention(I)$ è vero; inoltre se per soddisfare l'intenzione I viene attivato un piano, il predicato $active\_intention(I)$ è vero. Se per l'intenzione corrente nessun piano è attivo, l'intenzione I viene considerata sospesa e diventa vero il predicato $suspended\_intention(I)$.
I piani possono essere invocati in tre modi: un cambiamento delle credenze dell'agente (piano reattivo), un obiettivo intrinseco causa la selezione di un piano diretto dagli obiettivi, un obiettivo operazionale causa la selezione di un piano in seguito all'esecuzione di un sotto_obiettivo. Nei primi due casi viene creata una nuova intenzione per soddisfare l'obiettivo, mentre nel terzo caso il piano selezionato viene aggiunto ad una intenzione già esistente. In [MTF97] viene descritta la regola che seleziona un piano diretto dagli obiettivi selezionato da un obiettivo intrinseco. Per gli altri due casi viene fatto riferimento a [Mul97a]:

$$\begin{aligned}
&\bigoplus(plan(name(N), goal(G), \phi_{context}, \phi_{body}) \wedge goal(intrinsec, G) \wedge \phi_{context})\ \Rightarrow\\
&(option(name(N), intention(goal(G)), goal(G), \phi_{body}) \wedge\\
&new_intention(goal(G)) \wedge\\
&\neg achieved(intention(goal(G)))\ \mathcal{W}\ achieved(goal(G)) \wedge\\
&\neg failed(intention(goal(G)))\ \mathcal{W}\ failed(goal(G)))
\end{aligned}$$

Se nell'istante precedente a quello corrente l'obiettivo intrinseco G viene adottato e il contesto $\phi_{context}$ è creduto dall'agente, il piano diventa una opzione. Il predicato $achieved(intention(I))$ è vero se il piano adottato per l'intenzione I è stato eseguito con successo, ed il predicato $failed(intention(I))$ è vero se tutti i piani di una intenzione sono falliti.
Se possono essere adottati più piani per soddisfare un obiettivo, devono essere adottati uno alla volta. Se uno di questi termina con successo non serve eseguire gli altri. Per ogni coppia di piani per lo stesso obiettivo il predicato $preferred\_option$ asserisce quale di loro deve essere provato per primo.
Il comportamento dell'agente nei confronti della scelta dei piani è descritto dalla seguente regola:

$$\bigoplus(option(N, I, goal(G), \phi_1) \ \wedge \ option(M, I, goal(G), \phi_2) \wedge$$
$$preferred\_option(N, M)) \ \Rightarrow$$
$$\neg exec\_plan(M, I, goal(G), \phi_1) \ \mathcal{W} \ failed(plan(N))$$

2.4.4 Osservazioni

L'utilizzo di MetateM per specificare la componente interna degli agenti rende Concurrent MetateM particolarmente adatto per la specifica di sistemi composti da agenti reattivi. Tuttavia la struttura astratta del linguaggio permette anche la specifica di agenti con capacità proattive, rendendo Concurrent MetateM un linguaggio generale di specifica per agenti, a differenza di dMARS e AgentSpeak(L), linguaggi specializzati per l'architettura BDI.

2.5 DESIRE

La specifica di un sistema multi-agente attraverso l'uso di DESIRE è basata sull'analisi dei task che il sistema deve realizzare [BKJT95].
I task vengono decomposti in una struttura gerarchica e ad ogni task vengono associati uno o più agenti.
Nella specifica vengono descritti la struttura gerarchica dei task, l'interazione tra i livelli di tale struttura, la conoscenza che un agente ha dell'ambiente in cui è collocato e quella riguardante gli altri agenti del sistema.
I task trattati in DESIRE sono di due tipi:

- *task complessi*: sono descritti da uno o piú "sub task" che possono assumere ancora la forma di *task complessi*;
- *task primitivi*: non possono essere decomposti in sub task, possono solo essere eseguiti.

I sistemi descritti con DESIRE sono detti "sistemi composizionali": il sistema è formato da diverse componenti ognuna in relazione con i task individuati. Le componenti sono descritte attraverso un linguaggio logico basato su un insieme di predicati che possono assumere tre valori di verità: vero, falso, indefinito.
Esistono due tipi di componenti:

- *componenti composte*: descrivono la conoscenza necessaria per eseguire i task complessi del sistema, l'interfaccia di input-output della componente e le informazioni riguardo i "sub task";
- *componenti primitive*: sono in relazione con i task primitivi, specificano cioè la conoscenza necessaria per eseguire i relativi task.

Ogni componente ha uno *stato interno* che descrive sia le informazioni generate durante l'esecuzione dei task che quelle ottenute dall'interazione con le altre componenti. La componente può inoltre trovarsi in uno tre stati di attivazione: *active*, *idle*, *awake*. Una componente passa dallo stato *active* a quello *idle* quando termina il processo per cui è stata attivata, mentre è nello stato *awake* quando attende informazioni.
Le componenti interagiscono tra loro utilizzando interfacce di input-output descritte da due segnature:

- *segnatura di input*: descrive i fatti che la componente può ricevere come input;
- *segnatura di output*: descrive i possibili fatti di output che la componente può computare.

Lo scambio di informazioni tra le interfacce delle componenti è basato su strutture chiamate *collegamenti di informazione.* I *collegamenti di informazione* specificano quali valori di verità un atomo usato in una componente assume in un'altra componente. Possono assumere tre stati: *idle*, *up_to_date*, *awake*. Quando un *collegamenti di informazione*è nello stato *up_to_date* trasferisce l'informazione e ritorna nello stato *idle*, mentre è nello stato *awake* se attende informazioni.

In un sistema composizionale la possibilità di cambiare sia lo stato interno delle componenti che lo stato dell'ambiente in cui sono collocate, è ottenuta dall'interazione tra differenti "livelli di ragionamento": livello-oggetto, meta-livello, meta-meta-livello eccetera. Il livello-oggetto "ragiona" sullo stato dell'ambiente ed il meta-livello "ragiona" sul livello-oggetto. Questa struttura di livelli di ragionamento può essere ripetuta: in questo modo il meta-livello ha il ruolo di livello-oggetto per il meta-meta-livello e così via. Il risultato del ragionamento al meta-livello può essere utilizzato per guidare il ragionamento al livello-oggetto. Il meta-livello utilizza come informazione i risultati ottenuti al livello-oggetto, sia per il proprio ragionamento, sia per fornire al livello-oggetto le informazioni che necessitano per computare i propri risultati.

La specifica di un sistema composizionale descrive sia la struttura interna delle componenti che le relazioni tra queste.
La struttura interna delle componenti è caratterizzata da:

- conoscenza riguardo alla struttura dei task;
- conoscenza riguardo alla sequenza in cui vanno eseguiti i "sub task";
- conoscenza riguardante lo scambio di informazione tra i "sub task";

- conoscenza riguardante la conoscenza della struttura dei task e la conoscenza sulla decomposizione dei task;
- conoscenza riguardante la delega dei ruoli.

mentre le relazioni tra componenti riguardano:

- decomposizione dei task: descrive la gerarchia dei task e le segnature di input e output per ogni sotto task;
- conoscenza relativa al controllo dei task: specifica l'attivazione della componente e delle sotto-componenti;
- collegamenti di informazione;
- allocazione della conoscenza per i task;
- allocazione dei task (tra gli agenti).

Gli agenti descritti da DESIRE sono chiamati *agenti composizionali* e sono modellati come *componenti composte* con l'aggiunta della conoscenza riguardante l'ambiente e gli altri agenti del sistema.
La comunicazione tra agenti è quindi modellata allo stesso modo della comunicazione tra componenti, cioè tramite l'utilizzo di *collegamenti di informazione*. Ad esempio se un agente necessita di alcune informazioni per completare il proprio task trasferisce la richiesta ad un secondo agente tramite un *collegamento di informazione*. L'informazione richiesta è inviata, tramite un *collegamento di informazione*, all'agente richiedente, e può essere ottenuta sia come informazione presente nelle conoscenze dell'agente sia dal completamento di un task.
Lo scambio d'informazione tra *agente composizionale* ed ambiente avviene in modo simile a quella tra agente e agente.
L'osservazione dell'ambiente può essere modellata come una richiesta di informazione trasferita all'ambiente attraverso un *collegamento di informazione*; la risposta viene quindi inviata al richiedente tramite un apposito collegamento. Un altro tipo di comunicazione tra agente e ambiente è l'esecuzione di una azione da parte dell'agente. L'effetto di tale esecuzione trasferisce informazione dall'agente all'ambiente modificandone lo stato.
La conoscenza relativa al controllo dei tasks rende possibile la specifica del comportamento degli agenti: riguarda le interazioni richieste per assicurare i fatti in input agli agenti e l'insieme dei fatti di cui vengono cercati i valori di verità (insieme degli obiettivi).
Una componente ha successo se riesce a completare tutti i propri tasks, altrimenti fallisce.

Un esempio di regola di controllo dei tasks è quella in cui il successo della componente *"own_process_control"* deve precedere l'attivazione della componente *"update_snapshot"*:

> **if** *evaluation(own_process_control, ts_update, succeded)*
> **then** *next-component-state(update_snapshot, active)*
> **and** *next-target-set(own_process_control, ts_poss_control)*
> **and** *next-link-state(incoming_info_snapshot_update, up_to_date)*

il cui significato è :

> **if** la componente *own_process_control* ha successo nel completare l'insieme degli obiettivi *ts_update*
> **then** viene specificata la nuova componente da attivare cioè *update_snapshot*;
> alla componente *own_process_control* è assegnato un nuovo insieme di obiettivi da completare *ts_poss_control*;
> le informazioni di cui necessita la componente *update_snapshot* sono ottenute attraverso l'attivazione del collegamento di informazione *incoming_info_snapshot_update*.

L'attivazione di una componente non sempre dipende dal completamento di un'altra: essa può essere attivata da un appropriato input.

2.5.1 Modellazione di un agente BDI tramite DESIRE

In [MTF97] viene presentato un agente BDI descritto utilizzando DESIRE. L'ambiente in cui è collocato l'agente è un piano su cui sono posizionati dei blocchi. L'agente modifica l'ambiente attraverso azioni che modificano la posizione dei blocchi sul piano.

Nella figura 2.5 sono rappresentate le componenti composte *agente_BDI* e *ambiente* con i relativi *collegamenti di informazione*. Il *collegamento di informazione eventi_ambiente* trasferisce informazioni dall'ambiente all'agente, *obiettivi* trasferisce gli obiettivi intrinseci dall'ambiente all'agente e *azioni* trasferisce dall'agente all'ambiente l'informazione ottenuta dall'esecuzione delle azioni. Tale informazione è in grado di modificare lo stato dell'ambiente.

Nella figura 2.6 vengono mostrate le cinque componenti che costituiscono l'agente BDI:

- Nella BASE DELLA CONOSCENZA sono mantenute le credenze dell'agente, ottenute come percezioni dall'ambiente e come asserzioni generate durante l'esecuzione dei piani.

- In OBIETTIVI sono mantenuti sia gli obiettivi intrinseci sia i "sotto-obiettivi". In questa componente vengono anche determinati quali obiettivi sono stati completati e quali selezioneranno nuovi piani.

- Nella LIBRERIA DEI PIANI i piani sono selezionati dai cambiamenti nella componente OBIETTIVI e nella componente BASE DELLA CONOSCENZA. Se il contesto di un piano è in accordo con le credenze dell'agente allora il piano viene selezionato.

- Nella STRUTTURA DELLE INTENZIONI viene scelto il piano che diventerà una intenzione. Viene inoltre definito quale intenzione è attiva istante per istante.

- In ESECUZIONE DEI PIANI i piani vengono eseguiti.

Il funzionamento dell'agente BDI è simile a quello descritto nelle sezioni 2.2 e 2.2.3.
Ogni componente descritta nella fig 2.6 è autonoma in quanto contiene la conoscenza necessaria al proprio ruolo.

Il collegamento *percezioni* trasferisce informazioni dalla interfaccia di input dell'agente a quella della BASE DELLA CONOSCENZA; in quest'ultima componente le percezioni correnti sono distinte da quelle già assimilate. Il colle-

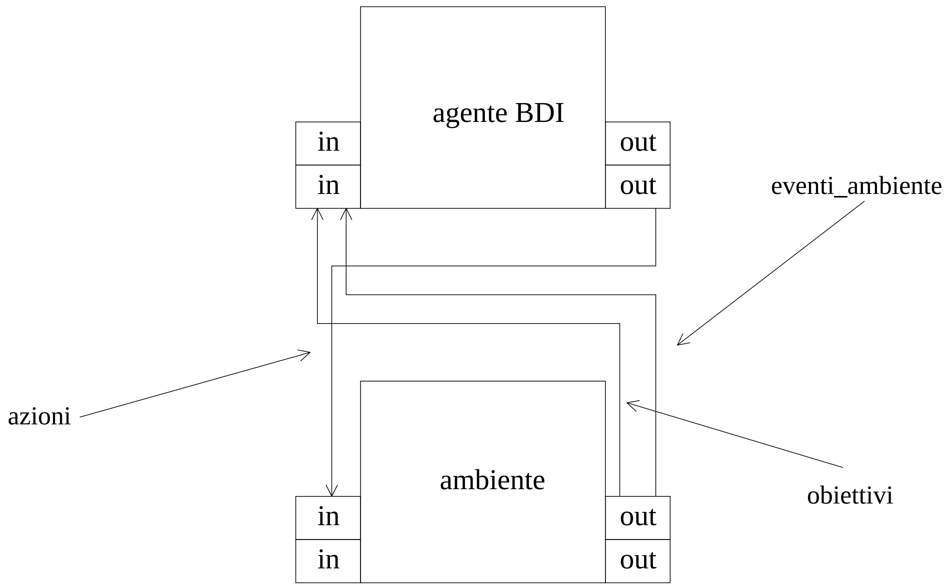
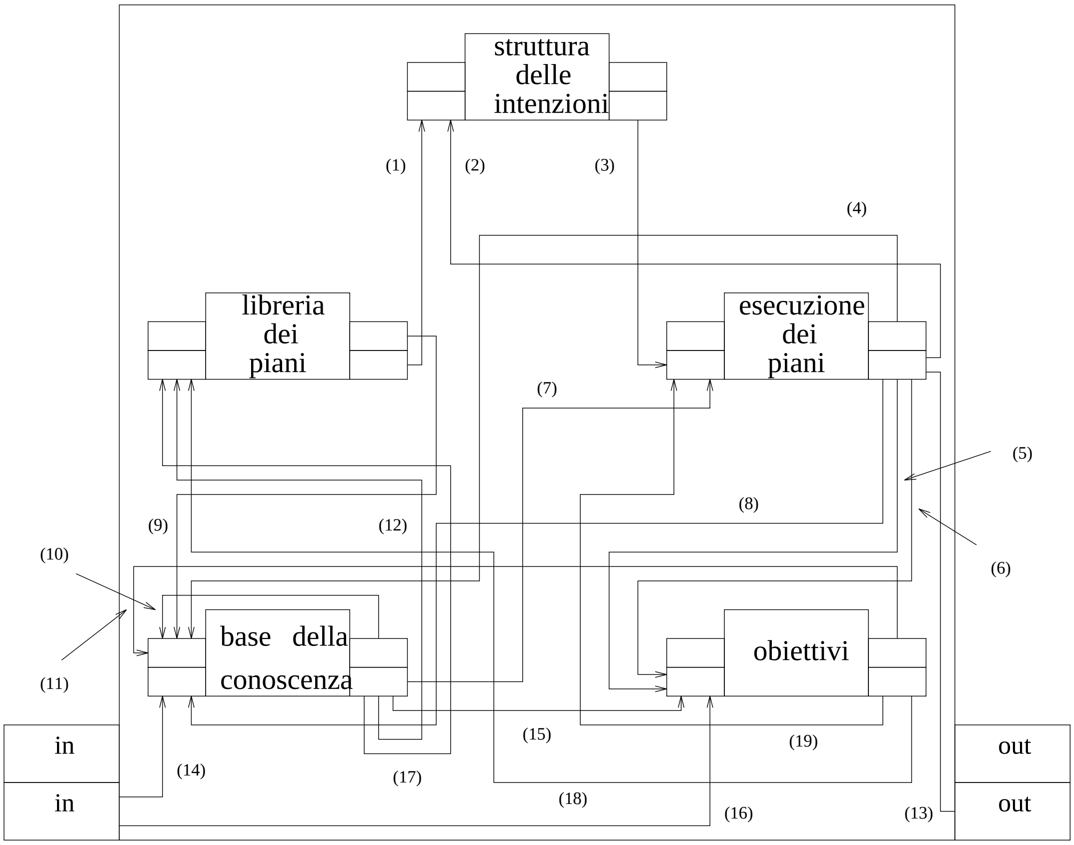


Figura 2.5: L'agente BDI e l'ambiente in cui è collocato.



(1)	piani_opzionali	(7)	credenze_testate	(13)	azioni
(2)	piani_eseguiti	(8)	asserzioni	(14)	percezioni
(3)	attivazione_piani	(9)	focus_trigger_e_contesto	(15)	obiettivi_testati
(4)	focus_tests	(10)	aggiornamento_percezioni	(16)	obiettivi_intrinseci
(5)	sotto_obiettivi	(11)	focus_obiettivi	(17)	trigger_credenze
(6)	obiettivi_raggiunti	(12)	contesto	(18)	trigger_obiettivi
				(19)	obiettivi_gia_raggiunti

Figura 2.6: Agente BDI.

gamento *aggiornamento_percezioni* aggiorna le percezioni presenti nella BASE DELLA CONOSCENZA. Il collegamento *asserzioni* trasferisce le asserzioni ottenute durante l'esecuzione dei piani dalla componente ESECUZIONE DEI PIANI a BASE DELLA CONOSCENZA. I collegamenti *focus_tests*, *focus_obiettivi*, *focus_trigger_e_contesto* sono utilizzati per derivare le credenze rilevanti. Le credenze utilizzate per l'invocazione dei piani sono inviate alla componente LIBRERIA DEI PIANI tramite i collegamenti *trigger_credenze* e *contesto*. Il collegamento *obiettivi_testati* trasferisce le credenze utilizzate per verificare se un obiettivo è stato completato. Le credenze utilizzate per effettuare i test necessari all'esecuzione dei piani sono trasferite tramite *credenze_testate*. Il collegamento *obiettivi_intrinseci* trasferisce gli obiettivi ottenuti dall'ambiente alla componente OBIETTIVI. Il collegamento *sotto_obiettivi* trasferisce i sotto-obiettivi, prodotti durante l'esecuzione dei piani, dalla componente ESECUZIONE DEI PIANI alla componente OBIETTIVI. Gli obiettivi che sono stati completati sono rimossi utilizzando il collegamento *obiettivi_raggiunti*. Il collegamento *trigger_obiettivi* trasferisce gli obiettivi che selezioneranno un piano alla componente LIBRERIA DEI PIANI. I piani invocati vengono trasferiti nella componente ESECUZIONE DEI PIANI dal collegamento *piani_opzionali*. Il collegamento *attivazione_piani* trasferisce le intenzioni scelte per l'esecuzione. Le informazioni riguardanti il successo o il fallimento dei piani sono trasferite tramite il collegamento *piani_eseguiti*. Le azioni scelte per essere eseguite sono trasferite all'interfaccia di output tramite il collegamento *azioni*.
Le componenti BASE DELLA CONOSCENZA, OBIETTIVI e LIBRERIA DEI PIANI sono componenti primitive: non sono descritte attraverso "sotto componenti", mentre le componenti STRUTTURA DELLE INTENZIONI ed ESECUZIONE DEI PIANI sono componenti composte.
Nella BASE DELLA CONOSCENZA le percezioni correnti e le asserzioni diventano credenze correnti; se il valore di verità di un atomo cambia, viene generata una nuova credenza. Un agente crede in tutte le congiunzioni e disgiunzioni delle sue credenze correnti ed assimilate.
Gli obiettivi sono aggiunti nella componente OBIETTIVI attraverso i collegamenti *obiettivi_intrinseci* e *sotto_obiettivi*, sono rimossi attraverso il collegamento *piani_eseguiti*. Quando deve essere eseguito un sotto obiettivo la componente OBIETTIVI verifica, attraverso il collegamento *focus_goal*, se serve oppure no eseguire il sotto obiettivo. Il risultato di questa verifica viene trasferito come credenza corrente tramite il collegamento *obiettivi_testati*. Con riferimento all'esempio del mondo dei blocchi, se un agente ha l'obiettivo di impilare ordinatamente due blocchi, come sotto obiettivo può avere quello di posizionare il primo dei due blocchi. Attraverso l'uso del collegamento *focus_goal* l'agente verifica se il primo blocco si trova già nella posizione corretta.
Obiettivi intrinseci e sotto-obiettivi non ancora completati vengono trasferiti al-

la componente LIBRERIA DEI PIANI tramite il collegamento *trigger_ obiettivi*. Gli obiettivi già completati sono trasferiti nella componente ESECUZIONE DEI PIANI tramite il collegamento *obiettivi_ già_ raggiunti*. In questo modo ESECUZIONE DEI PIANI è a conoscenza dei sotto-obiettivi che hanno terminato con successo la propria esecuzione.
La componente LIBRERIA DEI PIANI determina per quali piani deve essere verificato il contesto. Se tale contesto è derivato dalle credenze correnti della BASE DELLA CONOSCENZA il piano considerato diventa un piano opzionale ed è trasferito alla componente STRUTTURA DELLE INTENZIONI. Le credenze correnti ed il contesto vengono ottenute tramite i collegamenti *trigger_ credenze* e *contesto*. I nuovi sotto obiettivi e gli obiettivi intrinseci sono ottenuti dalla componente OBIETTIVI attraverso il collegamento *trigger_ obiettivi*.
La componente STRUTTURA DELLE INTENZIONI seleziona i piani opzionali che devono essere eseguiti. Basa il proprio funzionamento sia sui piani opzionali che sono stati invocati nella componente LIBRERIA DEI PIANI e trasferiti dal collegamento *piani_ opzionali* sia sui risultati dell'esecuzione dei piani provenienti dalla componente ESECUZIONE DEI PIANI, attraverso il collegamento *piani_ eseguiti*. La componente determina le intenzioni correnti e quelle la cui esecuzione è terminata con successo. Basandosi su queste informazioni la STRUTTURA DELLE INTENZIONI seleziona l'intenzione attiva cioè quella i cui piani sono correntemente eseguiti.
La componente ESECUZIONE DEI PIANI, basandosi sull'intenzione corrente e sullo stato del piano che deve essere eseguito, determina quale istruzione appartenente al piano deve essere eseguita.

2.5.2 Osservazioni

In DESIRE il comportamento dell'agente e la strategia utilizzata nell'architettura per realizzarlo sono aspetti entrambi descritti all'interno della componente composta relativa all'agente, ma specificati in sotto-componenti separate che interagiscono tramite canali di comunicazione. Canali di comunicazione che a loro volta devono essere descritti nella specifica dell'agente. In Concurrent MetateM l'utilizzo della logica temporale fa si che tali aspetti siano "incapsulati" tra di loro: che dipendano cioè dal meccanismo di valutazione delle formule logiche utilizzate. Ad esempio se per perseguire un obiettivo l'agente BDI individua più piani ne sceglie uno e lo esegue. Se tale piano fallisce ne sceglie un altro, prova ad eseguirlo e cosi via. Questo aspetto dinamico dell'agente viene realizzato nei due linguaggi in modi differenti: in Concurrent MetateM come si vede nella sezione 2.4.3 è definita la regola che sceglie un piano e lo esegue, mentre per DESIRE (figura 2.6) tale comportamento è ottenuto attraverso la comunicazione delle sotto-componenti ESECUZIONE DEI PIANI, LIBRERIA DEI PIANI, STRUTTURA DELLE IN-

TENZIONI.

Nel sistema multi-agente descritto attraverso l'uso di DESIRE gli agenti sono rappresentati come sotto-componenti della componente composta che rappresenta l'intero sistema multi-agente. Questa componente interagisce, attraverso canali di comunicazione, con la componente rappresentante l'ambiente. La componente che rappresenta un agente del sistema, è a sua volta una componente composta, in cui ogni sotto-componente ha una propria scala temporale. In Concurrent MetateM gli agenti sono modellati come oggetti non ulteriormente strutturati; ogni oggetto ha una propria scala temporale ed ha un comportamento descritto da un insieme di regole della logica temporale.

La comunicazione tra agenti in DESIRE viene realizzata tramite canali di comunicazione ognuno dei quali viene definito nella specifica dell'agente. Tale comunicazione viene realizzata utilizzando un protocollo "point-to-point": ogni agente sa con quale altro agente comunicare ed invia i messaggi solo a quell'agente. In Concurrent MetateM la comunicazione è realizzata tramite un protocollo "broadcast-message-passing".

In DESIRE il meta-ragionamento viene modellato usando una componente per il livello oggetto ed una per il livello meta. Ad esempio una componente può ragionare sul processo di ragionamento di un'altra componente. In Concurrent MetateM il meta ragionamento viene realizzato tramite l'utilizzo della logica MML (Meta MetateM Logic). In questa logica il dominio sul quale vengono definiti i termini viene esteso in modo da contenere il nome delle formule del livello oggetto.

2.6 ConGolog

ConGolog è un linguaggio di specifica che descrive il comportamento dell'agente basandosi sulle azioni che può eseguire [LLFL+95]. Le azioni vengono espresse attraverso precondizioni e postcondizioni utilizzando il *Situation Calculus*, un formalismo che modella gli aspetti dinamici del "mondo" attraverso il concetto di *situazione* (stato), che può cambiare solo come conseguenza dell'esecuzione di una azione.

2.6.1 *Situation Calculus*

Utilizzando il *Situation Calculus* possono essere formulate teorie che descrivono i cambiamenti del "mondo" come risultato dell'esecuzione di azioni [GLL99].
Nel *Situation Calculus*S_0 è la costante che denota la situazione iniziale, cioè la

situazione in cui non è ancora stata eseguita nessuna azione, La funzione $do(a, s)$ rappresenta la situazione che scaturisce dall'esecuzione dell'azione a nella situazione s, i *fluenti* sono simboli di predicato oppure di funzione che hanno una situazione come ultimo argomento ed il cui valore cambia da situazione a situazione ed infine $poss(a, s)$ è il predicato che asserisce che l'azione a è eseguibile nello situazione s (non è un fluente in quanto il suo valore di verità non cambia durante l'esecuzione del sistema).
Le teorie espresse nel *Situation Calculus*hanno la forma seguente:

- assiomi che descrivono la situazione iniziale S_0;
- assiomi che descrivono le precondizioni delle azioni. Viene definito un assioma per ogni azione primitiva a in modo che caratterizzi $poss(a, s)$;
- assiomi che esprimono le post-condizioni delle azioni, che asseriscono sotto quali condizioni il fluenti della forma $F(\vec{x}, do(a, s))$ valgono in funzione della situazione s;
- assiomi sui nomi delle azioni primitive.

Una pianificazione produce una sequenza di azioni la cui esecuzione genera uno stato (situazione) che soddisfa la formula che rappresenta l'obiettivo della pianificazione. Un'azione è eseguibile in un contesto se tale contesto soddisfa le precondizioni dell'azione.

PIANIFICAZIONE:

sia *axioms* una teoria assiomatica della forma appena descritta, $\phi(s)$ una formula rappresentante un obiettivo con la variabile libera s che può essere istanziata con una situazione. Il compito della pianificazione è cercare una sequenza di azioni $\vec{a}$ tale che:

$$axioms \models legal(\vec{a}, S_0) \ \wedge \ \phi(do(\vec{a}, S_0))$$

dove $do([a_1, \ldots , a_n], s)$ è una abbreviazione per

$$do(a_n, do(a_{n-1}, \ldots , do(a_1, s) \ldots))$$

e dove $legal([a_1, \ldots , a_n], s)$ è una abbreviazione per

$$poss(a_1, s) \ \wedge \ \ldots \ \wedge \ poss(a_n, do([a_1, \ldots , a_{n-1}], s)).$$

2.6.2 GOLOG

GOLOG è un linguaggio di programmazione che include i costrutti seguenti:

- a: azione primitiva. Si tratta di una azione del situation calcolus;
- $\phi?$: verifica ϕ nello stato corrente;
- $(\delta_1 \; ; \; \delta_2)$: concatenazione di istruzioni;
- $(\delta_1 \mid \delta_2)$: scelta non deterministica di istruzioni;
- $\pi v.\delta$: scelta non deterministica di argomenti;
- δ^*: iterazione non deterministica;
- $\{\mathbf{proc}\ P_1(\vec{v}_1)\delta_1\ \mathbf{end};\ \ldots\ \mathbf{proc}\ P_n(\vec{v}_n)\delta_n\ \mathbf{end};\ \delta\}$: procedure.

Nelle formule del *Situation Calculus* utilizzate in GOLOG tutti i parametri riguardanti la situazione sono rimpiazzati con la costante *now*.

Considerando ad esempio uno scenario in cui un robot può compiere azioni che modificano la posizione di un insieme di blocchi su di un piano, il seguente programma GOLOG libera il piano gettando via i blocchi:

$\{$ **proc** *removeAblock*

$\pi b.[OnTable(b, now)?;\ pickUp(b);\ putAway(b)]$

end;
removeAblock$*$;
$\neg\exists\ b\ OnTable(b, now)?\ \}$

Il programma inizia con la definizione di una procedura che rimuove un blocco da sopra il piano. Tale procedura utilizza l'operatore π per individuare il blocco oggetto dello spostamento: l'istruzione $\pi x.\delta(x)$ sceglie non deterministicamente un elemento x e per tale x esegue il programma $\delta(x)$. Il test $OnTable(b, now)?$ ha successo solo se l'elemento b è un blocco giacente sul piano. Il "main" del programma utilizza l'operatore di iterazione non deterministica $*$ per invocare la procedura *removeAblock*. Il programma termina se il test $\neg\exists\ b\ OnTable(b, now)?$ ha successo e questo avviene se l'invocazione della procedura *removeAblock* viene effettuato dall'operatore $*$ il "numero giusto" di volte, come verrà spiegato in seguito nella semantica dei programmi GOLOG.

L'operatore di scelta non deterministica ($\delta_1 \mid \delta_2$) permette la scelta tra due zioni: può essere eseguita δ_1 oppure δ_2.

L'esecuzione di un programma GOLOG è un caso particolare della PIANIFICAZIONE del *Situation Calculus* definita nella sezione 2.6.1:

ESECUZIONE DI UN PROGRAMMA:

> data una teoria $\mathcal{D}$, un programma δ ed una situazione s, l'esecuzione di un programma consiste nella ricerca di una sequenza di azioni $\vec{a}$ tale che:
>
> $$\mathcal{D} \models Do(\delta,\ s,\ do(\vec{a},\ s))$$

dove $Do(\delta, s, s^{'})$ significa che il programma δ quando eseguito nella situazione s ha $s^{'}$ come situazione finale.
Generalmente un programma GOLOG è non deterministico; l'esecuzione di un programma in una situazione può risultare in situazioni finali diverse per diverse esecuzioni.
$Do(\delta, s, s^{'})$ può essere vista come una abbreviazione per una formula del *Situation Calculus* definita induttivamente come:

1. azione primitiva:

$$Do(a, s, s') \ =_{def}\ poss(a[s], s)\ \wedge\ s' = do(a[s], s)$$

2. test:

$$Do(\phi?, s, s') \ =_{def}\ \phi[s]\ \wedge\ s = s'$$

3. sequenza:

$$Do(\delta_1; \delta_2, s, s') \ =_{def}\ \exists s''\ Do(\delta_1, s, s'')\ \wedge\ Do(\delta_2, s'', s')$$

4. scelta non deterministica di azioni:

$$Do(\delta_1 | \delta_2, s, s') \ =_{def}\ Do(\delta_1, s, s')\ \vee\ Do(\delta_2, s, s')$$

5. scelta non deterministica di argomenti:

$$Do(\pi x.\delta(x), s, s') \ =_{def}\ \exists x\ Do(\delta(x), s, s')$$

6. iterazione non deterministica:

$$Do(\delta *, s, s') \ =_{def}$$

$\forall p\{\forall s_1\, p(s_1, s_1) \wedge \forall s_1,\ s_2,\ s_3,\ [p(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset p(s_1, s_3)]\} \supset\ p(s, s')$

L'esecuzione dell'azione δ iterata zero o più volte, porta dalla situazione s alla situazione s' sse la coppia (s, s') è in tutti gli insiemi (e quindi nel più piccolo insieme) tali che:

- per qualsiasi s_1 la coppia (s_1, s_1) è nell'insieme;
- se (s_1, s_2) è nell'insieme e l'esecuzione di δ nella situazione s_2 conduce nella situazione s_3 allora la coppia (s_1, s_3) è nell'insieme.

L'iterazione non deterministica è basata sul concetto di chiusura transitiva: viene descritta attraverso la logica del secondo ordine in quanto non può essere espressa con la logica del primo ordine.

L'espressione $a[s]$ ($\phi[s]$) denota l'azione (la formula) del *Situation Calculus* ottenuta dal termine GOLOG a (ϕ) in cui sono state sostituite le occorrenze della costante *now* con la situazione s.
La descrizione del comportamento di Do quando applicato alle procedure, viene tralasciata in [GLL99] in quanto comporta una trattazione troppo complessa.

2.6.3 Definizione del linguaggio ConGolog

ConGolog (Concurrent GOLOG) è un'estensione del linguaggio GOLOG in cui vengono aggiunti costrutti che arricchiscono il trattamento della concorrenza:

- trattazione di processi concorrenti con diverse priorità;
- trattazione delle interruzioni;
- azioni esterne arbitrarie.

La concorrenza in ConGolog viene modellata come interleaving delle azioni primitive dei processi eseguiti concorrentemente.
In GOLOG se un processo δ deve eseguire l'azione primitiva a nella situazione s, ma il predicato $poss(a, s)$ è falso, oppure il test $\phi?$ contenuto nel programma δ fallisce, allora δ fallisce. Questo non succede in ConGolog: il processo viene sospeso in attesa che qualche altro processo eseguito concorrentemente renda possibile continuare l'esecuzione di δ.
La struttura del linguaggio ConGolog è esattamente come quella di GOLOG con l'aggiunta dei seguenti costrutti:

- **if** ϕ **then** δ_1 **else** δ_2: if-then-else sincronizzato;

- **while** ϕ **do** δ: ciclo condizionale;
- $(\delta_1 \| \delta_2)$: esecuzione concorrente;
- $(\delta_1 \gg \delta_2)$: esecuzione concorrente con diverse priorità;
- $\delta^{\|}$: iterazione concorrente;
- $\langle \phi \;\rightarrow\; \delta \rangle$: interrupt.

Nei costrutti if-then-else sincronizzato e ciclo condizionale la valutazione del test ϕ e l'esecuzione della prima azione del ramo scelto sono un'operazione atomica: tale valutazione si comporta come l'istruzione atomica test-and-set usata per la costruzione dei semafori nella programmazione concorrente.
Il costrutto $(\delta_1 \| \delta_2)$ denota l'esecuzione concorrente delle istruzioni δ_1 e δ_2, mentre $(\delta_1 \gg \delta_2)$ denota l'esecuzione concorrente delle istruzioni δ_1 e δ_2 in cui δ_1 ha maggiore priorità di δ_2: δ_2 viene eseguita solo quando δ_1 è bloccata oppure ha terminato la propria esecuzione.
L'iterazione concorrente $\delta^{\|}$ si comporta come l'iterazione non deterministica di GOLOG, con la differenza che le istanze di δ vengono eseguite concorrentemente piuttosto che in sequenza.
In GOLOG $\delta*$ assume il significato:

$$nil \mid \delta \mid \delta;\delta \mid \delta;\delta;\delta \mid \ldots$$

mentre in ConGolog $\delta^{\|}$ assume il significato:

$$nil \;\|\; \delta \;\|\; \delta;\delta \;\|\; \delta;\delta;\delta \;\|\; \ldots$$

dove nil rappresenta il programma vuoto.
Nel costrutto $\langle \phi \;\rightarrow\; \delta \rangle$ la condizione ϕ ha il ruolo di condizione trigger, mentre δ è il corpo dell'interrupt. Se ϕ diventa vera durante l'esecuzione di un processo δ viene eseguita e l'interrupt assume il completo controllo sull'esecuzione. Quando δ ha terminato la propria esecuzione il controllo ritorna al processo che ha reso vera ϕ e l'interrupt può essere selezionato di nuovo.
Con l'utilizzo degli interrupt in ConGolog possono essere facilmente specificati programmi che hanno comportamento reattivo.

2.6.4 Semantica transazionale di ConGolog

Sia $Trans(\delta,\ s,\ \delta',\ s')$ la relazione che associa al programma δ e alla situazione s la situazione s' ed il programma δ', dove s' è la situazione che risulta dall'esecuzione di una azione primitiva oppure di un test di δ, e δ' è δ senza l'azione o il test eseguiti.

Una configurazione è una coppia della forma (δ, s) dove δ è un programma e s una situazione. $Trans$ denota una relazione di transizione tra configurazioni.
Il predicato $Final(\delta, s)$ è vero se la configurazione (δ, s) è una configurazione finale.
La configurazione finale (δ_n, s_n), raggiunta in un numero finito di transizioni a partire dalla configurazione iniziale (δ, s), coincide con la terminazione del programma δ e soddisfa la relazione Do definita nella sezione 2.6.2. Se un programma non termina Do non viene soddisfatta, quindi risulta essere indefinita. $Trans$ al contrario di Do, caratterizza il significato dei programmi non terminanti tenendo conto delle configurazioni attraversate durante l'esecuzione.

Il significato del predicato $Trans$ è caratterizzato dal seguente insieme di assiomi $\mathcal{T}$:

1. programma vuoto:

$$Trans(nil, s, \delta', s') \equiv False$$

2. azione primitiva:

$$Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s)$$

3. test:

$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$$

4. sequenza di azioni:

$$\begin{aligned}&Trans(\delta_1; \delta_2, s, \delta', s') \equiv \\ &\quad (\exists \gamma. \delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s')) \vee \\ &\quad (Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s'))\end{aligned}$$

5. scelta non deterministica di azioni:

$$Trans(\delta_1 | \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$

6. scelta non deterministica di un argomento:

$$Trans(\pi v.\delta, s, \delta', s') \equiv \exists x. Trans(\delta^v_x, s, \delta', s')$$

dove δ^v_x è il programma che risulta da δ sostituendo v con la variabile x

7. iterazione non deterministica:

$$Trans(\delta*, s, \delta', s') \equiv \exists \gamma.(\delta' = \gamma; \delta*) \wedge Trans(\delta, s, \gamma, s')$$

8. **if-then-else** sincronizzato:

$$Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta', s') \equiv$$
$$\phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s')$$

9. **while** sincronizzato:

$$Trans(\textbf{while } \phi \textbf{ do } \delta, s, \delta', s') \equiv$$
$$\exists\gamma.(\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s')$$

10. esecuzione concorrente:

$$Trans(\delta_1 \| \delta_2, s, \delta', s') \equiv$$
$$(\exists\gamma.\delta' = (\gamma \| \delta_2) \wedge Trans(\delta_1, s, \gamma, s')) \vee$$
$$(\exists\gamma.\delta' = (\delta_1 \| \gamma) \wedge Trans(\delta_2, s, \gamma, s'))$$

11. esecuzione concorrente con priorità:

$$Trans(\delta_1 \gg \delta_2, s, \delta', s') \equiv$$
$$\exists\gamma.\delta' = (\gamma \gg \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee$$
$$\exists\gamma.\delta' = (\delta_1 \| \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''.Trans(\delta_1, s, \zeta, s'')$$

12. iterazione concorrente:

$$Trans(\delta^{\|}, s, \delta', s') \equiv \exists\gamma.\delta' = (\gamma \| \delta^{\|}) \wedge Trans(\delta, s, \gamma, s')$$

In $Trans(a, s, \delta', s')$ (rispettivamente $Trans(\phi?, s, \delta', s')$) l'argomento a (rispettivamente ϕ) è un termine del primo ordine che codifica la corrispondente azione (formula) del *Situation Calculus* in ConGolog col fine di poterla utilizzare come argomento dei predicati *poss* e *do*. La funzione $\cdot[\cdot]$ associa il termine a (ϕ) alla corrispondente azione (formula) del *Situation Calculus* $a[s]$ ($\phi[s]$) rimpiazzando la costante *now* con la situazione s.

Gli assiomi caratterizzano come una configurazione (δ, s) evolve, in un singolo passo, nella configurazione successiva:

1. (nil, s) non può evolvere in nessuna configurazione.

2. (a, s) evolve in $(nil, do(a[s], s)$ se è possibile eseguire l'azione $a[s]$ nella situazione s.

3. $(\phi?, s)$ evolve in (nil, s) se $\phi[s]$ vale nella situazione s. L'esecuzione di $\phi?$ non modifica la situazione in cui ϕ è testata.

4. $(\delta_1;\delta_2,s)$ evolve nella configurazione $(\delta_1';\delta_2,s')$ se (δ_1,s) evolve nella configurazione (δ_1',s'); evolve nella configurazione (δ_2',s') se (δ_1,s) è una configurazione finale e (δ_2,s) può evolvere in (δ_2',s').

5. $(\delta_1|\delta_2,s)$ evolve in (δ',s') se (δ_1,s) oppure (δ_2,s) possono evolvere in (δ',s').

6. $(\pi v.\delta,s)$ evolve in (δ',s') se esiste x tale che (δ_x^v,s) può evolvere in (δ',s'). In realtà v viene sostituito con un termine della forma *nameof(x)*, dove *nameof* converte oggetti del *Situation Calculus* in termini del primo ordine del tipo corrispondente per essere utilizzati come argomenti all'interno dei predicati.

7. $(\delta*,s)$ evolve in $(\delta';\delta*,s')$ se (δ,s) può evolvere in (δ',s').

8. (**if** ϕ **then** δ_1 **else** δ_2,s) evolve in (δ',s') se $\phi[s]$ vale e (δ_1,s) può evolvere in (δ',s'), oppure $\neg\phi[s]$ vale e (δ_2,s) può evolvere in (δ',s').

9. (**while** ϕ **do** δ,s) evolve in (δ';**while** ϕ **do** δ,s') se $\phi[s]$ vale e (δ,s) evolve in (δ',s').

10. $(\delta_1\|\delta_2,s)$ evolve in $(\delta_1'\|\delta_2,s')$ se (δ_1,s) evolve in (δ_1',s'); similmente, $(\delta_1\|\delta_2,s)$ evolve in $(\delta_1\|\delta_2',s')$ se (δ_2,s) evolve in (δ_2',s').

11. Il costrutto $\delta_1 \gg \delta_2$ è identico al costrutto $\delta_1\|\delta_2$, eccetto per il fatto che può essere eseguito un passo di δ_2 solo se non è possibile eseguire nessun passo di δ_1: $(\delta_1 \gg \delta_2,s)$ evolve in $(\delta_1'\|\delta_2,s')$ se (δ_1,s) evolve in (δ_1',s'). $(\delta_1\|\delta_2,s)$ evolve in $(\delta_1\|\delta_2',s')$ solo se (δ_1,s) è bloccato e (δ_2,s) può evolvere in (δ_2',s').

12. L'esecuzione di un passo di $\delta^{||}$ è ottenuta dall'esecuzione di un passo di δ in parallelo con $\delta^{||}$ stessa. Quello che si ottiene è un numero illimitato di istanze di δ eseguite concorrentemente: $(\delta^{||},s)$ evolve in $(\delta'\|\delta^{||},s')$ se (δ,s) evolve in (δ',s').

$Final(\delta,s)$ definisce quando il programma δ, nella situazione s, è in una configurazione finale.
Il significato del predicato *Final* per programmi senza procedure è caratterizzato dall'insieme di assiomi $\mathcal{F}$:

1. programma vuoto:

$$Final(nil,s) \;\equiv\; true$$

2. azione primitiva:

$$Final(a,s) \;\equiv\; false$$

3. test:

$$Final(\phi?, s) \equiv false$$

4. sequenza di azioni:

$$Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

5. scelta non deterministica di azioni:

$$Final(\delta_1 | \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

6. scelta non deterministica dell' argomento:

$$Final(\pi v.\delta, s) \equiv \exists x.Final(\delta^v_x, s)$$

7. iterazione non deterministica:

$$Final(\delta *, s) \equiv true$$

8. **if-then-else** sincronizzato:

$$\begin{aligned}&Final(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s) \equiv \\ &\quad \phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] \wedge Final(\delta_2, s)\end{aligned}$$

9. while sincronizzato:

$$Final(\textbf{while } \phi \textbf{ do } \delta, s) \equiv \neg\phi[s] \wedge Final(\delta, s)$$

10. esecuzione concorrente:

$$Final(\delta_1 \| \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

11. esecuzione concorrente priorizzata:

$$Final(\delta_1 \gg \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

12. iterazione concorrente:

$$Final(\delta^{||}, s) \equiv true$$

Il cui significato informale è :

1. (nil, s) è una configurazione finale.
2. (a, s) non è una configurazione finale in quanto il programma consistente nell'azione a non può considerarsi terminato fino a quando l'azione non viene eseguita.

3. $(\phi?, s)$ non è una configurazione finale in quanto il programma consistente nel test $\phi?$ non può considerarsi terminato fino a quando il test non viene eseguito.

4. $(\delta_1; \delta_2, s)$ può essere considerata finale se (δ_1, s) e (δ_2, s) lo sono.

5. $(\delta_1|\delta_2, s)$ può essere considerata finale se (δ_1, s) oppure (δ_2, s) lo sono.

6. $(\pi v.\delta, s)$ può essere considerata finale se esiste x tale che (δ_x^v, s) è una configurazione finale, dove δ_x^v è ottenuta dalla sostituzione di tutte le occorenze di v con x.

7. $(\delta*, s)$ è una configurazione finale poichè $\delta*$ può essere eseguita zero volte.

8. $(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s)$ è una configurazione finale se $\phi[s]$ vale e δ_1 è una configurazione finale oppure se $\neg\phi[s]$ vale e δ_2 è una configurazione finale.

9. $(\textbf{while } \phi \textbf{do } \delta, s)$ è una configurazione finale se $\phi[s]$ vale e δ è una configurazione finale.

10. $(\delta_1\|\delta_2, s)$ può essere considerata finale se (δ_1, s) e (δ_2, s) lo sono.

11. $(\delta_1 \gg \delta_2, s)$ può essere considerata finale se (δ_1, s) e (δ_2, s) lo sono.

12. $(\delta^{||}, s)$ è una configurazione finale poichè $\delta^{||}$ può essere eseguita zero o più volte.

Le possibili configurazioni raggiunte da un programma δ nella situazione s, sono quelle ottenute dalla chiusura transitiva riflessiva della relazione $Trans$ denotata con $Trans^*$ definita come formula del secondo ordine del *Situation Calculus*:

$$Trans^*(\delta, s, \delta', s') \ =_{def} \ \forall T[\ldots \ \supset \ T(\delta, s, \delta', s')]$$

dove ... è :

$$\begin{aligned}&\forall \delta, s, \ \delta', s', \ \delta'', s''(true \ \supset \ T(\delta, s, \delta, s) \wedge \\ &Trans(\delta, s, \delta'', s'') \ \wedge \ Trans(\delta'', s'', \delta', s') \ \supset \ T(\delta, s, \delta', s'))\end{aligned}$$

Usando $Trans^*$ e $Final$ la relazione Do può essere riscritta come:

$$Do(\delta, s, s') \ =_{def} \ \exists \delta'.Trans^*(\delta, s, \delta', s') \ \wedge \ Final(\delta', s')$$

In altre parole Do vale se è possibile ottenere una configurazione finale eseguendo a passi il programma δ a partire da una data situazione.

I predicati $Trans$ e $Final$ non sono definiti per gli interrupts in quanto questi ultimi sono realizzati attraverso l'uso di altri costrutti ConGolog:

$$\langle \phi \ \rightarrow \ \delta \rangle \ =_{def} \textbf{ while } Interrupts\_running \textbf{ do}$$

$$\textbf{if } \phi \textbf{ then } \delta \textbf{ else } false$$

Il fluente denotato da *Interrupts_running* è vero per default. Quando un interrupt prende il controllo esegue ripetutamente δ fino a che ϕ non diventa falso; a quel punto l'interrupt blocca la propria esecuzione ed il controllo ritorna al programma. Per terminare il ciclo in cui l'interrupt esegue ripetutamente δ viene utilizzata l'azione primitiva *stop_Interrupts* il cui effetto è quello di rendere *Interrupts_running* falso.
Sia, ad esempio, δ un programma che contiene l'interrupt e sia *start_Interrupts* una azione il cui effetto è quello di attivarlo, allora il programma:

$$\{start\_Interrupts; (\delta \gg stop\_interrupts)\}$$

ha l'effetto di terminare tutti i processi generati da δ, bloccati da qualche interrupt.

Le azioni esterne sono azioni primitive che non fanno parte delle conoscenze procedurali di alcun agente e possono essere eseguite asincronamente rispetto all'esecuzione degli agenti. Vengono dichiarate utilizzando il predicato *Exo*: se a è una azione exogena allora il predicato $Exo(a)$ risulta essere vero.
Ad esempio l'esecuzione delle azioni esterne è modellata attraverso il programma δ_{Exo}:

$$\delta_{Exo} \ =_{def} \ (\pi a.Exo(a)?; a)*$$

che viene eseguito concorrentemente ad un programma ConGolog arbitrario δ:

$$\delta || \delta_{Exo}$$

L'esecuzione in parallelo di δ_{Exo} con δ permette il manifestarsi di eventi al di fuori del controllo di δ generati da δ_{Exo}.

Per poter trattare le espressioni di ConGolog come argomenti dei predicati $Trans$ e $Final$, bisogna codificare tali espressioni come termini del primo ordine. Per fare questo bisogna introdurre simboli di costante, simboli di funzioni, tipi e assiomi adeguati. Oltre le sorts *Sit*, *Obj*, *Act* utilizzate per la definizione di *Situation Calculus*, in [GLL99] vengono aggiunte anche le sorts *Idx*, *PseudoSit*, *PseudoAct*, *PseudoObj*, *PseudoForm*, *ENV* e *PROG*. Intuitivamente gli elementi di *Idx* denotano numeri naturali utilizzati per definire indici. Gli elementi di *PseudoAct*, *PseudoObj*, *PseudoSit*, *PseudoForm* vengono utilizzati rispettivamente per definire la codifica di azioni, oggetti, situazioni e formule all'interno dei programmi. Gli elementi di *ENV* denotano gli ambienti utilizzati nelle procedure e gli elementi di *PROG* denotano la codifica dei programmi.

La semantica transazionale di ConGolog viene estesa alle procedure utilizzando la definizione del secondo ordine della relazione $Trans$.
Sia:

$$\textbf{proc } P_1(\vec{v}_1)\delta_1 \textbf{ end};\ \ldots;\textbf{proc } P_n(\vec{v}_n)\delta_n \textbf{ end}$$

una collezione di definizioni di procedure chiamata *ambiente* e denotata da *Env*. Nella definizione di procedura **proc** $P_i(\vec{v}_i)\delta_i$ **end** appartenente ad *Env*, P_i è il nome della i-esima procedura, $\vec{v}_i$ sono i parametri formali e δ_i è il corpo, che a sua volta può avere la forma di un programma ConGolog che include definizioni e chiamate di procedura. Nelle procedure il passaggio dei parametri è per valore, mentre lo scope delle variabili è statico.
Vengono introdotti tre costrutti:

- $P(\vec{t})$ è una chiamata di procedura dove P è un nome di procedura e $\vec{t}$ i parametri attuali associati alla procedura. In $\vec{t}$ gli argomenti che denotano situazioni sono sostituiti con la costante now.
- $\{Env; \delta\}$, dove Env è un ambiente e δ un programma, associa le chiamate di procedura in δ alle relative definizioni in Env.
  Ad esempio in $\{\textbf{proc } P_1()a \textbf{ end}; P_2(); P_1()\}$ $P_1()$ è legata ma $P_2()$ è libera.
- $[Env : P(\vec{t})]$, dove Env è un ambiente, P il nome di una procedura, $\vec{t}$ i parametri attuali associati alla procedura P, denota una chiamata di procedura già contestualizzata: l'ambiente in cui viene cercata la definizione di P è Env.

La semantica dei programmi ConGolog con procedure è data definendo $Trans$ e $Final$ come formule del secondo ordine:

$$Trans(\delta, s, \delta', s') \ \equiv\ \forall T[\ldots\ \supset\ T(\delta, s, \delta', s')]$$

In cui ... significa la congiunzione degli assiomi $\mathcal{T}$ definiti nella sezione 2.6.4 e $Trans$ viene sostituita da T, con le due asserzioni seguenti:

$$T(\{Env; \delta\}, s, \delta', s') \ \equiv\ T(\delta^{p_i(\vec{t})}_{[Env:p_i(\vec{t})]}, s, \delta', s')$$
$$T([Env : P(\vec{t})], s, \delta', s') \ \equiv\ T(\{Env; \delta^{\vec{v}_p}_{p_{\vec{t}[s]}}\}, s, \delta', s')$$

dove $\delta^{p_i(\vec{t})}_{[Env:p_i(\vec{t})]}$ denota il programma con tutte le procedure legate da Env e libere in δ rimpiazzate dalla loro versione contestualizzata. $\delta^{\vec{v}_p}_{p_{\vec{t}[s]}}$ denota il corpo della

procedura P in Env con i parametri formali $\vec{v}$ sostituiti con i parametri attuali $\vec{t}$ valutati nella situazione corrente.
In modo simile viene definito anche $Final$:

$$Final(\delta, s) \equiv \forall T[\ldots \supset F(\delta, s)]$$

In cui $\ldots$ significa la congiunzione degli assiomi $\mathcal{F}$ in cui $Final$ viene sostituita da F, con le due asserzioni seguenti:

$$\begin{aligned} F(\{Env; \delta\}, s) &\equiv F(\delta^{p_i(\vec{t})}_{[Env:p_i(\vec{t})]}, s) \\ F([Env : p(\vec{t})], s) &\equiv F(\{Env; \delta^{\vec{v}_p}_{p_{\vec{t}[s]}}\}, s) \end{aligned}$$

Le chiamate di procedura non legate a definizioni di procedura, non possono compiere transizioni e non sono considerate terminate con successo.
Quando viene eseguita una procedura i parametri attuali sono valutati nella situazione corrente, quindi vengono sostituiti a quelli formali nel corpo della procedura, creando così il passaggio per valore dei parametri.

Il programma in figura 2.7 illustra lo scoping statico di ConGolog ed ha come effetto l'esecuzione dell'azione primitiva a seguita dall'esecuzione dell'azione primitiva b, supponendo che tali azioni siano possibili.
Sia:

$Env_1 =_{def}$

proc $P_1()$ a **end**;
proc $P_2() P_1()$ **end**;
proc $P_3()$ $\{Env_2; P_2(); P_1()\}$ **end**;

$Env_2 =_{def}$

proc $P_1()$ b **end**;

è facile vedere che:

$$\begin{aligned} & Trans(\delta_{stsc}, s, \delta', s') \\ & \quad \equiv Trans(\{Env_1; P_3()\}, s, \delta', s') \\ & \quad \equiv Trans([Env_1 : P_3()], s, \delta', s') \\ & \quad \equiv Trans(\{Env_1; \{Env_2; P_2(); P_1()\}\}, s, \delta', s') \\ & \quad \equiv Trans(\{Env_2; [Env_1 : P_2()]; P_1()\}\}, s, \delta', s') \\ & \quad \equiv Trans([Env_1 : P_2()]; [Env_2 : P_1()], s, \delta', s') \\ & \quad \equiv Trans(\{Env_1; P_1()\}; [Env_2 : P_1()], s, \delta', s') \end{aligned}$$

```
{
    proc p1()
        a
    end;
    proc p2()
        p1()
    end;
    proc p3()
        { proc p1()
            b
            end;
            p2(); p1()
        }
    end;
    p3()
}
```

Figura 2.7:

$$\begin{aligned}
&\equiv\ Trans([Env_1 : P_1()];[Env_2 : P_1()], s, \delta', s')\\
&\equiv\ Trans(a;[Env_2 : P_1()], s, \delta', s')\\
&\equiv\ poss(a, s)\ \wedge\ s' = do(a, s)\ \wedge\ \delta' = (nil;[Env_2 : P_1()])
\end{aligned}$$

Similmente si possono dimostrare:

$$Trans([Env_2 : P_1()], do(a, s), nil, do(b, do(a, s)))$$

e

$$Final(nil, do(b, do(a, s))).$$

Le definizioni di $Trans$ e $Final$ come formule del secondo ordine servono per definire la semantica dei programmi ConGolog con procedure ricorsive, incluso quelle la cui ricorsione porta alla non terminazione, infatti:

$$\{\ \mathbf{proc}\ p()p()\ \mathbf{end};\ p()\ \}$$

non ha transizioni e non soddisfa $Final$ per nessuna situazione.

2.6.5 Un sistema multi-agente per organizzare meetings

In [LLFL+95] viene presentato un semplice esempio di sistema multi-agente per organizzare meetings: ogni utente ha un agente "schedule manager" che è a conoscenza degli impegni dell'utente. Quando un utente vuole organizzare un meeting crea un nuovo agente "meeting scheduler" che contatti lo "schedule manager" dei partecipanti, il quale, se è possibile, aggiunge l'impegno tra gli appuntamenti dell'utente.

Le nozioni descritte nella sezione 2.6.1 vengono estese in modo da contenere le informazioni riguardanti gli agenti del sistema.
L'espressione $do(a, s)$ diventa:

$$do(agt, a, s)$$

e l'espressione $Poss(a, s)$ diventa:

$$poss(agt, a, s)$$

L'espressione $do(agt, a, s)$ rappresenta lo stato ottenuto dall'esecuzione dell'azione a da parte dell'agente agt nella situazione s, $Poss(agt, a, s)$ è vero se l'agente agt può eseguire l'azione a nella situazione s.
Viene inoltre aggiunta l'espressione:

$$\mathbf{by}(agt, \delta)$$

dove δ è una espressione di ConGolog arbitrariamente complessa. L'espressione $by(agt, \delta)$ indica δ in cui ogni occorrenza di una azione primitiva act viene sostituita con $by(agt, act)$, ad esempio:

$$\begin{aligned}&\mathbf{by}(agt, \mathbf{while}\ \phi\ \mathbf{do}\ act_1; act_2) =_{def}\\ &\quad \mathbf{while}\ \phi\ \mathbf{do}\ \mathbf{by}(agt, act_1);\mathbf{by}(agt, act_2)\end{aligned}$$

Una azione è specificata definendo le condizioni sotto le quali può essere eseguita: assiomi per le precondizioni.
Nel caso dell'esempio l'aggiunta di un impegno viene effettuata tramite l'azione AddToSched il cui assioma per le precondizioni è :

$$\begin{aligned}&Poss(agt, \text{AddToSched}(user, period, activity, organizer), s) \equiv\\ &agt = \text{ScheduleManager}(user) \wedge\\ &\neg\exists activity', organizer'\ \text{Schedule}(user, period, activity', organizer', s)\end{aligned}$$

L'assioma per l'azione AddToSched specifica che è possibile, per un agente, prendere un impegno per un utente se e solo se l'agente è lo "schedule manager" e l'utente non ha altri impegni per quel periodo.
La rimozione di un impegno viene effettuata tramite l'azione RmvFromSched:

$$\begin{aligned}&Poss(agt, \text{RmvFromSched}(user, period), s) \equiv\\ &agt = \text{ScheduleManager}(user) \wedge\\ &\exists activity', organizer'\ \text{Schedule}(user, period, activity', organizer', s)\end{aligned}$$

Per un agente è possibile rimuovere una attività che impegna l'utente se e solo se l'agente che esegue l'azione RmvFromSched è lo "schedule manager" e l'utente *user* ha un impegno per il periodo *period*.
Gli assiomi sulle postcondizioni delle azioni definiscono in che modo le azioni cambiano lo stato dell'ambiente. Tali assiomi devono specificare quando un fluente cambia valore di verità e quando tale valore rimane invariato, ad esempio l'assioma:

$$\begin{aligned}&Poss(agt, act, s) \supset\\ &[\text{Schedule}(user, period, activ, org, do(agt, act, s)) \equiv\\ &act = \text{AddToSched}(user, period, activ, org) \vee\\ &\text{Schedule}(user, period, activ, org, s) \wedge\\ &act \neq \text{RmvFromSched}(user, period)].\end{aligned}$$

asserisce che un'attività impegna l'utente *user* se è stata aggiunta dall'agente *agt* tramite l'azione AddToSched oppure se era già presente tra gli impegni dell'utente nella situazione s e non è stata rimossa tramite l'azione RmvFromSched.

La comunicazione tra gli agenti è realizzata definendo un insieme di azioni primitive specializzate nel ricevere e spedire messaggi.
Ogni agente ha una coda di messaggi il cui contenuto nella situazione s è denotato dal fluente $MsgQ(agt, s)$.
Le azioni che operano sulla coda di messaggi sono:

- $SendMsg(recipient, msg)$: manda il messaggio msg al destinatario $recipient$. Nella coda dell'agente destinatario il messaggio ricevuto viene associato al mittente tramite la coppia $(senderId, msg)$;
- $SenseMsg$: legge il messaggio nella testa della coda dei messaggi;
- $RmvMsg$: rimuove il messaggio contenuto nella testa della coda dei messaggi.

Come per le altre azioni vengono definiti gli assiomi per le precondizioni:

$$Poss(agt, RmvMsg, s) \equiv \neg Empty(MsgQ(agt, s))$$

Un agente può rimuovere un messaggio dalla propria coda utilizzando $RmvMsg$ se e solo se la coda non è vuota. Le azioni $SenseMsg$ e $SendMsg(recipient, msg)$ sono sempre possibili quindi non hanno precondizioni.
Gli effetti delle azioni di comunicazione sono descritti dall'assioma per le postcondizioni:

$$\begin{aligned}
&Poss(agt, act, s) \supset \\
&[MsgQ(agt', do(agt, act, s)) = q \equiv \\
&(\exists m(act = SendMsg(agt', m) \wedge q = append(MsgQ(agt', s), [\langle agt, m\rangle])))) \\
&\vee (act = RmvMsg \wedge agt = agt' \wedge q = rest(MsgQ(agt', s))) \\
&\vee (q = MsgQ(agt', s) \wedge \\
&\forall m(act \neq SendMsg(agt', m)) \wedge (act \neq RmvMsg \vee agt \neq agt'))]
\end{aligned}$$

Utilizzando queste azioni primitive possono essere definite azioni comunicative più complesse:

proc $INFORM(agt, \phi)$
    $know(\phi)?;\ SendMsg(agt, \text{“}inform(\phi)\text{”})$
end

proc $INFORMWHETHER(agt, \phi)$

$INFORM(agt, \phi) \mid INFORM(agt, \neg\phi) \mid INFORM(agt, \neg kwhether(\phi))$
end

proc $REQUEST(agt, \delta)$
$SendMsg(agt, \text{“}request(\delta)\text{”})$
end

proc $QUERYWHETHER(agt, \phi)$
$REQUEST(agt, INFORMWHETHER(self, \phi))$
end

dove $self$ è una costante utilizzata per indicare l'agente che sta eseguendo la richiesta, $know(\phi)$ è vero se ϕ fa parte delle conoscenze dell'agente e $kwhether(\phi)$ è definito come $kwhether(\phi) \equiv know(\phi) \vee know(\neg\phi)$.

proc ScheduleMeeting($organizer, participant, period$)
  for p : partecipant(p) **do**
    REQUEST(ScheduleManager(p), AddToSchedule($p, period, MEETING, organizer$));
    QUERYWHETHER(ScheduleManager(p),AgreedToMeet($p, self, period, organizer$))
  endfor
  while $\neg kwhether(p, self, \forall p[participant(p) \supset$
      AgreedToMeet$(p, self, period, organizer)], now)$ **do**
    SenseMsg;
    if $\neg Empty(MsgQ(self))$ **then** RmvMsg **endif**
  endwhile
  INFORMWHETHER($organizer$,
      $\forall p[participant(p) \supset$ AgreedToMeet$(p, self, period, organizer)]$);
  if $\neg\forall p[participant(p) \supset$ AgreedToMeet$(p, self, period, organizer)]$ **then**
    for p : partecipant(p) **do**
      REQUEST(ScheduleManager(p), RmvFromSched($p, period$))
    endfor
  endif
endproc

Figura 2.8: procedura ScheduleMeeting dell'agente meeting scheduler

L'agente "meeting scheduler", definito dalla procedura di figura 2.9, ha il compito di provare ad organizzare un meeting tra i partecipanti *participant* nel periodo *period*. All'agente "Schedule Manager" di ogni partecipante viene chiesto di aderire al meeting e di informare il "meeting scheduler" di tale adesione. Il "meeting scheduler" legge i messaggi arrivati fino a che non ha informazioni sull'adesione

di tutti i partecipanti. Se non tutti i partecipanti aderiscono alla richiesta di partecipare al meeting, il tentativo di organizzare il meeting fallisce e a tutti i partecipanti viene inviata la richiesta di annullare l'impegno per il meeting.
L'agente "schedule manager" richiama la procedura:

proc ManageSchedule($user$)
  while true do
    SenseMsg;
    if $\neg Empty(MsgQ(self))$ **then**
      if $first(MsgQ(self)) = \langle queryer,$"QUERYWHETHER($\phi$)"$\rangle$ **then**
        INFORMWHETHER($queryer, \phi$) **endif**;
      if $first(MsgQ(self)) = \langle requester,$
          "REQUEST(AddToSchedule($user, period, activ, organizer$))"$\rangle$
        $\wedge$ PermittedToAddToSched($organizer$)
        $\wedge$ $poss(self,$AddToSchedule$(user, period, activ, organizer), now)$
        then AddToSchedule($user, period, activ, organizer$) **endif**;
      if $first(MsgQ(self)) = \langle requester,$"REQUEST(RmvFromSchedule($user, period$))"$\rangle$
        $\wedge$ $poss(self,$RmvFromSchedule$(user, period), now)$
        $\wedge$ $\exists activschedule(user, period, activ,$Owner$(requester), now)$
        then RmvFromSchedule($user, period$) **endif**;
      RmvMsg;
    endif
  endwhile
endproc

Figura 2.9: procedura ManageSchedule dell'agente Schedule Manager

L'agente "Schedule Manager" è in attesa dell'arrivo di un messaggio dall'organizzatore del meeting. Se viene richiesta la partecipazione al meeting l'agente verifica la possibilità di partecipare e se tale possibilità esiste aggiunge il meeting agli impegni dell'utente. Se il messaggio ricevuto consiste nella richiesta di annullamento della partecipazione al meeting l'agente "Schedule Manager" rimuove l'impegno.

Un esempio di programma di attivazione del sistema per l'organizzazione dei meetings è :

by (sm_1, ManageSchedule($user_1$)) $\|$
by (sm_2, ManageSchedule($user_2$)) $\|$
by (sm_3, ManageSchedule($user_3$)) $\|$
by (ms_1, ScheduleMeeting($user_1, \{user_1, user_3\}, noon$)) $\|$
by (ms_2, ScheduleMeeting($user_2, \{user_2, user_3\}, noon$)).

Il programma attiva tre agenti "Schedule Manager" e due agenti "meeting scheduler" che provano entrambi ad ottenere un appuntamento con $user_3$ a mezzogiorno.

2.6.6 Osservazioni

Il comportamento degli agenti ConGolog si basa sulla definizione delle azioni che l'agente può compiere. Ogni azione è specificata da assiomi che caratterizzano le precondizioni che devono essere verificate per effettuare l'azione e da assiomi che asseriscono in quale modo l'esecuzione di un'azione cambia l'ambiente in cui l'agente è collocato.

I costrutti sintattici che formano la struttura del linguaggio permettono di definire procedure, all'interno delle quali vengono eseguite le azioni. Il comportamento dell'agente è quindi specificato dalla definizione delle procedure e dalle loro chiamate.

Un sistema multi-agente descritto con ConGolog consiste in un programma principale che esegue concorrentemente le chiamate di procedura che attivano gli agenti. La comunicazione tra gli agenti del sistema viene realizzata tramite la definizione di azioni primitive in grado di mandare e ricevere messaggi secondo qualche protocollo di comunicazione arbitrario. Utilizzando le azioni primitive si possono sviluppare procedure complesse che gli agenti possono utilizzare, ad esempio, per informare altri agenti di fatti rilevanti oppure per richiedere alcuni servizi ad agenti specializzati nel risolvere compiti particolari.

La capacità di ragionamento negli agenti ConGolog è delegata alle azioni primitive che possono essere definite, ai costrutti sintattici per la concorrenza che fanno parte del linguaggio e alla capacità di comunicazione degli agenti.

2.7 Hindriks et al.: Agent Programming Language

Il linguaggio di specifica proposto in [HdBvdHjCM98] incorpora i principi base sia di AGENT 0 (Sezione 2.1) che di AgentSpeak(L) (Sezione 2.3). Secondo gli autori i due linguaggi sono fondamentalmente simili; le differenze risiedono nelle strutture di controllo.
Il linguaggio proposto combina sia caratteristiche imperative che caratteristiche tipiche della programmazione logica. I concetti di programmazione imperativa vengono utilizzati per descrivere l'esecuzione degli obiettivi, mentre quelli di programmazione logica per descrivere le capacità riflessive (o di meta-ragionamento)

dell'agente.
La semantica operazionale viene definita attraverso l'uso di un sistema di transizione.

In [HdBvdHjCM98] l'agente viene descritto come un'entità con le seguenti caratteristiche:

- uno stato mentale complesso basato su credenze, desideri, piani ed intenzioni che cambiano nel tempo;
- comportamento sia *proattivo*, cioè dipendente dagli obiettivi che l'agente deve perseguire, sia *reattivo*, cioè dipendente dai cambiamenti che si manifestano nell'ambiente in cui è collocato;
- comportamento riflessivo.

Un linguaggio in grado di specificare tali entità deve avere almeno le seguenti caratteristiche:

- operazioni per l'aggiornamento delle credenze in base ai dati osservati, per la comunicazione e per la derivazione di nuovi fatti;
- operazioni per l'aggiornamento degli obiettivi;
- ragionamento pratico per perseguire gli obiettivi.

2.7.1 Descrizione del linguaggio

Le *credenze* sono formule del primo ordine del linguaggio $\mathcal{L}$ definito dalla segnatura Σ. Σ viene definita da una quadrupla $(P,\ F,\ C,\ A)$ in cui P è un insieme di simboli di predicato, F è un insieme di simboli di funzione, C è un insieme di simboli di costante e A un insieme di simboli di azione. L'insieme degli atomi su $\mathcal{L}$ è denotato da At.

Gli *obiettivi* vengono distinti in due categorie: obiettivi che eseguono azioni ed obiettivi che verificano lo stato dell'agente oppure dell'ambiente.
Sia $\Sigma = (P,\ F,\ C,\ A)$ una segnatura, $Gvar$ un insieme di variabili, l'insieme degli obiettivi $\mathcal{L}^g$ è definito induttivamente da:

- $A \subseteq \mathcal{L}^g$: azioni base;
- $At \subseteq \mathcal{L}^g$;
- se $\varphi \in \mathcal{L}$ allora $\varphi? \in \mathcal{L}^g$;

- $Gvar \in \mathcal{L}^g$;
- se $\pi_1, \pi_2 \in \mathcal{L}^g$ allora $\pi_1; \pi_2,\ \pi_1 + \pi_2,\ \pi_1 \| \pi_2 \in \mathcal{L}^g$.

Soddisfare obiettivi della forma $p(t) \in At$ equivale alla ricerca dello stato in cui $p(t)$ vale, mentre il soddisfacimento di quelli della forma $\varphi?$ consiste nella verifica del valore di verità della proposizione φ rispetto alle credenze dell'agente. Gli operatori di composizione tra obiettivi ";", "+" e "$\|$" sono rispettivamente la concatenazione, la scelta non deterministica e l'esecuzione in interleaving tra obiettivi.

Affinché vengano soddisfatti gli obiettivi vengono definite delle *regole di ragionamento pratico* che descrivono il modo in cui viene pianificata l'esecuzione dell'obiettivo.
$\mathcal{L}^p$ è l'insieme delle regole di ragionamento pratico. Una regola $r \in \mathcal{L}^p$ è definito come:

$$\pi \leftarrow \varphi | \pi'$$

dove:

- $\pi \in \mathcal{L}^g$ è la *testa* della regola;
- $\pi' \in \mathcal{L}^g$ è il *corpo* della regola;
- $\varphi \in \mathcal{L}$ è la *guardia* della regola;
- le variabili libere della testa della regola sono chiamate le *variabili globali* della regola;
- le variabili della regola che non sono globali sono chiamate *variabili locali.*

La *guardia* della regola può essere utilizzata sia per specificare il contesto in cui viene applicata la regola sia per ricevere dati dall'insieme delle conoscenze, unificando i parametri della guardia con le credenze.
La distinzione tra variabili locali e globali serve per separare i dati locali utilizzati nell'esecuzione del body dai dati globali utilizzati per comunicare i risultati dell'esecuzione alle altre parti del programma.
Le regole di ragionamento pratico hanno due funzioni: quando la testa della regola ha la forma $p(t) \in At$, il corpo della regola è un piano che descrive la conoscenza procedurale dell'agente nell'eseguire l'obiettivo $p(t)$. Se la testa della regola è un programma arbitrario, il corpo del piano descrive la capacità riflessiva di modificare il comportamento descritto dal programma arbitrario nella testa: il programma che costituisce la testa non viene eseguito, viene invece eseguito il programma che

costituiisce il corpo della regola.
Ad esempio, si consideri uno scenario in cui il robot greedy può muoversi all'interno di un territorio in cui si trovano dei diamanti ed abbia il compito di raccoglierli. Sia G un'istruzione arbitrariamente complessa e si supponga che dopo l'esecuzione di G greedy debba portarsi nella posizione (x, y); l'esistenza della seguente regola di ragionamento pratico, in accordo con la propria guardia, modifica tale comportamento:

$$G; greedy(x, y) \leftarrow$$
$$greedy(x0, y0) \wedge nearest\_diam(x0, y0, x1, y1) \wedge \neg(x = x1 \wedge y = y1) \mid$$
$$greedy(x1, y1)$$

La testa non è formata da un elemento di At ma da un programma arbitrario della forma $G; greedy(x, y)$, tale programma viene sostituito da quello appartenente al corpo della regola: se greedy è nella posizione $(x0, y0)$ e deve raggiungere la posizione (x, y), ma esiste un diamante nella posizione $(x1, y1)$, il robot aggiorna il proprio obiettivo raggiungendo la posizione $(x1, y1)$.

2.7.2 L'agente come programma

Durante l'esecuzione dell'agente le uniche componenti che cambiano sono le credenze e gli obiettivi, mentre l'insieme delle regole di ragionamento pratico e l'insieme delle azioni base non vengono modificate. Da questo punto di vista credenze ed obiettivi caratterizzano lo stato mentale dell'agente rappresentato da una coppia della forma $\langle \Pi, \sigma \rangle$ dove:

- $\Pi \subseteq \mathcal{L}^g$ è la base degli obiettivi;
- $\sigma \subseteq \mathcal{L}$ è la base delle credenze.

Sia $\mathcal{B}$ l'insieme delle possibili basi delle credenze, la semantica di un insieme di azioni base è data dalla funzione di transizione:

$$\mathcal{T} : \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{P}(A)$$

dove $\mathcal{P}(A)$ rappresenta l'insieme delle parti di A. Sia $a \in A$, nel seguito verrà utilizzata la notazione

$$\langle \sigma, \sigma' \rangle a$$

al posto di

$$a \in \mathcal{T}(\sigma, \sigma').$$

Facendo riferimento all'esempio precedente, la semantica dell'azione base *pickup*, che ha come effetto la raccolta di un diamante, ha la forma seguente:

$\langle\{\ldots\,,\ diam(x,y),\ greedy(x,y),\ \ldots\},$
$\{\ldots\,,\ greedy(x,y),\ \neg diam(x,y),\ \ldots\}\rangle\ pickup$

Viste le nozioni di credenza, obiettivo, regola di ragionamento pratico, azione base, un agente può essere visto come una quadrupla della forma:

$$\langle\mathcal{T},\ \Pi_0,\ \sigma_0,\ \Gamma\rangle$$

dove:

- $\mathcal{T}$ è la funzione di transizione che caratterizza la semantica delle azioni base;
- Π_0 è l'insieme iniziale degli obiettivi;
- σ_0 è la base della conoscenza iniziale;
- Γ è l'insieme delle regole di ragionamento pratico.

2.7.3 Semantica operazionale

La semantica operazionale del linguaggio viene descritta da un sistema di transizione tra configurazioni. Una configurazione corrisponde allo stato mentale dell'agente ed una regola di transizione associa una configurazione ad un'altra in cui è stato eseguito un obiettivo, oppure, la testa di una regola per il ragionamento pratico è stata unificata.

L'applicazione di una regola per il ragionamento pratico è definita dalla seguente meta-regola in cui θ è un mgu per π e π', γ è una sostituzione di variabili e V un insieme di variabili:

$$\frac{\pi' \ \leftarrow\ \varphi|\pi''\ \in'\ \Gamma \quad e \quad \sigma \models \forall(\varphi\theta\gamma)}{\langle\pi,\sigma\rangle_V \longrightarrow_{\theta\gamma} \langle\pi''\theta\gamma,\sigma\rangle}$$

dove:

- $\pi' \leftarrow \varphi|\pi'' \ \in' \Gamma$ significa che $\pi' \leftarrow \varphi|\pi''$ è una variante di una regola per il ragionamento pratico appartenente a Γ che ha variabili "fresche" non appartenenti a V;
- γ è una sostituzione tale che per nessuna variabile x, $\gamma(x) \in V$

Nel seguito il simbolo E denoterà la terminazione di una istruzione, quindi $E;\pi$ sarà identificato con π, ed $E\|\pi$ con π.

Eseguire una azione base, come giá descritto precedentemente dalla funzione $\mathcal{T}$, consiste nel modificare le credenze dell'agente:

$$\frac{\langle\sigma,\sigma'\rangle a}{\langle a,\sigma\rangle_V \longrightarrow_{\emptyset} \langle E,\sigma'\rangle}$$

dove $\emptyset$ denota la sostituzione vuota.

L'effetto dei test è controllare se qualche condizione è conseguenza logica della base della conoscenza:

$$\frac{\sigma \models \forall(\varphi\theta)}{\langle\varphi?,\sigma\rangle_V \longrightarrow_{\theta} \langle E,\sigma\rangle}$$

La concatenazione, la scelta non deterministica, l'esecuzione in parallelo sono rispettivamente descritte dalle meta-regole seguenti:

$$\frac{\langle\pi_1,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_1',\sigma'\rangle}{\langle\pi_1;\pi_2,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_1';\pi_2\theta,\sigma'\rangle}$$

$$\frac{\langle\pi_1,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_1',\sigma'\rangle}{\langle\pi_1+\pi_2,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_1',\sigma'\rangle}$$

$$\frac{\langle\pi_2,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_1',\sigma'\rangle}{\langle\pi_1+\pi_2,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_2',\sigma'\rangle}$$

$$\frac{\langle\pi_1,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_1',\sigma'\rangle}{\langle\pi_1\|\pi_2,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_1'\|\pi_2\theta,\sigma'\rangle}$$

$$\frac{\langle\pi_2,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_2',\sigma'\rangle}{\langle\pi_1\|\pi_2,\sigma\rangle_V \longrightarrow_{\theta} \langle\pi_1\theta\|\pi_2',\sigma'\rangle}$$

Sia $\Pi = \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\} \subseteq \mathcal{L}^g$ e $V = Free(\Pi)$ sono le variabili libere in Π. La meta-regola che permette l'esecuzione di un obiettivo dell'insieme di obiettivi Π è:

$$\frac{\langle \pi_i, \sigma\rangle_V \longrightarrow_\theta \langle \pi_i', \sigma'\rangle}{\langle\{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\}, \sigma\rangle_V \longrightarrow_\theta \langle\{\pi_0, \ldots, \pi_{i-1}, \pi_i', \pi_{i+1}, \ldots\}, \sigma'\rangle}$$

Con la semantica operazionale appena descritta l'esecuzione di un agente consiste in una sequenza finita o infinita di stati mentali:

$$\langle \Pi_0, \sigma_0\rangle,\ \langle \Pi_1, \sigma_1\rangle, \ldots,\ \langle \Pi_n, \sigma_n\rangle,\ \ldots$$

tali che per ogni i si ha $\langle \Pi_i, \sigma_i\rangle \longrightarrow \langle \Pi_{i+1}, \sigma_{i+1}\rangle$.

2.7.4 Osservazioni

il linguaggio descritto in [HdBvdHjCM98] è ispirato ai linguaggi di specifica per agenti AGENT 0 e AgentSpeak(L) descritti rispettivamente nelle sezioni 2.1 e 2.3. I costrutti sintattici base di AGENT 0 sono le obbligazioni e le azioni per la comunicazione con altri agenti. Le obbligazioni vengono eseguite in base ai messaggi ricevuti dall'agente allo stato mentale corrente, avendo come effetto l'esecuzione di azioni eventualmente comunicative.
Il linguaggio descritto in questa sezione è sprovvisto di costrutti espliciti per la comunicazione e l'ambiente è rappresentato completamente nelle credenze dell'agente. Non vengono modellati costrutti espliciti per la percezione degli eventi esterni e l'aggiornamento dell'insieme delle credenze avviene come conseguenza dell'esecuzione di azioni.
A differenza delle obbligazioni di AGENT 0, le regole di ragionamento pratico in cui le azioni da eseguire sono combinate dagli operatori di concatenazione, scelta non deterministica ed interleaving, sono un potente meccanismo per il completamento e revisione degli obiettivi.
AgentSpeak(L) è molto simile al linguaggio descritto in questa sezione in quanto è un linguaggio basato su regole. A differenza il linguaggio proposto non basa la scelta delle regole da applicare sulla percezione di eventi, ma sulla scelta non deterministica di uno dei propri obiettivi. Gli agenti non hanno nessuna struttura dati per la gestione degli eventi: utilizzano l'insieme delle credenze per avere una rappresentazione completa dell'ambiente in cui sono collocati, modificano tale rappresentazione attraverso azioni che hanno l'effetto di eliminare e/o aggiungere credenze all'agente. Questo tipo di agenti manifestano un comportamento completamente proattivo.

2.8 Beyssade et al.: Cooperating Logical Agents

In [BEL95] viene presentato un modello basato sulla logica per la progettazione e la realizzazione di sistemi multi-agente. Gli agenti vengono concepiti come processi autonomi con capacità di calcolo, ragionamento complesso e comunicazione tramite scambio di messaggi.

Gli autori dell'articolo non credono che l'estensione del concetto di agente razionale autonomo con costrutti per la concorrenza e comunicazione sia abbastanza per realizzare un sistema multi-agente. Di fondamentale importanza per realizzare il M.A.S. è considerare la natura eterogenea del sistema: ogni agente deve avere il proprio dominio di competenza con conoscenza specifica, specifici metodi e algoritmi per il suo trattamento.

Nella specifica viene enfatizzata l'attività cognitiva dell'agente rispetto allo stato mentale, cioè non viene dato peso allo stato dell'agente ma piuttosto al processo per ottenere quello stato. Da questo punto di vista le "operazioni cognitive" sono dimostrazioni: "A crede P" viene intesa come "A ha una dimostrazione per P".

Il modello proposto è formulato tramite concetti della logica d'ordine superiore (H.O.L. : Higher Order Logic) in modo da specificare e realizzare omogeneamente differenti tipi di agente che costituiscono un M.A.S. eterogeneo: ogni agente ha un proprio comportamento, proprie tecniche di risoluzione dei problemi ed è descritto da una logica specifica.

2.8.1 Definizione del linguaggio

Un M.A.S. consiste in un insieme di processi chiamati *agenti*. Ogni agente è caratterizzato da:

- un linguaggio H.O.L. di rappresentazione della conoscenza L_A;
- un motore computazionale ed un processo per il trattamento della conoscenza, entrambi specifici e non condivisi con nessun altro agente del M.A.S..

Rappresentazione della conoscenza

Il linguaggio per la rappresentazione della conoscenza si basa sul $\lambda - calcolo$ di Church. Ogni agente ha il proprio linguaggio L_A caratterizzato da un insieme di costanti tipate, la stessa nozione di logica dell'ordine superiore ed almeno il seguente insieme di tipi e costanti:

- tipi:
  - *prop* (proposizioni);
  - agt (agenti);
  - arg (argomenti).
- costanti:
  - tt,ff: *prop*;
  - $\neg$: $prop \rightarrow prop$;
  - $\wedge, \vee, \supset, =$: $prop \rightarrow prop \rightarrow prop$;
  - $\forall, \exists : (\alpha \rightarrow prop) \rightarrow prop$;
  - $=, \Longrightarrow$: $(\alpha \rightarrow \alpha) \rightarrow prop$.

Le costanti $\neg, \wedge, \vee, \supset, =$ corrispondono agli usuali connettivi logici. I quantificatori universale ed esistenziale sono codificati dagli operatori $\forall$ ed $\exists$ di tipo rispettivamente $(\forall(\lambda x\ P))$ e $(\exists(\lambda x\ P))$.
Un predicato è qualsiasi costante o termine con tipo:

$$t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow prop$$

rappresentante una relazione tra oggetti del tipo indicato. I simboli $=$ e $\Longrightarrow$ rappresentano rispettivamente l'ugualianza e la valutazione tra predicati.

Giudizi e proposizioni

Gli agenti eseguono ragionamenti ed operazioni di calcolo relativi alla propria conoscenza, i risultati vengono chiamati *giudizi*. Tipici sono “la proposizione P è vera”, oppure “l'espressione E viene valutata in V”. Nel modello proposto vi è differenza tra proposizione e giudizio: una proposizione non ha un valore di verità, esprime una relazione tra oggetti e non interessa se questa relazione vale in qualche contesto o situazione. Un giudizio invece, è una affermazione che un agente asserisce oppure nega.
Un agente è un programma in $\lambda - prolog$: il $\lambda - prolog$ viene utilizzato come metalinguaggio per esprimere giudizi all'interno dell'agente. Il tipo di una $\lambda - formula$ che rappresenta un giudizio è “o” (simile ma distinto da *prop* nel linguaggio di rappresentazione della conoscenza).
I giudizi sono espressi da costanti:

- $TRUE : prop \rightarrow o$, $(TRUE\ P)$ significa “P è vero”;

- $DEMO : arg \to prop \to o$, $(DEMO\ \rho\ P)$ significa "è stata trovata una prova ρ per P";
- $PROOF : arg \to prop \to o$, $(PROOF\ \rho\ P)$ significa "è stato verificato che ρ è una prova per P";
- $MEMO : arg \to prop \to o$, $(MEMO\ \rho\ P)$ significa "P è in memoria con posizione ρ".

Comunicazione

Per garantire l'autonomia degli agenti è stata definita una rappresentazione della conoscenza eterogenea all'interno del M.A.S., eterogeneità che causa problemi di comunicazione. Un primo requisito per far si che possa avvenire la comunicazione è che gli agenti condividano un linguaggio L_C con cui possano costruire messaggi da scambiare.

Il linguaggio L_C appartiene all'intersezione di tutti i linguaggi L_A. Ogni linguaggio L_A contiene inoltre la costante *interp* tale che il giudizio:

$$TRUE(interp\ E\ C)$$

viene letto: "l'espressione E è interpretata dal concetto C". Nel seguito l'espressione 'C' del linguaggio L_C sarà interpretata dalla costante C del linguaggio L_A.

I messaggi scambiati dagli agenti sono composti da una intestazione h, che contiene informazioni riguardanti il mittente, il destinatario, il tipo di messaggio scambiato e da espressioni del linguaggio L_C e da costanti che riguardano il tipo di messaggio scambiato. Ad esempio:

- $(decide\ h\ `P')$ è un messaggio con intestazione h e contenuto 'P' il cui significato è una richiesta di decidere se la proposizione P, codificata in 'P' nel linguaggio L_C, è vera;
- $(which\ h\ `(\lambda x : \alpha\ Q)')$ significa "quale x soddisfa Q?".

2.8.2 Significato delle modalità epistemiche

Una questione centrale nella specifica dei M.A.S. è la differenza tra conoscenza *oggettiva* e *soggettiva* dell'attività mentale di un agente da parte di un altro agente.
Negli approcci tradizionali tale conoscenza è espressa *oggettivamente* attraverso l'uso di un linguaggio logico la cui semantica è comune a tutti gli agenti.

Nel modello proposto tale conoscenza è espressa, seppur in modo parziale ed incompleto, *soggettivamente*: un agente è in grado di rappresentare l'attività mentale di un altro agente nel proprio sistema concettuale, malgrado i due sistemi concettuali non abbiano niente in comune. La conseguenza è una *società* di agenti cognitivi in cui le modalità epistemiche sono in grado di specificare il comportamento cognitivo dei membri della società.

Specificare l'attività di un agente consiste nel descrivere la sua abilità nel derivare giudizi. Per ottenere ciò il linguaggio per la rappresentazione della conoscenza di ogni agente viene esteso con quattro costanti corrispondenti ai quattro tipi di giudizi definiti nella sezione 2.8.1:

- $TRUE^{\star} : agt \rightarrow prop \rightarrow prop$;
- $DEMO^{\star} : agt \rightarrow arg \rightarrow prop \rightarrow prop$;
- $PROOF^{\star} : agt \rightarrow arg \rightarrow prop \rightarrow prop$;
- $MEMO^{\star} : agt \rightarrow arg \rightarrow prop \rightarrow prop$.

con la seguente semantica informale:

- $TRUE^{\star}\ A\ P$ significa $\vdash_A TRUE\ P$;
- $DEMO^{\star}\ A\ \rho\ P$ significa $\vdash_A DEMO\ \rho\ P$;
- $PROOF^{\star}\ A\ \rho\ P$ significa $\vdash_A PROOF\ \rho\ P$;
- $MEMO^{\star}\ A\ \rho\ P$ significa $\vdash_A MEMO\ \rho\ P$.

Queste quattro costanti permettono la specifica della conoscenza dell'agente A nel linguaggio di rappresentazione della conoscenza dell'agente B.

Vi sono differenti nozioni di credenze: credenze esplicite e credenze implicite. Un agente crede esplicitamente P se è "convinto" di P, oppure se può ottenere immediatamente tale convinzione.
La credenza esplicita viene formalizzata utilizzando la costante $MEMO^{\star}$:

$$explicit\_bel\ A\ p\ =\ \exists\rho(MEMO^{\star}\ A\ \rho\ P)$$

Un agente crede implicitamente P se P può essere ottenuta da qualche processo deduttivo. I processi deduttivi per ottenere la prova di P possono essere di due tipi: la prova di P può essere cercata, oppure l'agente può verificare la validità di una data prova per P:

$$\begin{aligned} implicit\_bel1\ A\ p &= \exists\rho(DEMO^\star\ A\ \rho\ P) \\ implicit\_bel2\ A\ p &= \exists\rho(PROOF^\star\ A\ \rho\ P) \end{aligned}$$

Una proprietà importante di un agente è la *competenza* rispetto ad un problema:

$$competent\ A\ P\ \supset\ (\exists\rho(PROOF^\star\ A\ \rho\ P)\ \supset\ P)$$

Un agente è competente su P se accetta una prova per P e se P vale.

Attraverso la nozione di competenza può essere definita una nozione di verità che non dipende dalla convinzione di un singolo agente, ma che coinvolge un insieme di agenti:

$$\begin{aligned} valid\ \rho\ P\ =\ &\exists X(competent\ X\ P)\ \wedge\ \forall X((competent\ X\ P)\ \supset \\ &(PROOF^\star\ X\ \rho\ P)) \end{aligned}$$

Ad esempio, un teorema matematico è considerato "valido" se per la comunità matematica lo è .

Il significato del verbo conoscere è aperto a diverse interpretazioni. Nel modello proposto viene inteso come una "credenza valida socialmente": un agente conosce P se ha una dimostrazione per P accettata dagli agenti competenti su P. Questo concetto viene formalizzato nel modo seguente per la conoscenza implicita:

$$knows^+\ A\ P\ =\ \exists\rho(PROOF^\star\ A\ \rho\ P)\ \wedge\ (valid\ \rho\ P)$$

mentre per la conoscenza esplicita la costante $PROOF^\star$ viene sostituita con la costante $MEMO^\star$:

$$knows^-\ A\ P\ =\ \exists\rho(MEMO^\star\ A\ \rho\ P)\ \wedge\ (valid\ \rho\ P)$$

2.8.3 Eterogeneità e comunicazione

A causa dell'eterogeneità della rappresentazione della conoscenza tra gli agenti del M.A.S., può venir meno la mutua comprensione in seguito ad una qualche azione comunicativa; poichè la stessa espressione appartenente al linguaggio L_C in ognuno dei sistemi concettuali ha una propria interpretazione e tali sistemi concettuali, a priori, non hanno niente in comune.
Una soluzione a questo problema è postulare un qualche tipo di corrispondenza tra i sistemi di rappresentazione della conoscenza degli agenti.
Ad ogni tipo del linguaggio L_A (ad esempio *prop*, *int*, *obj*)viene associato un

insieme di "concetti astratti" chiamati *abscons*, in modo che ad ogni termine di L_C venga associato un elemento di *abscons* che ne caratterizzi il significato. Ad esempio il concetto astratto $(fact\ 5)$ significa "il valore della funzione fattoriale applicata a 5" ed il concetto astratto 120 significa "il numero intero 120".
La sintassi del linguaggio L_C è condivisa da tutti gli agenti del M.A.S., ma il significato delle espressioni è definito in modo proprio da ogni agente attraverso il predicato *interp*. Ad esempio due agenti A e B danno lo stesso significato al termine T di L_C se per qualche concetto astratto C, $TRUE$ $(interp\ T\ C)$ vale sia per A che per B. Questa situazione è descritta dalla figura 2.10.

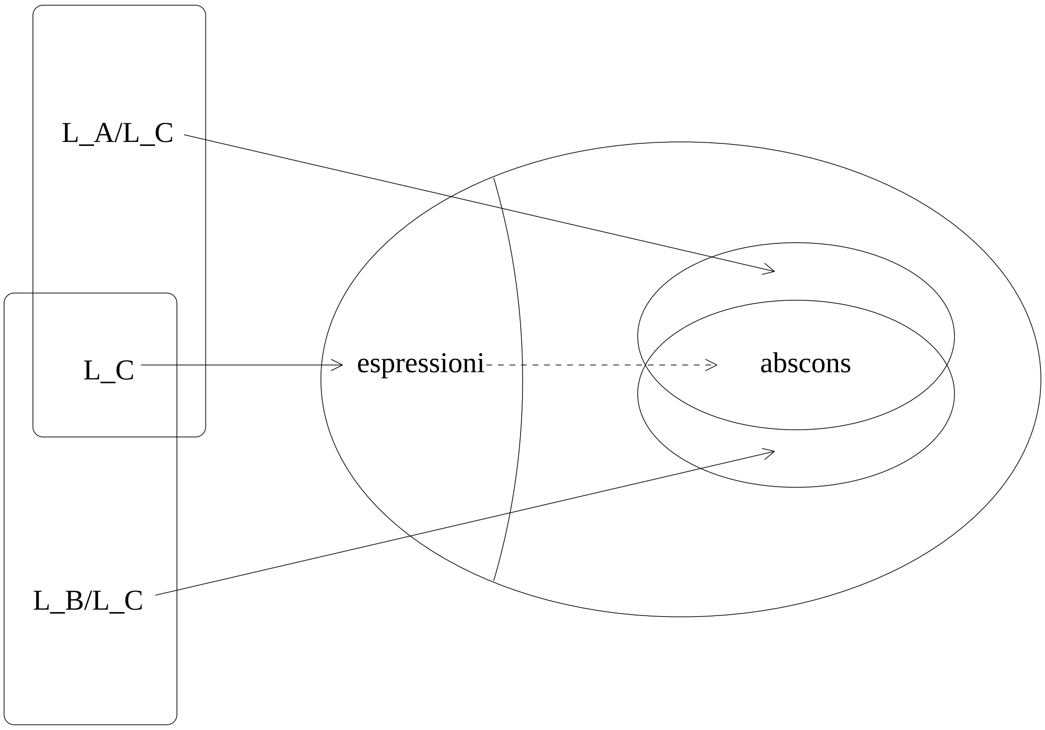


Figura 2.10: I linguaggi ed i loro domini concettuali.

Un esempio di situazione complessa è quello in cui l'agente A ha la seguente definizione di commutatività:

$$commutativo\ =\ \lambda op(\lambda x, y(op\ x\ y)\ =\ \lambda x, y(op\ y\ x))$$

mentre l'agente B *sa* che l'operazione "+" è commutativa ma non ha una definizione del termine *commutativo*.
L'agente A può validamente asserire sia:

$(bel\ B(commutativo\ +))$

sia:

$(bel\ B(\lambda x, y(+\ x\ y)\ =\ \lambda x, y(+\ y\ x)))$

ma A può anche credere che B non conosca il significato del termine *commutativo*:

$\neg\exists C(bel\ B(interp\ \text{`}commutativo\text{'}\ C)$

Rispetto alla conoscenza di A, "*commutativo*" e "$(\lambda x, y(+\ x\ y)\ =\ \lambda x, y(+\ y\ x))$", rappresentano lo stesso concetto astratto condiviso con B, cioè la commutatività. D'altra parte A e B non possono comunicare attraverso il linguaggio L_C perché B non conosce il significato del termine *commutativo*.

2.8.4 Osservazioni

Nel modello proposto il M.A.S. è un insieme di processi indipendenti ognuno caratterizzato da un insieme di credenze/conoscenze, da un motore computazionale per l'esecuzione dei calcoli e da un sotto-processo per il trattamento delle credenze/conoscenze. L'articolo che descrive questo modello [BEL95] non si sofferma né sul motore computazionale né sul processo per il trattamento delle credenze/conoscenze, ma descrive approfonditamente il formalismo e le modalità utilizzate per rappresentare la conoscenza.

All'interno del M.A.S. gli agenti possono avere architetture e comportamenti diversi tra loro, tutti descritti omogeneamente dal linguaggio logico d'ordine superiore utilizzato nel modello (λ-prolog). Possono essere equivalentemente descritti sia agenti reattivi che proattivi il cui differente comportamento è essenzialmente caratterizzato dalle credenze/conoscenze dell'agente.

La comunicazione tra gli agenti è un aspetto importante e complicato. Viene realizzata tramite un apposito linguaggio condiviso tra tutti gli agenti del sistema, i cui termini possono essere interpretati diversamente da ogni agente; solo in questo modo si ottiene una rappresentazione "soggettiva" della conoscenza.

2.9 Kinny et al.: Modelling and Design of Multi-Agent Systems

In [KG96] viene presentata una tecnica per modellare agenti e sistemi multi-agente che adatta ed estende le tradizionali tecniche di rappresentazione orientate ad oggetti.

Tale metodologia è basata su di una struttura per la specifica e la progettazione di sistemi con architettura BDI.

Nello specificare agenti viene utilizzato un insieme di modelli che operano a due livelli di astrazione:

- *livello esterno*: il sistema viene modellato come una gerarchia di classi tra le quali esiste una relazione di ereditarietà e le cui istanze sono agenti individuali. Tali classi sono caratterizzate dall'utilizzo, responsabilità, servizi offerti, informazioni richieste ed interazioni con il mondo esterno;

- *livello interno*: consiste in un insieme di modelli che descrivono la manipolazione dell'informazione ed il controllo delle strutture dati che determinano il comportamento dell'agente.

La metodologia di progetto è basata sull'identificazione dei ruoli chiave di una applicazione. Ruoli e relazioni tra ruoli permettono la realizzazione di una gerarchia di classi-agente.
L'analisi delle responsabilità di ogni ruolo, rappresentato da una classe-agente, conduce all'identificazione dei servizi che l'agente deve offrire, le interazioni con gli altri agenti e gli obiettivi ed eventi a cui deve rispondere.

2.9.1 Una tecnica di modellazione orientata agli agenti

Le tecniche di modellazione orientate ad oggetti descrivono un sistema identificando le classi in un dominio applicativo e specificandone il comportamento e le relazioni con altre classi.
I dettagli riguardanti il progetto di un sistema ad oggetti possono essere riassunti in tre modelli:

- *modello oggetto*: descrive le informazioni riguardanti gli oggetti all'interno del sistema. Descrive la loro struttura dati, le relazioni e le operazioni supportate;

- *modello dinamico*: descrive gli stati, le transizioni, gli eventi, le azioni, le attività e le interazioni che caratterizzano il comportamento del sistema;

- *modello funzionale*: descrive il flusso dei dati durante l'attività del sistema sia all'interno delle componenti sia tra di loro.

Nella specifica di sistemi multi-agente, viene proposto un insieme di modelli più specializzato, che opera secondo due livelli di astrazione: livello *esterno* e livello *interno*.
Per la formalizzazione del *livello esterno* vengono considerati i modelli seguenti:

- *modello agente*: descrive la gerarchia tra classi-agente concrete ed astratte (le classi concrete sono istanziabili, quelle astratte no). Identifica le istanze di agenti presenti all'interno del sistema, la loro molteplicità e la loro creazione;

- *modello d'interazione*: descrive le responsabilità di una classe-agente, i servizi offerti, le interazioni e le relazioni di controllo tra classi-agente. In questo modello vengono quindi definite sintassi e semantica dei messaggi scambiati tra gli agenti, e tra agenti ed altre componenti quali le interfacce utente.

Per la formalizzazione del *livello interno* vengono considerati gli elementi necessari per modellare l'architettura BDIdell'agente.

- *modello delle credenze*: descrive le informazioni riguardanti lo stato interno dell'agente, l'ambiente in cui è collocato e le azioni che può eseguire, le possibili credenze di un agente e le loro proprietà: le credenze possono cambiare oppure no durante l'esecuzione.
  Le credenze sono contenute nell'*insieme delle credenze*, mentre le credenze iniziali sono contenute nello *stato delle credenze*;

- *modello degli obiettivi*: descrive gli obiettivi che un agente può adottare e gli eventi a cui esso risponde. Il modello è costituito dall'*insieme degli obiettivi* che caratterizza il dominio degli obiettivi, dall'insieme degli eventi, e dallo *stato degli obiettivi* che descrive gli obiettivi iniziali dell'agente;

- *modello dei piani*: descrive i piani che un agente può impiegare in seguito al manifestarsi di un evento oppure per completare i propri obiettivi. Il modello dei piani è costituito da un *insieme dei piani* che descrive le proprità e le strutture di controllo di ogni singolo piano.

2.9.2 Modellazione di un singolo agente

Il modello delle credenze

Il *modello delle credenze* è costituito da un *insieme delle credenze* e da uno o più *stati delle credenze.*
L'*insieme delle credenze* è descritto da un insieme di *diagrammi oggetto* che definiscono il dominio delle credenze di una classe-agente. Uno *stato delle credenze* è un insieme di *diagrammi istanza* ognuna dei quali definisce un'istanza dell'*insieme delle credenze.*

L'insieme delle credenze: è un insieme di predicati e funzioni i cui argomenti sono termini definiti su domini che possono essere sia predefiniti che definiti dall'utente. Questi domini, predicati e funzioni sono direttamente derivati dalle definizioni di credenze all'interno dei diagrammi oggetto che descrivono l'insieme delle classi. Ogni definizione di classe definisce uno o più domini ed ogni attributo, operazione, associazione definisce schemi di predicato e/o di funzione.

Un diagramma oggetto dell'insieme delle credenze è un grafo aciclico e diretto, i cui nodi rappresentano sia classi astratte che concrete (istanziabili). Nei diagrammi, le classi sono rappresentate da icone. Le classi astratte mostrano il simbolo **A** nella parte superiore. Gli archi del grafo che rappresentano la relazione di ereditarietà contengono un triangolo con un vertice che punta verso la superclasse.

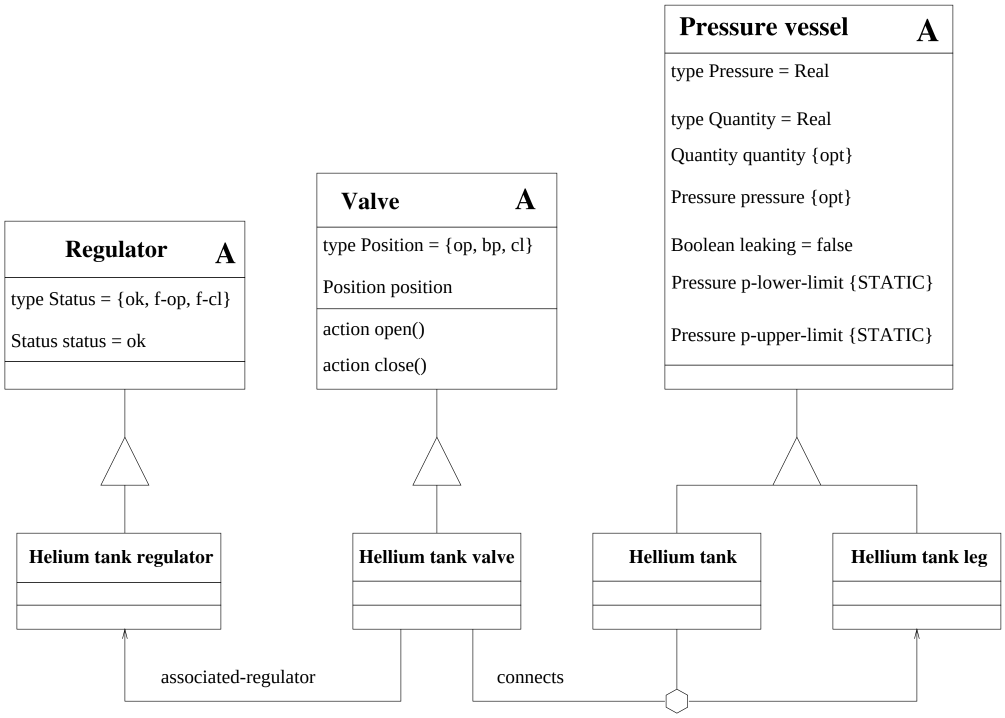
Ogni classe definisce i tipi, gli attributi dell'oggetto, le funzioni ed i predicati che coinvolgono l'oggetto.
L'insieme delle credenze è formalmente definito come segue:

> Sia $\mathcal{I}$ un insieme di identificatori, $\mathcal{D} = \{D_1, \ldots, D_n\}$ un insieme di domini e $\mathcal{Z}$ un insieme di specificatori di proprietà delle credenze.
>
> - Un attributo A è una coppia della forma $\langle n, D_i \rangle$, dove n è il nome dell'attributo, e D_i il tipo. Il tipo dell'attributo è denotato da $dom(A)$;
> - Uno schema di predicato su $\{\mathcal{I}, \mathcal{D}, \mathcal{Z}\}$, denotato da $p(A_1, \ldots, A_k)$ è una tupla $\langle p, Y, A_1, \ldots, A_k \rangle$ tale che $p \in \mathcal{I},\ Y \subset \mathcal{Z}$, $\forall i = 1 \ldots k,\ dom(A_i) \in \mathcal{D}$;
> - Uno schema di funzione su $\{\mathcal{I}, \mathcal{D}, \mathcal{Z}\}$, denotato da $f(A_1, \ldots, A_{k-1}) \rightarrow A_k$ è una tupla $\langle f, Y, A_1, \ldots, A_k \rangle$ tale che $f \in \mathcal{I},\ Y \subset \mathcal{Z},\ \forall i = 1 \ldots k,\ dom(A_i) \in \mathcal{D}$;
> - l'*insieme delle credenze* è una tupla $\langle \mathcal{I}, \mathcal{D}, \mathcal{Z}, P, F \rangle$ dove P è un insieme di schemi di predicati su $\{\mathcal{I}, \mathcal{D}, \mathcal{Z}\}$ e F è un insieme di schemi di funzione su $\{\mathcal{I}, \mathcal{D}, \mathcal{I}\}$.

La figura 2.11 mostra il diagramma relativo ad alcune classi rappresentanti le credenze di una applicazione per la diagnosi dei guasti a bordo dello shuttle.

Proprità delle credenze: i predicati e le funzioni che costituiscono le credenze di un agente possono essere classificate come: *estensionali*, *valutabili* oppure *effimeri*.

- i predicati estensionali sono relazioni sulle credenze;



predicati derivati:

status(Regulator, Status)

position(Valve, Position)

quantity(Pressure-vessel, Quantity)

pressure(Pressure-vessel, Pressure)

leaking(Pressure-vessel, Boolean)

p-lower-limit(Pressure-vessel, Pressure)

p-upper-limit(Pressure-vessel, Pressure)

associated-regulator(Hellium-tank-valve, Hellium-tank-regulator)

connects(Hellium-tank-valve, Hellium-tank, Hellium-tank-leg)

funzioni derivate:

Status status(Regulator)

Position position(valve)

Boolean leaking(Pressure-vessel)

Pressure p-lower-limit(Pressure-vessel)

Pressure p-upper-limit(Pressure-vessel)

Figura 2.11: Classi delle credenze, predicati e funzioni derivate.

- le funzioni estensionali sono generate da alcuni predicati estensionali;
- predicati e funzioni valutabili sono calcolati quando richiesto. Le azioni sono particolari predicati valutabili che manifestano il loro effetto sull'ambiente esterno;
- i predicati effimeri rappresentano proprietà o associazioni che non vengono ricordati da istante ad istante.

I predicati e le funzioni sia estensionali che valutabili, possono essere statici oppure dinamici. Se statici rappresentano relazioni fisse e funzioni la cui definizione non cambia durante l'esecuzione. Se dinamici, sia le definizioni dei predicati che quelle delle funzioni, cambiano durante l'esecuzione.
Le proprità di predicati e funzioni sono rappresentate nei diagrammi da parole chiave. Ad esempio nella figura 2.11 l'attributo $p - lower - limit$ della classe $Pressure\ vessel$ ha la proprietà $STATIC$, indicante che la sua definizione non cambia nel tempo, l'attributo $pressure$ ha la proprietà *opt*, indicante che il suo valore può essere indeterminato.

Stato delle credenze: consiste in un insieme di istanze di predicati estensionali, descritti da un insieme di diagrammi contenenti istanze che rappresentano oggetti e associazioni tra loro. Stato e modello delle credenze sono formalmente definiti come segue:

Sia $\mathcal{B} = \langle \mathcal{I}, \mathcal{D}, \mathcal{Z}, P, F \rangle$ un insieme di credenze.

- L'istanza di un predicato $p(A_1, \ldots, A_k) \in P$ è una tupla della forma $p(a_1, \ldots, a_k) \in A_1 \times \cdots \times A_k$;
- uno stato delle credenze $\mathcal{S}$ di $\mathcal{B}$ è un insieme di istanze di predicati in P;
- un modello delle credenze è una coppia $\langle \mathcal{B}, \mathcal{S} \rangle$, dove $\mathcal{S}$ è un insieme di stati di credenza di $\mathcal{B}$.

Il modello degli obiettivi

Ad ogni classe-agente è associato un modello degli obiettivi che consiste in un *insieme degli obiettivi* ed uno o più *stati degli obiettivi*.

Uno stato degli obiettivi è un insieme di istanze ground di elementi appartenenti all'insieme degli obiettivi. Può essere usato per inizializzare lo stato mentale dell'agente.

L'insieme degli obiettivi è costituito da un insieme di segnature che descrivono le formule rappresentanti gli obiettivi e gli eventi.
Ogni formula è formata da un operatore modale applicato ad un predicato appartenente all'insieme delle credenze.
Gli operatori modali sono:

- ! (realizzazione): l'obiettivo di realizzazione $!\phi$ viene completato con successo se ϕ è creduto essere vero dall'agente. Nel caso in cui ϕ non fosse vero, viene selezionato un piano; se l'esecuzione del piano termina con successo l'obiettivo ha successo e ϕ viene creduto vero dall'agente, altrimenti l'obiettivo fallisce;

- ? (verifica): l'obiettivo di verifica $?\phi$ viene completato con successo se ϕ è vero per l'agente e fallisce se l'agente crede che ϕ non sia vero. Nel caso in cui il valore di verità sia indefinito, viene eseguito un piano. Se l'esecuzione del piano termina con successo l'obiettivo viene considerato completato con successo ed il predicato ϕ viene creduto vero dall'agente, altrimenti l'obiettivo fallisce;

- \$ (determinazione): l'obiettivo di determinazione $\$\phi$ viene completato con successo se il predicato ϕ è creduto vero oppure falso, mentre comporta l'esecuzione di un piano se il suo valore di verità è indefinito. Se il piano termina con successo anche l'obiettivo ha successo ed il valore di verità di ϕ non è più indefinito;

La definizione degli obiettivi richiede compatibilità tra credenze ed obiettivi: tutti gli obiettivi devono essere definiti su predicati che appartengono alle credenze dell'agente.
L'insieme e lo stato degli obiettivi sono definiti formalmente come:

Sia $\mathcal{B} = \langle \mathcal{I}, \mathcal{D}, \mathcal{Z}, P, F \rangle$ l'insieme di credenze.

- L'insieme degli obiettivi $\mathcal{B}$-compatibili G è una tripla della forma $\langle o, p, Y \rangle$, dove $o \in \{!, ?, \$\}$, $p \in P$ e Y è l'insieme di specificatori di proprietà degli obiettivi;
- L'insieme di stati degli obiettivi $\mathcal{B}$-compatibili T è un insieme di coppie della forma $\langle o, q \rangle$, dove $o \in \{!, ?, \$\}$ e q è una istanza ground di un predicato $p \in P$.

Non tutte le combinazioni tra gli operatori modali ed i predicati hanno senso: ad esempio un obiettivo di determinazione applicato ad un predicato valutabile avrà sempre successo.

Il modello dei piani

Il modello dei piani è costituito da un insieme di piani, ognuno rappresentato da un diagramma nel quale gli attributi appartengono ad un insieme predefinito di attributi riservati.

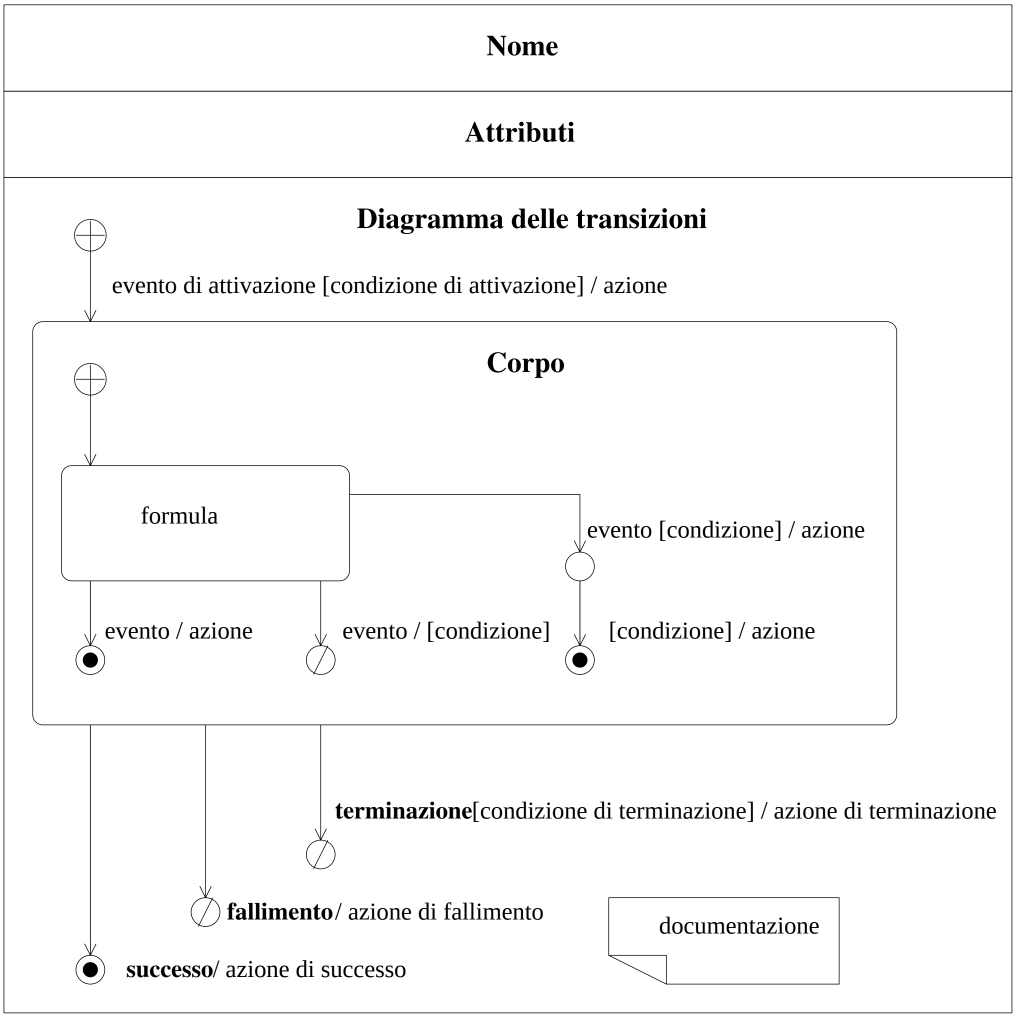


Figura 2.12: Rappresentazione di un piano.

Nella figura 2.12 viene mostrato il diagramma di un piano generico. Il diagramma è diviso in tre sezioni: il nome del piano, gli attributi, ed il diagramma delle transizioni del piano.

Il diagramma delle transizioni è costituito da archi diretti che denotano transizioni tra gli stati e tre tipi di nodi che denotano i tre possibili stati del diagramma: stato di attivazione, stato interno e stato finale.
Lo stato di attivazione è rappresentato dal simbolo $\oplus$. Lo stato finale può essere di successo oppure di fallimento, è rappresentato dal simbolo $\odot$ nel primo caso, dal simbolo $\oslash$ nel secondo. Lo stato interno può essere attivo oppure passivo: lo stato interno passivo non ha associato nessuna sottostruttura ed è rappresentato dal simbolo $\bigcirc$, mentre allo stato interno attivo viene associata una attività rappresentata da una icona. Lo stato interno attivo è un sotto obiettivo rappresentato da formule che descrivono obiettivi eventualmente contenute in cicli while oppure da un sotto grafo.

Le transizioni da uno stato all'altro avvengono quando si manifesta un evento e viene verificata una data condizione. In conseguenza ad una transizione può essere eseguita una azione.

Fallimento dei piani: Si manifesta quando una azione associata ad una transizione fallisce, quando all'interno del diagramma delle transizioni viene eseguita una transizione ad uno stato finale di fallimento oppure quando l'esecuzione di uno stato attivo termina con il fallimento e nessun'altra transizione è permessa.

Esecuzione dei piani: Alla transizione iniziale di un diagramma rappresentante un piano sono associati un evento di attivazione ed una condizione di attivazione che determinano quando, ed in quale contesto, il piano comincia la propria esecuzione. L'evento di attivazione può manifestarsi in funzione di un cambiamento esterno, di un cambiamento tra le credenze dell'agente oppure in conseguenza dell'esecuzione di un sotto obiettivo di un altro piano.

Quando più piani sono associati allo stesso evento ed allo stesso contesto vengono eseguiti in parallelo, mentre vengono eseguiti in successione, fino alla terminazione con successo di uno di loro, se la loro esecuzione è causata da un obiettivo.

Proprietà dei piani: Ad ogni piano possono essere associate delle proprietà che ne influenzano l'esecuzione:

- priorità: determina la priorità con cui vengono eseguiti in parallelo i piani associati ad un evento e ad una condizione;
- precedenza: determina l'ordine in cui vengono eseguiti diversi piani associati ad un singolo obiettivo;

- non-riesecuzione: specifica che l'obiettivo non deve essere provato con un altro piano se il piano corrente è terminato con il fallimento.

2.9.3 Modellazione di un sistema multi-agente

Il modello agente ha due componenti: il *modello delle classi-agente* ed il *modello delle istanze*.
Il modello delle classi-agente definisce le classi-agente astratte e concrete, le relazioni di ereditarietà e di aggregazione. Il modello delle istanze definisce invece le istanze delle classi concrete (solo le classi concrete sono istanziabili) e le loro proprietà.

Il modello delle classi-agente è un grafo diretto e aciclico i cui nodi sono icone rappresentanti classi astratte e concrete ed i cui archi rappresentano le relazioni di ereditarietà ed aggregazione.
Gli attributi delle classi non sono arbitrari ed appartengono ad un insieme predefinito, ad esempio, ad ogni classe devono essere associati credenze, obiettivi e piani rappresentati dagli attributi CREDENZE, OBIETTIVI e PIANI.

L'ereditarietà è una relazione del tipo "is a": permette la definizione di un agente come estensione o raffinamento del super-agente, mentre l'aggregazione è una relazione del tipo "part off".

L'agente eredita il modello delle credenze, degli obiettivi e dei piani dal super-agente.
Gli elementi dell'insieme delle credenze di un agente sono predicati e funzioni derivati dal diagramma delle credenze e sono combinati con il modello delle credenze del super-agente: ai tipi, predicati e funzioni derivati dal diagramma delle credenze della classe-agente vengono aggiunti tipi, predicati e funzioni della super-classe-agente, ottenendo così l'estensione delle credenze dell'agente. I tipi, predicati e funzioni definiti nella super-classe-agente non vengono sovrapposti agli stessi giá presenti nella classe-agente, in questo modo si realizza il meccanismo di raffinamento delle credenze.
L'agente eredita gli obiettivi del super-agente nello stesso modo in cui eredita le credenze, può quindi estendere e specializzare gli obiettivi ereditati ridefinendo ad esempio le loro proprietà.
Anche i piani ereditati dalla super-classe-agente vengono combinati con quelli della classe-agente in modo analogo alle credenze: ai piani della sotto-classe-agente vengono aggiunti i piani della super-classe-agente compatibili con le credenze e gli obiettivi della sotto-classe-agente ed il cui nome non compare tra i piani della

sotto-classe-agente e neppure nei piani di eventuali altri super-classi-agente.

L'aggregazione consiste nell'incapsulamento di un insieme di "sotto-agenti" da parte di un agente. I "sotto-agenti" aggregati non hanno credenze, obiettivi e piani in comune tra loro.

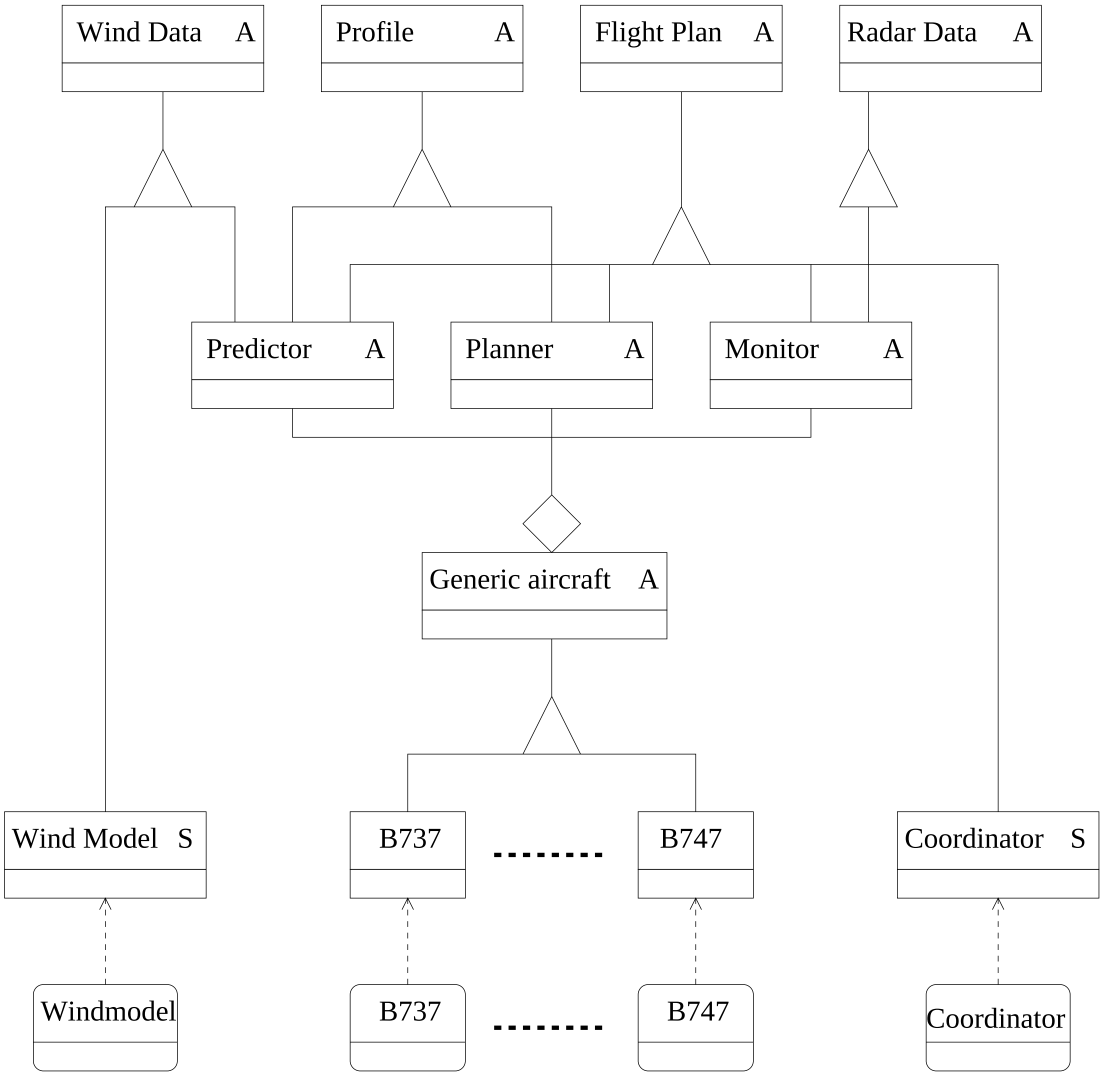


Figura 2.13: Classi-agente e loro istanze.

Ad esempio, con riferimento alla figura 2.13, *Monitor* è un agente astratto "sotto-agente" di *Generic Aircraft*. *Monitor* è sia un agente *Radar Data* che un agente *Flight Plan*. Anche *Predictor* è un "sotto-agente" di *Generic Aircraft*

ed è sia un *Wind Data* che un *Profile* che un *Flight Plan. Monitor* e *Predictor* non condividono credenze riguardanti i piani di volo, *Monitor* non ha credenze riguardanti i dati del vento e *Predictor* non ha credenze riguardanti i dati del radar.
Nella figura 2.13, le icone che hanno il simbolo **S** nell'intestazione rappresentano classi concrete che devono essere istanziate all'inizializzazione del sistema, mentre le classi concrete che non hanno il simbolo **S** possono essere istanziate durante l'esecuzione del sistema. Le istanze delle classi concrete sono rappresentate con archi tratteggiati.

2.9.4 Osservazioni

Nell'articolo [KG96] viene proposto una metodologia di specifica che descrive sistemi multi-agente basati su agenti BDI, ottenuta adattando metodologie di specifica orientate ad oggetti giá esistenti.

I meccanismi di ereditarietà e di aggregazione permettono la definizione di agenti che possono condividere ed estendere sia lo stato mentale, rappresentato dalle credenze, dagli obiettivi e dai piani, che la conoscenza procedurale da cui deriva il comportamento dell'agente.

Dal punto di vista interno l'agente viene rappresentato da tre modelli: uno per le credenze, uno per gli obiettivi ed uno per i piani. Ognuno di questi tre modelli viene rappresentato con diagrammi costituiti dalle classi che sono oggetto del modello e dalle relazioni tra di esse.
Dal punto di vista esterno, il sistema multi-agente è rappresentato dal modello agente che descrive la struttura gerarchica tra gli agenti, cioè le relazioni di ereditarietà, di aggregazione e la creazione delle istanze e della loro molteplicità.

La comunicazione tra agenti non viene affrontata, ma può essere ottenuta dall'esecuzione di azioni specifiche; la ricezione di un messaggio potrebbe consistere, in tal caso, nella generazione di un evento in grado di invocare l'esecuzione di un piano.

2.10 Conclusioni

Alcuni linguaggi di specifica trattati nella rassegna sono abbastanza generali da poter descrivere sistemi multi-agente composti da agenti caratterizzati da architetture arbitrarie e comportamenti arbitrari:

- Concurrent MetateM;

- DESIRE;
- ConGolog;

Altri specificano sistemi multi-agente composti da agenti il cui comportamento è arbitrario ma descritto da qualità mentali:

- AGENT 0;
- il linguaggio descritto nella sezione 2.7;
- il linguaggio descritto nella sezione 2.8.

Altri ancora, specificano sistemi multi-agente composti esclusivamente da agenti con architettura BDI:

- dMARS;
- AgentSpeak(L);
- il linguaggio descritto nella sezione 2.9.

I requisiti che caratterizzano l'agente e che il linguaggio di specifica deve essere in grado di descrivere sono giá stati commentati nella sezione e sono:

- autonomia;
- reattività;
- proattività;
- capacità sociale.

Nel seguito viene considerato l'adattamento dei linguaggi presentati in questo capitolo alla modellazione di queste quattro caratteristiche.

2.10.1 Autonomia

Consiste nella capacità di operare senza l'intervento di entità esterne e di avere controllo sul proprio comportamento: durante la propria attività, l'agente ha l'iniziativa sulla "scelta" delle azioni da eseguire, eventualmente modificando la strategia intrapresa per completare i propri obiettivi oppure per reagire agli eventi percepiti, è inoltre in grado di modificare il proprio stato interno.

Il comportamento e lo stato interno degli agenti Concurrent MetateM sono specificati da un insieme di assiomi della logica temporale del primo ordine MetateM. Gli assiomi caratterizzano quali sono le componenti dello stato interno ed in

che modo l'agente le modifica, descrivono quali sono le azioni che possono essere eseguite e la capacità di iniziativa dell'agente.
L'autonomia degli agenti Concurrent MetateM è quindi determinata non da una caratteristica del linguaggio, ma dal particolare insieme di assiomi che definisce l'agente.

Lo stato interno di un agente DESIRE è costituito dallo stato delle componenti di cui è costituito e dalle relazioni tra loro. L'agente manifesta la sua autonomia modificando il proprio stato interno e compiendo azioni come risultato del "meta-ragionamento" associato alla gerarchia delle componenti.
Anche in questo caso come in Concurrent MetateMl'autonomia non dipende da una particolare caratteristica di DESIRE, ma da come vengono definte le componenti, dai ruoli a loro associati e dalle relazioni tra le componenti.
Il "meta-ragionamento" viene inoltre utilizzato per realizzare il meccanismo di revisione dei compiti che l'agente deve portare a termine: l'agente è in grado di percepire il fallimento della strategia che sta utilizzando e quindi modificarla.

L'agente ConGolog è descritto da un programma ed il suo comportamento è caratterizzato dalle istruzioni e dalle chiamate di procedura di cui è costituito.
Lo stato interno dell'agente è determinato dallo stato di esecuzione della procedura e la capacità di controllo su questo e sulle azioni che l'agente può eseguire è determinato dai costrutti sintattici (IF-THEN-ELSE, WHILE, esecuzione concorrente, scelta e iterazione non deterministica di istruzioni) che controllano l'esecuzione della procedura.

Il linguaggio AGENT 0 descrive il comportamento di un agente attraverso credenze, obbligazioni e capacità di compiere azioni.
Gli agenti realizzati con AGENT 0 sono in grado di operare senza l'intervento di entità esterne sfruttando le proprie obbligazioni e credenze iniziali che, nel caso di obbligazioni verso l'agente stesso, corrispondono a decisioni prese dall'agente.
Sono in grado di eseguire azioni, ricevere oppure inviare autonomamente messaggi ad altri agenti. L'agente è in grado di eliminare autonomamente, sia le obbligazioni intraprese verso se stesso sia verso altri agenti, modificando in questo modo il proprio modo di agire ed influenzando quello degli altri.

L'agente specificato dal linguaggio descritto nella sezione 2.7 non è in grado di scambiare messaggi con nessuna altra entità e non è in grado di percepire eventi dall'ambiente circostante. La sua attività comincia con l'esecuzione di uno degli obiettivi appartenenti all'insieme iniziale, l'obiettivo selezionato comporta l'esecuzione di un piano dell'agente e l'esecuzione delle azioni che costituiscono il piano è in grado di modificare lo stato mentale dell'agente, aggiungendo e/o rimuovendo

credenze e portando a termine l'obiettivo selezionato. L'agente pertanto è autonomo in quanto agisce senza l'intervento esterno da parte di altre entità.
Il linguaggio è dotato di un meccanismo di revisione dell'esecuzione degli obiettivi basato sulle regole di ragionamento pratico: il comportamento dell'agente in relazione all'esecuzione di un obiettivo può essere modificato in funzione della rappresentazione che l'agente ha dell'ambiente in cui è collocato.

Nel linguaggio descritto nella sezione 2.8 il controllo sullo stato interno dell'agente e l'esecuzione delle azioni sono modellati da un motore computazionale e da una componente per il trattamento delle credenze/conoscenze, che in [BEL95] non vengono descritti. L'attenzione viene infatti focalizzata sulla rappresentazione delle credenze, delle conoscenze e sui problemi di comunicazione all'interno del sistema multi-agente.

I linguaggi dMARS, AgentSpeak(L) e quello descritto nella sezione 2.9 specificano agenti caratterizzati dall'architettura BDI, cioè agenti il cui stato interno è descritto dalle componenti che rappresentano credenze, desideri, intenzioni e la cui conoscenza procedurale è costituita da librerie di piani.

Per quanto riguarda dMARS l'autonomia è legata all'inizializzazione delle credenze e della coda degli eventi: in questo modo con l'attivazione degli agenti vengono generate le intenzioni iniziali e quindi avviata l'attività dell'agente. Una volta avviato, l'agente è in grado di compiere azioni che modificano lo stato interno, aggiungendo e/o rimuovendo credenze, oppure generando nuovi desideri e/o completando quelli giá esistenti. Le azioni eseguite possono inoltre modificare l'ambiente in cui l'agente è collocato ed effettuare eventuali comunicazioni con altri agenti.
L'agente dMARS revisiona l'esecuzione delle intenzioni percependo il fallimento di una istanza di piano all'interno di una intenzione e causando la selezione di un piano alternativo. L'intenzione fallisce solo quando non è più possibile selezionare piani alternativi.

L'attività degli agenti AgentSpeak(L) comincia, come per gli agenti dMARS, processando gli eventi iniziali oppure con la percezione dell'ambiente; gli eventi percepiti vengono inseriti nell'insieme degli eventi e quindi processati.
Gli elementi che appartengono all'insieme degli eventi, se in accordo con le credenze dell'agente, possono attivare piani la cui esecuzione modifica lo stato interno dell'agente in modo simile a quello di dMARS, cioè modificando credenze, intenzioni ed eventualmente eseguendo azioni il cui effetto si manifesta sull'ambiente.

Anche nel linguaggio descritto nella sezione 2.9 l'esecuzione dei piani è causata da eventi e le azioni eseguite sono in grado di modificare sia lo stato interno dell'agente che lo stato dell'ambiente. Il modo in cui l'agente manifesta il controllo sul proprio stato interno, quindi sulle azioni che lo modificano dipende essenzialmente dall'architettura utilizzata, quindi dalla definizione dei modelli delle credenze, degli obiettivi e dei piani.

2.10.2 Reattività

Consiste nella capacità di percepire e quindi di reagire agli eventi che si manifestano nell'ambiente. In funzione degli eventi percepiti l'agente è in grado di modificare il proprio comportamento e quindi di aggiornare il proprio stato interno e compiere azioni adeguate.

In Concurrent MetateM gli eventi sono percepiti dall'agente come messaggi inviati dall'ambiente.
La ricezione dei messaggi avviene tramite i predicati ambiente: un predicato ambiente è vero se e solo se è stato ricevuto il messaggio corrispondente. Ad esempio l'agente BDI della sezione 2.4.3 percepisce l'ambiente sfruttando il predicato ambiente *percept*(p): il predicato *percept* diventa vero solo dopo la ricezione del messaggio p. La capacità di reagire ad un evento è realizzata sfruttando le regole MetateM, che hanno la forma seguente:

$$p \Rightarrow f$$

dove p è un predicato ambiente, e f una formula temporale che rappresenta l'azione da intraprendere come conseguenza dell'evento manifestatosi. Nell'esempio della sezione 2.4.3 l'agente reagisce ad una percezione con l'aggiornamento delle proprie credenze.

Anche in DESIRE la percezione degli eventi avviene tramite la ricezione di messaggi inviati dall'ambiente.
Il messaggio ricevuto viene processato all'interno della componente ed il risultato dell'elaborazione causa l'esecuzione di una azione oppure l'aggiornamento dello stato interno oppure la comunicazione di una informazione ad un'altra componente o ad un altro agente.

In ConGolog l'ambiente è modellato da un programma eseguito concorrentemente rispetto agli agenti del sistema. Gli eventi che si manifestano nell'ambiente sono rappresentati dalle azioni eseguite dal programma, azioni in grado di modificare lo stato del "mondo", rappresentato dal concetto di *situazione* del *Situation Calculus*.
L'agente ConGolog esegue azioni in accordo con le relative precondizioni, le quali

dipendono dallo stato (*situazione*) del "mondo". Il manifestarsi di un evento può quindi rendere possibile l'esecuzione di una azione arbitrariamente complessa.

AGENT 0 non ha costrutti per la percezione di eventi e specifica entità il cui comportamento è descritto da qualità mentali; l'ambiente viene quindi simulato da un agente il cui comportamento è descritto da qualità mentali.
Gli eventi dell'ambiente vengono simulati sotto forma di messaggi, i quali, insieme allo stato mentale dell'agente causano l'esecuzione di obbligazioni e quindi la reazione all'evento.

Il linguaggio della sezione 2.7 non ha costrutti per la percezione degli eventi in quanto l'ambiente viene completamente rappresentato nell'insieme delle credenze dell'agente. Gli agenti specificati con questo linguaggio non hanno capacità reattive.

Nell'articolo descritto nella sezione 2.8 vengono enfatizzati gli aspetti relativi alla rappresentazione della conoscenza e alla comunicazione, ma non quello relativo all'elaborazione della conoscenza, quindi relativo alla reattività e proattività.

dMARS e AgentSpeak(L) percepiscono gli eventi prodotti dall'ambiente memorizzandoli in apposite strutture dati: una coda per gli agenti dMARS, un insieme per gli agenti AgentSpeak(L). Gli agenti manifestano la reazione agli eventi percepiti in quanto tali eventi sono responsabili della selezione dei piani e quindi dell'esecuzione delle azioni.

Nel linguaggio trattato nella sezione 2.9 la percezione degli eventi esterni è realizzata da funzioni valutabili: funzioni in grado di acquisire lo stato degli oggetti che costituiscono l'ambiente. La percezione degli eventi, in accordo con le credenze dell'agente, causa l'esecuzione dei piani e quindi il comportamento reattivo.

2.10.3 Proattività

Consiste nella capacità di assumere un comportamento che dipende dall'obiettivo che l'agente deve perseguire: l'agente compie azioni e modifica il proprio stato interno non solo in funzione degli eventi che percepisce, ma al fine di raggiungere i propri obiettivi.
L'agente può avere una rappresentazione esplicita dell'obiettivo, che viene utilizzata per "guidarne" il comportamento; se così non è il comportamento proattivo "emerge" dall'interazione tra agente e le altre entità collocate nell'ambiente.

La capacità di essere proattivo dipende dall'architettura dell'agente. Linguaggi come Concurrent MetateM, DESIRE, ConGolog sono basati su formalismi abbastanza astratti da poter specificare agenti con architetture arbitrarie che possono essere proattivi oppure no.

AGENT 0 ha la possibilità di rappresentare gli obiettivi da completare come obbligazioni che possono essere richieste da altri agenti, oppure definite come obbligazioni iniziali. Le obbligazioni vengono eseguite in accordo con lo stato mentale dell'agente e la loro esecuzione ne caratterizza il comportamento.

Il linguaggio descritto nella sezione 2.7 rappresenta esplicitamente gli obiettivi dell'agente. Possono essere rappresentati due tipi di obiettivi (sezione 2.7.1): perseguire il primo tipo significa ottenere uno stato mentale in cui una data condizione viene soddisfatta; il secondo tipo consiste in un test il cui risultato è "vero" oppure "falso".
Gli obiettivi vengono raggiunti utilizzando le "regole di ragionamento pratico": procedure selezionate dall'obiettivo che l'agente vuole completare in accordo con le credenze dell'agente; il corpo delle procedure è costituito da un programma in grado di eseguire azioni primitive oppure altri obiettivi.
Utilizzando gli obiettivi e le "regole di ragionamento pratico" l'agente compie azioni che modificano il suo stato interno assumendo così un comportamento puramente proattivo.

Anche i linguaggi dMARS, AgentSpeak(L) e quello specificato nella sezione 2.9 hanno una rappresentazione esplicita degli obiettivi che l'agente deve perseguire, ma in questo caso il comportamento dell'agente è il risultato sia dell'esecuzione degli obiettivi con cui vengono inizializzati (capacità proattiva), sia dell'esecuzione degli obiettivi generati dagli eventi percepiti dall'ambiente (capacità reattiva); in questo senso l'agente ha un comportamento "ibrido", cioè sia proattivo che reattivo.
Gli agenti dMARS e AgentSpeak(L) hanno due tipi di obiettivi (sezioni 2.2.1 e 2.3.1) simili a quelli del linguaggio descritto nella sezione 2.7: un obiettivo del primo tipo viene soddisfatto se tra le conoscenze procedurali dell'agente vi è un piano la cui esecuzione ha come risultato una sequenza di azioni in grado di soddisfare una data condizione; un obiettivo del secondo tipo viene soddisfatto se l'obiettivo è conseguenza logica delle credenze, mentre se non lo è fallisce.
Gli agenti specificati dal linguaggio descritto nella sezione 2.9 hanno tre tipi di obiettivi (sezione 2.9.2): il primo tipo viene completato con successo se la condizione di cui è costituito è creduta vera dall'agente, altrimenti viene selezionato un piano, il cui successo comporta il successo dell'obiettivo, se no il fallimento; un obiettivo appartenente al secondo tipo viene completato con successo se la condizione è creduta vera dall'agente, fallisce se è creduta falsa. Nel caso in cui

risulti indeterminata viene selezionato un piano il cui successo comporta il successo dell'obiettivo, altrimenti il fallimento; un obiettivo appartenente al terzo tipo viene completato con successo se la condizione è deteminata (sia vera che falsa), viene invocato un piano nel caso non lo sia. Se il piano termina con successo anche l'obiettivo ha successo, se l'esecuzione del piano fallisce la condizione rimane indeterminata e l'obiettivo fallisce.

2.10.4 Capacità sociale

Gli agenti appartenenti ad un sistema multi-agente hanno capacità sociale se sono in grado di comunicare tra loro.
Un linguaggio di specifica dovrebbe possedere costrutti per la descrizione di un sistema multi-agente e quindi per la comunicazione: l'agente deve poter eseguire azioni comunicative per l'invio dei messaggi ed avere la capacità di riceverne.

Il linguaggio Concurrent MetateM è in grado di descrivere sistemi in cui gli agenti sono in grado di scambiarsi messaggi utilizzando il protocollo di comunicazione broadcasting.
L'agente è in grado di comunicare utilizzando una componente interfaccia che contiene informazioni riguardanti il nome dell'agente, i messaggi che può inviare agli altri agenti del sistema e quelli che può ricevere. Il messaggio viene inviato in seguito al soddisfacimento di un predicato componente, mentre la ricezione causa il soddisfacimento di un predicato ambiente.

Come in Concurrent MetateM, gli agenti DESIRE hanno un'interfaccia che specifica i messaggi che l'agente può inviare e quelli che può ricevere. Gli agenti DESIRE non possono scegliere arbitrariamente con quale agente comunicare, in quanto la comunicazione avviene tramite canali chiamati "collegamenti di informazione", che "collegano" staticamente l'interfaccia di output del mittente con quella di input del destinatario. L'attività dei "collegamenti di informazione" è caratterizzata da tre stati che rappresentano l'attesa del passaggio di informazione, il passaggio dell'informazione, la non utilizzazione del canale.

In ConGolog la comunicazione avviene attraverso la definizione di azioni primitive specializzate nella comunicazione. Le azioni che un agente ConGolog può eseguire sono specificate utilizzando precondizioni e postcondizioni la cui definizione è del tutto arbitraria; in questo modo ogni agente può avere operazioni comunicative e protocolli di comunicazione personali.
Nell'esempio della sezione 2.6.5 ogni agente ha una "coda dei messaggi" su cui operano azioni comunicative il cui effetto è quello di aggiungere un messaggio (nella

coda del destinatario), leggerlo oppure rimuoverlo.

Le comunicazioni tra agenti AGENT 0 avvengono utilizzando le tre azioni INFORM, REQUEST ed UNREQUEST, le quali hanno l'effetto rispettivamente di informare il desinatario di un fatto, chiedere al destinatario di eseguire una azione, ritirare una richiesta precedentemente eseguita. La ricezione dei messaggi ha come effetto l'aggiornamento delle credenze dell'agente se il messaggio è del tipo INFORM, l'aggiornamento delle obbligazioni se è del tipo REQUEST oppure UNREQUEST.

Nel linguaggio descritto nella sezione 2.7 non sono realizzati costrutti per la comunicazione: l'ambiente viene completamente rappresentato nelle credenze dell'agente e quindi non può essere condiviso da altri. Gli agenti non hanno capacità sociale.

Gli agenti descritti nel linguaggio della sezione 2.8 condividono un linguaggio che permette lo scambio di informazioni.
Gli agenti inviano messaggi formati da: una costante che determina il tipo di messaggio, una intestazione che contiene informazioni sul mittente, il contenuto del messaggio rappresentato da una espressione del linguaggio per la comunicazione condiviso dagli agenti. Quando un messaggio viene ricevuto, il contenuto viene interpretato in una espressione del linguaggio di rappresentazione della conoscenza locale all'agente.

Negli articoli citati per la descrizione dei linguaggi dMARS, AgentSpeak(L) e quello della sezione 2.9 non viene affrontato il problema della comunicazione, anche se è intuibile che ogni linguaggio possa essere esteso con costrutti per l'invio e la ricezione dei messaggi.
L'articolo citato nella sezione 2.2 ([dKLW98]) non affronta il problema della comunicazione tra agenti dMARS, fa solo riferimento all'esistenza di un evento trigger generato dalla ricezione di un messaggio.
La struttura di dMARS sembra permettere la comunicazione tra agenti attraverso l'uso di azioni esterne specializzate, le quali possono causare nell'agente destinatario un evento in grado di attivare un piano mper gestire il messaggio ricevuto.
A differenza di dMARS nel linguaggio AgentSpeak(L) non compare esplicitamente la possibilità dell'invocazione di un piano in seguito alla ricezione di un messaggio: né la semantica operazionale né la teoria per la dimostrazione delle proprietà comportamentali dell'agente trattano l'invio e la ricezione dei messaggi.
Nel linguaggio descritto nella sezione 2.9 la comunicazione potrebbe essere effettuata da azioni specializzate il cui effetto sarebbe quello di causare un evento in grado di invocare un piano dell'agente.

2.10.5 Flessibilità dei linguaggi trattati

Un linguaggio di specifica oltre a descrivere le caratteristiche di autonomia, reattività, proattività e capacità sociale dovrebbe essere abbastanza flessibile da poter specificare agenti con architetture eterogenee ed avere una semantica che descriva in modo significativo il comportamento del sistema affinché sia realizzato correttamente.
Inoltre se il linguaggio di specifica fosse eseguibile o animabile, si otterrebbero notevoli vantaggi in quanto si potrebbe simulare l'esecuzione del sistema.

Linguaggi come dMARS, AgentSpeak(L) e la metodologia di specifica descritta nella sezione 2.9 descrivono agenti basati sulla architettura BDI, non sono quindi in grado di descrivere agenti con architetture arbitrarie costringendo il loro utilzzo alla specifica di un numero limitato di applicazioni; d'altra parte hanno una semantica che caratterizza pienamente il comportamento dell'agente rendendo più semplice la realizzazione corretta dell'agente e fornendo un meccanismo per la verifica delle sue proprietà.

AGENT 0 ed i linguaggi descritti nelle sezioni 2.7 e 2.8, pur essendo costituiti da formalismi basati su qualità mentali, sono in grado di specificare agenti con architetture diverse da quella BDI, ma sempre architetture vincolate dalla semantica delle qualità mentali su cui sono basati i linguaggi. Come nel caso precedente l'utilizzo di qualitàmentali favorisce la descrizione del comportamento dell'agente e facilita la specifica dell'applicazione.

I linguaggi Concurrent MetateM, DESIRE, ConGolog, sono basati su formalismi che permettono la realizzazione di agenti con architeture arbitrarie ma la cui semantica non caratterizza direttamente il comportamento dell'agente rendendo complicata la fase di specifica e di verifica del sistema multi-agente. Inoltre Concurrent MetateMe DESIREnon hanno una struttura che permette l'estensione incrementale del sistema: aggiungere uno o più agenti può comportare la modifica delle interfacce per la comunicazione degli agenti già presenti.

Parte II

Un nuovo linguaggio di specifica per sistemi multi-agente

Capitolo 3

Concetti di base del modello proposto

Nel seguito della tesi considereremo un agente come una entità caratterizzata da

- uno **stato**;
- un **programma**;
- un **engine**.

Lo *stato* contiene le informazioni che, durante l'esecuzione dell'agente, possono subire modifiche. Ad esempio lo stato di un agente che percepisce e gestisce eventi inserendoli e prelevandoli in/da una *event queue*, e le cui informazioni sul mondo sono rappresentate da *credenze*, conterrà la *event queue* e le *credenze*.
Il *programma* contiene le informazioni che non cambiano durante l'esecuzione. per il semplice agente descritto sopra, il *programma* potrebbe essere costituito da un insieme di regole che descrivono quali azioni eseguire in funzione di un evento e delle credenze correnti. Le azioni che un agente può eseguire sono di svariata natura, e potrebbero comprendere ad esempio aggiornamenti dello stato, azioni sull'ambiente e comunicazioni con altri agenti.
Infine l'*engine* governa l'esecuzione dell'agente. Sempre riferendosi al caso precedente, un tipico *engine* potrebbe prevedere l'estrazione di un evento da una coda, la scelta di una regola adatta a gestire l'evento ed applicabile nello stato corrente, ed infine l'esecuzione di tutte le azioni indicate nella regola.
Va notato come l'engine ed il programma si collochino a due livelli di astrazione diversi: l'engine rappresenta infatti un meta-interprete per il programma e per i dati (stato) su cui il programma opera.

La applicazione delle istruzioni dell'engine allo stato ed al programma dell'agente ne determinano il *comportamento*.

La **architettura** di un agente deve consentirne la rappresentazione nei termini sopra descritti. Essa sarà quindi caratterizzata da componenti che contengono lo stato, componenti che contengono il programma, e da un engine che opera su queste componenti.
Due agenti possono avere le stesse componenti, contenenti gli stessi dati, ma engines diversi, oppure possono avere lo stesso engine e le stesse componenti, contenenti però dati diversi. In entrambi i casi il comportamento degli agenti risulterà in generale differente.

L'obiettivo di questa tesi è lo sviluppo di un linguaggio in grado di descrivere sistemi multi-agente eterogenei permettendo la definizione delle architetture degli agenti, ovvero delle loro componenti e del loro engine. Il contentuto delle componenti, caratterizzante lo stato ed il programma degli agenti, sarà espresso in un linguaggio "oggetto" dipendente dalla architettura; l'engine fornirà un interprete per tale linguaggio.

Al fine di permettere una maggiore modularità e flessibilità nella definizione di agenti eterogenei introduciamo una gerarchia di livelli di astrazione, illustrata in figura 3.1.

- **Architettura astratta**. La architettura astratta definisce le componenti e la struttura di base dell'engine. Per ogni "macro-istruzione" presente nell'engine è possibile fornire indicazioni sui possibili modi per realizzarla, senza necessariamente arrivare ad un livello di dettaglio paragonabile ad una implementazione.
  Riprendendo ancora il semplice esempio descritto sopra, una architettura astratta per l'agente dovrebbe prevedere

  – una componente per contenere le *credenze*;

  – una componente per la *event queue*;

  – una componente per contenere le *regole* che definiscono il programma.

  L'engine potrebbe essere quello già descritto, composto dalle seguenti "macro-istruzioni"

  1. estrai un evento dalla struttura *event queue*;
  2. scegli una regola per gestire l'evento, in base anche alle credenze correnti;
  3. esegui tutte le azioni indicate dalla regola.

La "macro-istruzione" dell'engine *scegli una regola per gestire l'evento* potrebbe essere realizzata in modi diversi:

- 1. scegli a caso una regola;
  2. verifica che la testa della regola sia uguale all'evento;
  3. verifica che le condizioni sulle credenze espresse nella regola siano soddisfatte;
  4. se entrambe le verifiche hanno successo restituisci la regola scelta, altrimenti ritorna al punto 1.
- 1. costruisci l'insieme di tutte le regole applicabili;
  2. restituisci la regola che, tra queste, prevede il minor numero di azioni da eseguire.

A sua volta *costruisci l'insieme di tutte le regole applicabili* potrebbe espandersi in altre istruzioni ad alto livello: non si pongono limiti sul livello di annidamento delle definizioni delle "macro-istruzioni".

- **Architettura concreta**. Una architettura concreta si definisce a partire da una architettura astratta nel seguente modo:

  - ad ogni componente si associa un tipo tra quelli previsti dal linguaggio;
  - per ogni istruzione ad alto livello contenuta nell'engine si sceglie una delle possibili indicazioni su come realizzarla (se tali indicazioni sono fornite) e se ne dà una implementazione concreta.

  Ad esempio, si potrebbe stabilire che le credenze di un agente sono contenute in una struttura dati di tipo insieme, mentre le regole sono contenute in una lista. La event queue sarà ragionevolmente implementata con una coda. Inoltre si dovrebbe scegliere che interpretazione dare, tra quelle proposte, alla istruzione *scegli una regola*, e si dovrebbe infine fornirne una implementazione coerente con le indicazioni fornite.

- **Classe di agente**. Una classe si definisce a partire da una architettura concreta istanziando le componenti contenenti il *programma* dell'agente. Vedremo nel seguito esempi articolato che chiariranno il concetto di *classe di agenti*.

- **Istanza di agente**. A partire da una classe, si completa la istanziazione delle componenti della architettura definendo il contenuto iniziale delle componenti che contengono lo stato.

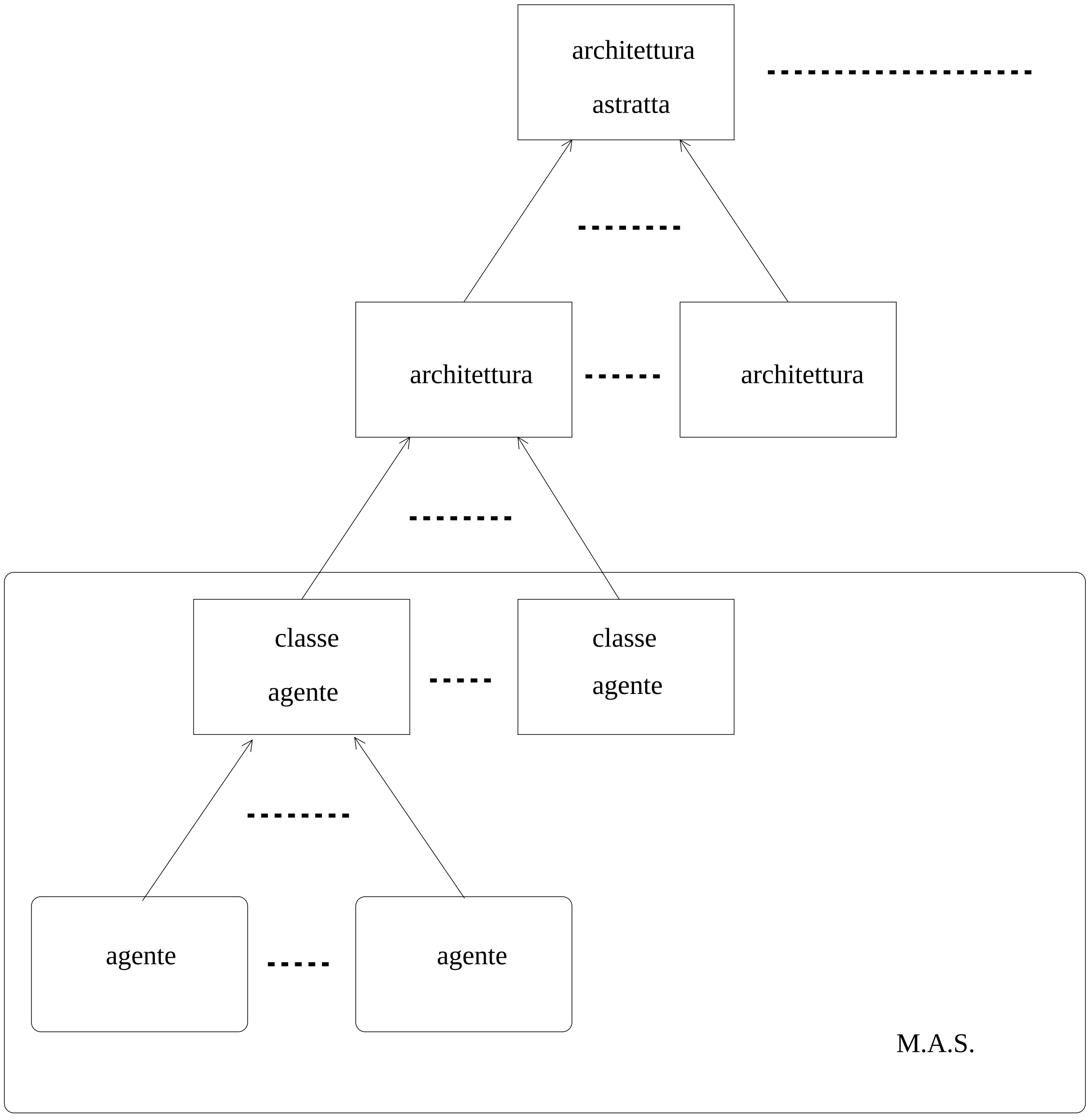


Figura 3.1: Struttura di un MAS.

Per illustrare meglio le potenzialità della gerarchia introdotta consideriamo uno scenario in cui sono presenti due agenti con architettura BDI (vedi sezione 1.2.1). Il primo agente esegue la diagnosi dei guasti di un particolare sistema, il secondo è un robot che si muove all'interno di un ambiente con il fine di raccogliere oggetti.
L'architettura del primo agente è costituita da un engine simile al ciclo descritto nella figura 1.1 e da tre componenti: la prima componente mantiene le credenze riguardanti il funzionamento del sistema che deve essere diagnosticato; la seconda mantiene la conoscenza procedurale dell'agente costituita dai piani che l'agente può intraprendere quando l'evento percepito denota un guasto; infine, la terza componente mantiene le intenzioni dell'agente, cioè i piani scelti per essere eseguiti. La prima e la terza componente contengono pertanto lo *stato* dell'agente, mentre la seconda contiene il *programma*.
Anche l'architettura del secondo agente è costituita da un engine come quello descritto nella figura 1.1 ed anch'essa è formata da tre componenti: la prima mantiene la conoscenza dell'ambiente in cui è collocato il robot, la seconda i piani che vengono attivati in funzione degli eventi percepiti e la terza le intenzioni dell'agente.

Il comportamento dei due agenti dell'esempio differisce a causa dell'informazione contenuta nelle loro componenti: nel primo agente la conoscenza riguardante l'ambiente si riferisce al funzionamento del sistema, mentre la conoscenza procedurale consiste in piani il cui scopo è diagnosticare guasti per quel sistema. Se inizialmente la terza componente contiene delle intenzioni, esse vengono eseguite all'attivazione dell'agente. Nel secondo agente la conoscenza riguardante l'ambiente si riferisce ad uno scenario composto da "strade" nelle quali il robot può muoversi, da ostacoli che deve evitare mentre percorre le strade e da oggetti che può raccogliere. La conoscenza procedurale consiste in un insieme di piani il cui scopo è quello di scegliere una strada, evitare ostacoli e raccogliere oggetti. Anche in questo caso l'inizializzazione della terza componente determina il comportamento iniziale dell'agente.

Le componenti dell'architettura corrispondono alle strutture dati dell'agente. Quando le componenti vengono implementate bisogna assegnare loro un tipo, al fine di poterle accedere e manipolare utilizzando le istruzioni appropriate. Ad esempio la componente che mantiene le intenzioni del robot può essere implementata come uno stack, mentre quella dell'agente per la diagnosi può essere una lista. Nelle architetture concrete che implementano una architettura astratta, una stessa componente può essere realizzata utilizzando tipi diversi. Questo non "snatura" la architettura degli agenti, che a livello astratto rimane invariata.

L'engine dell'architettura è un programma composto da istruzioni che agiscono sulle componenti. Facendo riferimento all'esempio precedente, l'attivazione di un piano è subordinata alla sua scelta all'interno della componente "conoscenza procedurale". Il modo in cui viene effettuata questa scelta è codificato nell'engine della architettura concreta, e può differire da una architettura concreta ad un'altra. Ad esempio, il robot può scegliere i piani in modo diverso dall'agente per la diagnosi dei guasti: il robot sceglie il primo piano che può essere utilizzato, mentre l'altro agente sceglie il migliore, rispetto ad un certo criterio, tra tutti quelli che possono essere utilizzati. Questo comporta una diversa definizione della procedura che descrive la scelta di un piano all'interno della componente "conoscenza procedurale". Anche in questo caso l'architettura astratta degli agenti è la stessa nonostante una procedura dell'engine abbia definizioni differenti.

La definizione di un sistema multi-agente deve quindi prevedere la descrizione delle architetture astratte, delle architetture concrete, delle classi di agenti e delle istanze di queste classi. Ciò che deve essere descritto è schematicamente rappresentato nella figura 3.2. Quanto descritto fino ad ora riguarda la rappresentazione e la realizzazione di ciò che sta all'interno di un agente. Abbiamo però osservato che un agente, per essere tale, deve poter compiere delle azioni che lo mettono in relazione con l'ambiente e con gli altri agenti nel sistema. Abbiamo pertanto la necessità di rappresentare l'ambiente in cui gli agenti sono collocati e di consentire loro di percepirlo e modificarlo. Dobbiamo inoltre fornire agli agenti uno strumento per comunicare.

La soluzione adottata prevede l'esistenza di un **ambiente (ENV)** che contiene la rappresentazione di tutti i fatti dell'ambiente fisico che sono rilevanti per il particolare dominio applicativo. Gli agenti possono percepire l'ambiente direttamente, ma lo possono modificare soltanto interagendo con un "agente ambiente". Si supponga ad esempio che l'agente robot con architettura BDI dell'esempio possa eseguire una azione primitiva `accendi_il_motore`. Anche se l'agente robot non lo può sapere, gli effetti di questa azione sull'ambiente sono molteplici: aumenteranno ad esempio il rumore, la temperatura ed il tasso di inquinamento. Non sarebbe quindi realistico consentire all'agente robot di modificare direttamente l'ambiente aggiungendovi solo l'informazione che il suo motore è stato acceso: ci sono conseguenze di questa azione che non sta al robot calcolare, ma che possono essere rilevanti nel dominio applicativo considerato.
Per questa ragione si introduce l'"agente ambiente", la cui definizione è dipendente dal dominio applicativo ed il cui compito è quello di calcolare tutte le conseguenze delle azioni primitive che gli agenti eseguono e di aggiornare l'ambiente di conseguenza. Le azioni primitive sull'ambiente che un agente può compiere sono tradotte in messaggi all'"agente ambiente" che sa come gestirle e come modificare

PARTE INDIPENDENTE DAL DOMINIO APPLICATIVO

abstract_architecture($abs\_arch\_name_1$)
{ *definizione della architettura astratta* $abs\_arch\_name_1$*: si definiscono le componenti della architettura, si definisce un engine astratto composto da macro-istruzioni, eventualmente si forniscono delle indicazioni su come realizzare alcune macro-istruzioni* }
$\vdots$
abstract_architecture($abs\_arch\_name_l$)
{ *definizione della architettura astratta* $abs\_arch\_name_l$ }

architecture($arch\_name_1$) is_a ($abs\_arch\_name_i$)
{ *implementazione della architettura concreta* $arch\_name_1$*: si assegna un tipo alle componenti della architettura astratta, si sceglie quali indicazioni seguire (se ne vengono date) per la realizzazione delle macro-istruzioni dell'engine astratto, si implementano tali macro-istruzioni* }
$\vdots$
architecture($arch\_name_m$) is_a ($abs\_arch\_name_j$)
{ *implementazione della architettura concreta* $arch\_name_j$ }

PARTE DIPENDENTE DAL DOMINIO APPLICATIVO

class_agent($arch\_name_k$, $agent\_class\_name_1$)
{ *istanziazione delle componenti "programma"* }
$\vdots$
class_agent($arch\_name_h$, $agent\_class\_name_n$)
{ *istanziazione delle componenti "programma"* }

agent($agent\_class\_name_x, agent\_name_1$)
{ *istanziazione delle componenti "stato"* }
$\vdots$
agent($agent\_class\_name_y, agent\_name_p$)
{ *istanziazione delle componenti "stato"* }

Figura 3.2: Descrizione di un sistema multi-agente.

ENV coerentemente.

Passando infine alla comunicazione, gli agenti comunicano con l'"agente ambiente" e tra loro tramite l'**etere (ETH)**, una struttura dati nella/dalla quale vengono scritti/letti i messaggi. Il linguaggio per la implementazione concreta dell'engine di un agente fornisce primitive per la comunicazione di tipo `send`, per mettere un messaggio nell'etere, e `receive` per leggere un messaggio dall'etere, nonché varianti di queste primitive di base.

I concetti descritti in questa introduzione verranno ripresi, espansi ed approfonditi nei successivi capitoli. In particolare il capitolo 3 descrive la sintassi del linguaggio per definire le architetture astratte, concrete, le classi di agente, le loro istanze ed i MAS; il capitolo 4 fornisce la semantica di questo linguaggio; il capitolo 5 riporta alcune considerazioni conclusive e possibili sviluppi del lavoro.

Capitolo 4

Sintassi del linguaggio

In questo capitolo viene definita la sintassi del linguaggio per definire le architetture astratte, le architetture concrete, le classi di agente e le loro istanze.
Per poter definire queste categorie sintattiche bisogna prima definire:

- l'insieme dei tipi $\mathcal{T}$supportati dal linguaggio;
- il linguaggio delle espressioni;
- l'insieme delle istruzioni per l'accesso e la modifica delle componenti della architettura;
- l'insieme delle istruzioni per la ricezione e l'invio di messaggi;
- l'insieme delle istruzioni per la percezione e la generazione di eventi nell'ambiente.

4.1 L'insieme dei tipi e delle espressioni tipate

Le strutture dati che vengono utilizzate all'interno dell'architettura concreta sono caratterizzate da un tipo: ad esempio una componente può essere strutturata come una lista oppure come uno stack, ed anche le variabili dell'engine possono essere istanziate con espressioni tipate.
Per rendere il linguaggio in grado di specificare applicazioni reali viene definito un insieme minimale di tipi $\mathcal{T}$. A questo insieme appartengono i tipi base che corrispondono agli interi, ai reali, ai booleani ed alle stringhe, denotati rispettivamente da **int**, **real**, **boolean** e **string**, ed alcuni tipi strutturati quali liste, stack, code insiemi e tuple, denotati rispettivamente da **list**, **stack**, **queue**, **set** e **tupla**. Per i tipi strutturati il linguaggio definisce esplicitamente quali sono i costruttori e le operazioni di accesso e di modifica, mentre per i tipi base si assume di disporre

delle operazioni “canoniche”.
L’insieme minimale dei tipi è quindi

$$\mathcal{T} = \{\textbf{int}, \textbf{real}, \textbf{boolean}, \textbf{string}, \textbf{list}, \textbf{stack}, \textbf{queue}, \textbf{set}, \textbf{tupla}\}$$

4.1.1 Espressioni tipate

Gli ingredienti per costruire espressioni tipate sono, per ogni tipo

- un insieme non vuoto di costanti,
- un insieme di operazioni che operano su argomenti eterogenei e che restituiscono risultati del tipo in questione.

La sintassi rigorosa del linguaggio per le espressioni tipate è descritta da regole BNF riportate nell’Appendice.

Nel seguito si fornisce una trattazione informale delle operazioni disponibili per i vari tipi strutturati e del loro significato. Le operazioni attinenti ad un certo tipo di dato vengono descritte tutte insieme, anche se il tipo delle espressioni contenenti quelle operazioni può non essere omogeneo. Si pensi ad esempio alla operazione di estrazione dell’elemento in testa ad una lista l di interi. Il tipo dell’espressione **gethlist**(l) è **int** e non **list**, tuttavia l’operazione **gethlist**(l) è caratteristica delle liste e viene pertanto descritta nel relativo paragrafo.

Operazioni sulle liste

- **e_l**: elemento costante corrispondente alla lista vuota.
- **inshlist**(l, e): inserimento in testa alla lista l dell’elemento e. Il risultato dell’operazione è una nuova lista uguale ad l con e in testa. l non viene modificata.
- **instlist**(l, e): inserimento in coda alla lista l dell’elemento e. Valgono considerazioni analoghe a quelle per l’operazione precedente.
- **gethlist**(l): recupero dell’elemento in testa alla lista l. Il tipo del risultato è il tipo dell’elemento estratto. La lista l non viene modificata.
- **gettlist**(l): recupero dell’elemento in coda alla lista l. Come per l’operazione precedente, il tipo del risultato dipende dall’elemento estratto e la lista l non viene modificata.
- **delhlist**(l): rimozione dell’elemento in testa alla lista l. Il risultato dell’operazione è una nuova lista uguale ad l privata dell’elemento in testa.

- **deltlist**(l): rimozione dell'elemento in coda alla lista l. Il risultato dell'operazione è uguale alla lista l privata dell'elemento in coda.

- **append**(l_1, l_2): concatenazione di liste. Il risultato è una lista contenente gli elementi della prima lista seguiti (in coda) dagli elementi della seconda lista.

Operazioni sugli stack

- **e_s**: elemento costante che rappresenta lo stack vuoto.

- **put_stack**(s, e): inserimento dell'elemento e nello stack s. Il risultato è uguale allo stack s sulla cui cima è stato aggiunto l'elemento e.

- **top_stack**(s): recupero dell'elemento in cima allo stack. Il tipo del risultato dipende dal tipo dell'elemento estratto. Lo stack s non viene modificato.

- **pop_stack**(s): rimozione della testa dello stack s. Il risultato è uno stack uguale ad s ma privato dell'elemento in cima. s non viene modificato.

Operazioni sulle code

- **e_q**: elemento costante corrispondente alla coda vuota.

- **insqueue**(c, e): inserimento dell'elemento e nella coda c. Il risultato è una nuova coda; c non viene modificata.

- **getqueue**(c): recupero di un elemento dalla coda c. Il tipo del risultato dipende dall'elemento estratto e la coda c non viene modificata da questa operazione. Rispetto alla operazione di estrazione sugli stack, che estrae l'ultimo elemento inserito, l'operazione sulle code estrae il primo elemento inserito.

- **delqueue**(c): rimozione di un elemento dalla coda. Il risultato dell'operazione è una nuova coda uguale alla coda c privata dell'elemento inserito per primo. La coda c non viene modificata.

Operazioni sugli insiemi

- $\emptyset$: costante corrispondente all'insieme vuoto.

- **insert**(s, e): inserimento dell'elemento e nell'insieme s. Il risultato dell'operazione è un nuovo insieme contenente tutti gli elementi dell'insieme s ed anche l'elemento e (se non c'era giá). s non viene modificato.

- **getset**(s): estrazione non deterministica di un elemento dall'insieme s. Il tipo del risultato dipende dal tipo degli elementi di s, che non viene modificato dall'operazione.

- **removeset**(s, e): rimozione dell'elemento e dall'insieme s. Il risultato della operazione è un insieme contenente tutti gli elementi di s escluso e. Se e non era presente in s, il risultato coinciderà con s. In ogni caso s non viene modificato.

Operazioni sulle tuple

- **e_t**(n): costante corrispondente alla tupla vuota di dimensione n. Il fatto che la tupla sia vuota significa che in tutte le posizioni della tupla ci sono degli "argomenti nulli" che possono poi essere sostituiti, utilizzando le operazioni che seguono, con elementi aventi un tipo qualunque.

- **puttupla**(t, p, e): scrittura dell'elemento e in posizione p nella tupla t. Il risultato dell'operazione è una nuova tupla uguale a t tranne che per l'elemento in posizione p. t non viene modificata dalla operazione.

- **gettupla**(t, p): lettura dell'elemento in posizione p nella tupla t. Il tipo del risultato è il tipo dell'elemento letto.

4.2 Le istruzioni del linguaggio

L'engine dell'architettura consiste in un programma in cui gli operatori:

- ";": concatenazione di istruzioni;

- "||": esecuzione in interleaving di istruzioni;

- "|": scelta non deterministica di istruzioni;

- "**if then else endif**";

- "**while do endwhile**";

vengono applicati alle seguenti istruzioni primitive, la cui sintassi formale viene data nell'appendice A:

- **decl** variabile: dichiarazione di variabile.
  Al momento della dichiarazione, alla variabile viene associato il valore **NIL**. In questo modo può essere associato alla variabile, ad esempio tramite assegnazione, un valore di tipo qualsiasi.

- $P(a_1, \ldots, a_n)$: chiamata di procedura.
  Le variabili di una chiamata di procedura possono essere passate per valore oppure per riferimento, queste ultime sono contrassegnate dal simbolo "!". Come nei linguaggi imperativi tradizionali, le modifiche del valore associato ad una variabile passata per valore, non sono visibili al di fuori del corpo della procedura, mentre per una variabile passata per riferimento lo sono. Inoltre, quando una variabile viene passata per riferimento in una chiamata di procedura, all'interno della chiamata sono visibili le modifiche a tale variabile apportate dall'esterno della procedura da una eventuale seconda procedura in interleaving con la prima.

- *variabile* := *espressione*: assegnazione di variabile.
  Alla variabile viene associato il valore ottenuto dalla valutazione dell'espressione.

- **skip**: comando nullo.
  L'esecuzione di questo comando non ha alcun effetto.

- operazioni sulle componenti dell'architettura:

  - **get_comp**(componente, modo, variabile): estrazione di un elemento dalla componente.
    Alla variabile viene associato un elemento della componente selezionato in funzione del tipo della componente e del modo specificato.
  - **put_comp**(componente, modo, variabile): inserimento di un elemento nella componente.
    Il valore associato alla variabile viene inserito nella componente tenendo conto del tipo della componente e del modo specificato.
  - **del_comp**(componente, modo): cancellazione di un elemento contenuto nella componente.
    L'elemento viene cancellato in funzione del tipo della componente e del modo specificato.

- operazioni per la ricezione e l'invio dei messaggi:

  - **send**(destinatario, messaggio): invio di un messaggio.
    Il messaggio viene inviato al destinatario inserendo nell'etere del sistema la tripla:

    (mittente, destinatario, messaggio).
  - **receive**(mittente, messaggio): ricezione asincrona di un messaggio.
    La ricezione asincrona consiste nel prelevare il contenuto di un messaggio, in funzione del mittente, dall'etere del sistema, se questo non

è vuoto. Nel caso l'etere sia vuoto l'istruzione associa alla variabile *messaggio* la costante **NIL**.

  - **receive_sync**(mittente, messaggio): ricezione sincrona di un messaggio.
    La ricezione sincrona è simile alla precedente, differisce solo nel caso in cui l'etere non contenga alcun messaggio: in tal caso l'agente attende l'inserimento di un messaggio, mentre nel caso asincrono semplicemente non l'istruzione non ha nessun effetto.
  - **exec**(azione): esecuzione di una azione.
    Eseguire una azione consiste nell'inviare un messaggio con contenuto "azione" all'agente che simula l'ambiente del sistema.

- operazioni per la percezione e la generazione di eventi:
  - **percept**(variabile): percezione di un evento.
    La percezione istanzia la variabile con l'evento prelevato dall'ambiente del sistema.
  - **put_event**(evento): inserimento di un evento nell'ambiente di sistema.
    Questa istruzione viene eseguita dall'agente che simula l'ambiente.
  - **remove_event**(evento): rimozione di un evento dall'ambiente.
    Come la precedente, viene eseguita dall'agente che simula l'ambiente.

Il corpo delle procedure utilizzate nell'engine è costituito dagli operatori di concatenazione, interleaving, scelta non deterministica e dalle istruzioni **if then else endif** e **while do endwhile**, applicati alle istruzioni menzionate precedentemente.

4.3 Definizione delle architetture

Come già osservato in precedenza l'architettura concreta di un agente è ottenuta come istanza di una architettura astratta.
Nella architettura astratta vengono definite le componenti, le possibili procedure che costituiscono l'engine astratto e l'engine astratto stesso.
Nell'architettura concreta vengono definite le procedure dell'engine astratto che non sono definite nell'architettura astratta. Queste procedure non vengono definite nell'architettura astratta in quanto dipendono da scelte implementative che non hanno alcuna rilevanza nella descrizione ad alto livello dell'architettura. Ad esempio nella descrizione astratta dell'architettura BDI la semantica informale della

abstract_architecture $\{abs\_arch\_name\}$
{
    components
    {
        $mode\ comp\_name_1;$
        $\vdots$
        $mode\ comp\_name_{n_1};$
    };

    procedures
    {
        $proc_1$ { ... };
        $\vdots$
        $proc_r$ { ... };
    };

    engine
    {
        decl x_1
        $\vdots$
        decl x_n

        $\mathrm{proc}_1\ op \ldots\ op\ \mathrm{proc}_r$
    }
};

Figura 4.1: Schema della definizione di una architettura astratta

procedura $get\_trigger(p,\ t)$ è quella di associare alla variabile t il trigger del piano associato a p, questa procedura viene definita nell'architettura concreta in quanto bisogna conoscere la struttura dati utilizzata per realizzare il piano, mentre nell'architettura astratta questa informazione manca, non essendo rilevante nel descrivere il funzionamento dell'architettura.

4.3.1 Definizione dell'architettura astratta

La sintassi della definizione di una architettura astratta ha la forma mostrata in figura 4.1. abs_arch_name rappresenta il nome dell'architettura astratta. **components** introduce la definizione delle componenti dell'architettura astratta: *mode* può assumere i valori **class** e **agent**, che significano rispettivamente che la componente deve essere istanziata nella definizione di classe oppure al momento della creazione dell'agente. Le componenti dichiarate in modalità **class** contengono le informazioni che caratterizzano il *programma* dell'agente, mentre le componenti dichiarate in modalità **agent** contengono quelle relative allo *stato*.

$$comp\_name_1 \ldots comp\_name_{n_1}$$

sono i nomi delle componenti. **procedures** introduce le definizioni ad alto livello delle procedure che vengono chiamate nell'engine. Per ogni procedura è possibile fornire indicazioni alternative (eventualmente una o nessuna) su come andrà realizzata:

$$proc_1\{\ldots\} \ldots proc_r\{\ldots\}$$

i cui corpi sono composti dalle istruzioni descritte nella sezione 4.2.
La definizione dell'engine consiste nella definizione del programma che determina il comportamento dell'architettura:

$$\mathrm{proc}_1 \; op \ldots \; op \; \mathrm{proc}_r.$$

Esempio 4.3.1
Si consideri una architettura BDI costituita da tre componenti:

- *plans_components*: la cui istanza rappresenta il *programma* dell'agente, nel caso dell'architettura BDI contiene i piani corrispondenti alla conoscenza procedurale dell'agente;

- *beliefs_components* la cui istanza rappresenta lo *stato* dell'agente, in questo caso le credenze;

- *intentions_components*: anche l'istanza di questa componente insieme alla precedente fa parte dello *stato* dell'agente, in particolare le intenzioni.

il cui engine ha il compito di:

- percepire un evento;

- selezionare un piano;

- inserire il piano selezionato nelle intenzioni;

- selezionare una intenzione;

- eseguire una istruzione dell'intenzione selezionata.

L'architettura astratta viene completamente definita nell'appendice B.
La componente *plans_ components* dell'architettura astratta viene dichiarata in modalità **class**, ciò significa che verrà istsanziata durante la fase di definizione della classe di agente, mentre le componenti *beliefs_ components* e *intentions_ components* sono dichiarate in modalità **agent**, quindi verranno istanziate nella fase di creazione dell'agente.

Nell'engine dell'architettura astratta *bdi* vengono dichiarate le quattro variabili globali:

- *event*;
- *selected_event*;
- *triggered_plans*;
- *selected_plans*.

Le prime due variabili vengono utilizzate per mantenere gli eventi percepiti dall'ambiente del MAS, le altre mantengono informazioni riguardanti rispettivamente i piani triggerati dall'evento e tra questi ultimi quelli le cui condizioni sono verificate.
La dichiarazione di queste tre variabili nell'engine dell'architettura astratta le rende globali cioè visibili in tutte le procedure dell'engine a qualsiasi livello di annidamento.
Le quattro variabili globali non vengono inizializzate in quanto non è ancora conosciuta l'implementazione degli eventi percepiti, della struttura dati che mantiene i piani triggerati e di quella che mantiene i piani la cui condizione è verificata. La loro inizializzazione avverrà nella fase di definizione dell'architettura concreta.

La procedura *percept_event* ha lo scopo di percepire un evento utilizzando la procedura *percept_event_from_extern*, quindi di inserirlo nella variabile globale *event*.
Al livello di descrizione astratto dell'architettura non si possono fare ipotesi sull'implementazione di *event*, quindi *percept_event_from_extern* verrà definita nell'architettura concreta.

La procedura *plan_triggering* ha lo scopo di istanziare la variabile globale *triggered_plans* con i piani triggerati dall'evento percepito dalla procedura *percept_event*.
In questa procedura vengono dichiarate cinque variabili locali: *event_trigger*,

plan_trigger, *plan*, *aux_plans* e *empty*, utilizzate per contenere rispettivamente il trigger di un evento, il trigger di un piano, un piano, uno stack di piani ed infine un valore booleano utilizzato come guardia.
Le prime tre variabili contengono informazioni non specificate nell'architettura astratta, quindi non vengono inizializzate, *aux_plans*, invece, è una variabile di servizio il cui utilizzo non caratterizza il comportamento dell'architettura BDI, quindi può essere inizializzata, in questo caso, con uno stack vuoto. La variabile booleana *empty* viene inizializzata nelle procedure in cui è utilizzata.
Per accedere alle variabili ed alle componenti vengono utilizzate procedure quali *get_event*, *get_event_trigger*, *is_empty_plans_component*, *get_plans_ component* ed altre, che verranno definite nella architettura concreta.

La procedura *plan_selection* ha il compito di valutare le condizioni dei piani triggerati, e quindi istanziare la variabile *selected_plans* con i piani la cui condizione è verificata.
Anche in questa procedura vengono dichiarate variabili locali e vengono utilizzate procedure che verranno definite nell'architettura concreta al fine di accedere alle informazioni contenute nelle componenti e manipolare piani, oppure condizioni di piani.

Una volta ottenuto l'insieme dei piani che sono stati triggerati e le cui condizioni sono verificate, la procedura *upgrade_intentions_component* ne sceglie uno e lo inserisce nella componente *intentions_component*.
La scelta del piano da inserire nella componente delle intenzioni viene fatta tramite la procedura *select_plan* che viene definita nell'architettura concreta.
La definizione di *select_plan* corrisponde alla definizione della strategia con cui viene scelto un piano affinché diventi una intenzione; diverse architetture possono avere diverse strategie per trasformare un piano in una intenzione pur essendo tutte architetture BDI.

La procedura *exec_intention*() seleziona una intenzione dalla componente *intentions_ component*, quindi seleziona una azione di tale intenzione e la esegue.
La procedura *get_intentions_component* estrae l'intenzione, *get_action* estrae una azione da tale intenzione e quindi *exec_action* la esegue. □

4.3.2 Definizione dell'architettura concreta

La sintassi della definizione dell'architettura concreta ha la forma mostrata nella figura 4.2.
Il termine *abs_arch_name* è il nome dell'architettura astratta che viene istan-

architecture $\{arch\_name\}$ **is a** $\{abs\_arch\_name\}$
{
    components
    {
        $mode\ comp\_name_1 : \ comp\_type_1;$
        $\vdots$
        $mode\ comp\_name_n : \ comp\_type_c;$
    };

    init_global_vars
    {
        $v_1 := exp_1;$
        $\vdots$
        $v_n := exp_n;$
    }

    procedures
    {
        $proc_1 \ \{ \ \ldots \ \};$
        $\vdots$
        $proc_r \ \{ \ \ldots \ \};$
    }
};

Figura 4.2: Schema della definizione dell'architettura concreta ziata ed $arch\_name$ è il nome dell'istanza, nella sezione **components** viene attribuito un tipo alle componenti dichiarate nell'architettura abs_arch_name, nella sezione **init_global_vars** vengono istanziate le variabili globali $v_1 \dots v_n$ rispettivamente con le espressioni $exp_1 \dots exp_n$, mentre in **procedures** vengono definite le procedure dell'engine dell'architettura astratta che dipendono dall'implementazione.

Esempio 4.3.2
Nella definizione dell'architettura concreta vengono fissati i dettagli che dipendono dall'implementazione, quindi vengono associati i tipi alle componenti, vengono fissate le strutture dati che rappresentano l'informazione all'interno delle componenti e le strategie legate alla loro manipolazione.
In questo esempio vengono definite due istanze, bdi_1 e bdi_2, dell'architettura astratta definita nell'esempio 4.3.1.
In entrambe le istanze dell'architettura *bdi* alle componenti $plans\_component$ e $beliefs\_component$ viene associato il tipo **list**, mentre alla componente *intentions_component* il tipo **set**. Quest'ultima scelta permette una selezione non deterministica delle intenzioni da eseguire.

Gli eventi percepiti sono coppie formate da un'evento trigger e da un valore:

evento: (trigger, valore)

L'evento trigger descrive un fatto manifestatosi nell'ambiente, il valore rappresenta una un'informazione associata a tale fatto.

Le credenze contenute nella componente $beliefs\_component$ sono rappresentate come identificatori di credenza a cui viene associato un valore, tale associazione viene rappresentata da una coppia il cui primo elemento rappresenta l'identificatore ed il secondo il valore. Ad esempio se un agente crede che il valore di corrente che passa all'interno di un cavo elettrico sia 5, questa credenza sarà formalizzata dalla coppia (corrente, 5).

I piani contenuti nella componente $plans\_component$ sono rappresentati da triple il cui primo elemento è il trigger, il secondo la condizione ed il terzo una sequenza di azioni:

piano: (trigger, condizione, sequenza di azioni)

Il trigger del piano è una stringa di caratteri, la condizione una espressione che può essere valutata in funzione del valore associato all'evento percepito ed alle credenze mantenute.
In questo esempio i piani possono avere tre possibili condizioni:

- $(soglia\_max, id)$: verificata se il valore associato alla credenza id è maggiore del valore percepito nell'evento e;

- $(soglia\_min, id)$: verificata se il valore associato alla credenza id è minore del valore percepito nell'evento e;

- (?, id): verificata se il valore associato alla credenza id corrisponde a al valore booleano true.

La sequenza di azioni viene implementata come una lista di coppie il cui primo elemento rappresenta il tipo di azione e il secondo un'azione vera e propria oppure un messaggio da inviare, oppure infine, una credenza da modificare.
Le possibili azioni sono quindi:

- $(\$, a)$: se il primo elemento della coppia è il simbolo \$ l'azione a modifica solo l'ambiente in cui è collocato l'agente;

- $(send, m)$: se il primo elemento della coppia è $send$ l'esecuzione di tale azione consiste nell'inviare il messaggio m al destinatario specificato nella credenza identificata da $receiver$;

- $(\#, (id, v))$: Il simbolo # indica che l'esecuzione dell'azione modifica il valore della credenza id, se tale credenza appartiene già alla componente $beliefs\_component$, alimenti aggiunge la nuova credenza (id, v).

Le intenzioni contenute nella componente $intentions\_component$ corrispondono alla sequenza di azioni dei piani il cui trigger era in accordo con l'evento percepito e la cui condizione era soddisfatta.

Nell'appendice B vengono definite due istanze dell'architettura astratta definita nell'esempio 4.3.1.

Le due istanze differiscono sia nel modo in cui vengono percepiti gli eventi che nella strategia con cui viene scelto un piano affinché diventi una intenzione.

Un agente basato sull'architettura bdi_1 percepisce gli eventi che si manifestano, sia come messaggi inviati da altri agenti, che come eventi prelevati dall'ambiente in cui è collocato, mentre un agente basato sull'architettura bdi_2 percepisce solo gli eventi che si manifestano nell'ambiente e non è in grado di ricevere messaggi.

La procedura $percept\_event\_from\_extern$ è utilizzata nell'architettura astratta con il fine di associare alla variabile globale $event$ l'evento percepito, nelle due istanze la definizione di tale procedura e l'inizializzazione della variabile *event* vengono realizzati in modo differente.

Nella definizione relativa all'architettura *bdi_1*, la variabile *event* viene istanziata con espressioni di tipo coda, in modo da contenere gli eventi percepiti e la procedura *percept_event_from_extern* viene realizzata in modo da permettere l'interleaving tra la ricezione di un messaggio e la percezione di un evento:

```
percept_event_from_extern()
{
   decl sender;
   decl e_1;
   decl e_2;

   (
     percept(e);
     event := insqueue(e_1)
   ) ||
   (
     get_beliefe_component("sender", sender);
     receive(sender, e_2);
     event := insqueue(e_1)
   )
}
```

L'istruzione primitiva del linguaggio **percept**($\cdot$) realizza la percezione dell'evento, la procedura *get_beliefe_component* associa alla variabile *sender* il mittente del messaggio ed infine l'istruzione primitiva **receive**($\cdot$, $\cdot$) riceve il messaggio se tale messaggio è stato inviato.
Nella definizione relativa all'architettura *bdi_2* la variabile *event* viene istanziata con la tupla vuota di dimensione due in modo da contenere un solo evento, la procedura *percept_event_from_extern* viene realizzata in modo da percepire soltanto gli eventi tramite la primitiva **percept**($\cdot$):

```
percept_event_from_extern()
{
   percept(event)
}
```

I piani il cui trigger è in accordo con l'evento percepito e le cui condizioni sono soddisfatte, vengono inseriti nella variabile globale *selected_plans*, e da questa

variabile viene selezionato tramite la procedura *select_plan*, il piano da inserire nella componente *intentions_component*.
Nell'architettura bdi_1 la variabile *selected_plans* viene istanziata con una *lista* di piani, mentre nell'architettura bdi_2 viene istanziata con un *insieme* di piani, in questo modo nel primo caso viene selezionato il primo elemento della lista:

```
select_plan(plan)
{
   plan := gethlist(selected_plans)
}
```

nel secondo caso viene selezionato in modo casuale un piano appartenente all'insieme:

```
select_plan(plan)
{
  plan := getset(selected_plans)
}
```

□

4.4 Definizione del M.A.S.

Il MAS consiste in un insieme di agenti che condividono sia l'ambiente **ENV** in cui sono collocati, che un etere **ETH** utilizzato nella comunicazione tra agenti.
La definizione del MAS (figura 4.3) consiste nel:

- definire la classe degli agenti;
- definire l'inizializzazione dell'ambiente;
- definire l'inizializzazione dell'etere;
- definire la creazione delle istanze dei tipi di agenti, cioè gli agenti veri e propri.

La definizione della *classe dell'agente* consiste nell'istanziazione delle componenti con le informazioni che caratterizzano il *programma* dell'agente.
La definizione della classe dell'agente è denotata dalla parola chiave **class_agent** in cui $arch\_name_1$ e $classagent\_name_1$ sono, rispettivamente il nome dell'architettura e della classe dell'agente.
Le componenti istanziate, $comp\_name_1 \ldots comp\_name_c$, sono quelle che nella

MAS
{

class_agent $\{arch\_name_1,\ classagent\_name_1\}$
{
init_comp $comp\_name_1$ $[elem_{1_1},\ \ldots,\ elem_{1_i}]$;
$\vdots$
init_comp $comp\_name_c$ $[elem_{c_1},\ \ldots,\ elem_{c_j}]$;
};
$\vdots$
class_agent $\{arch\_name_a,\ classagent\_name_a\}$
{
$\vdots$
};

init_ENV $[event_1,\ \ldots,\ event_e]$;
init_ETH $[msg_1,\ \ldots,\ msg_m]$;

create_agent$(typeagent\_name_{t_1},\ agent\_name_1)$;
{
init_comp $comp\_name_1$ $[elem_{1_1},\ \ldots,\ elem_{1_i}]$;
$\vdots$
init_comp $comp\_name_c$ $[elem_{c_1},\ \ldots,\ elem_{c_j}]$;
}
$\vdots$
create_agent$(typeagent\_name_{t_{ag}},\ agent\_name_{ag})$;
{
$\vdots$
}
}

Figura 4.3: Schema della definizione di un MAS definizione dell'architettura astratta sono dichiarate come **class** e vengono istanziate rispettivamente con i valori $elem_{1_1}$, ... , $elem_{1_i}$ e $elem_{c_1}$, ... , $elem_{c_j}$.

Ambiente e etere vengono inizializzati in **init_ENV** e **init_ETH** rispettivamente con gli eventi iniziali $event_1$, ... , $event_e$ e con i messaggi iniziali msg_1, ... , msg_m.

Infine le istanze degli agenti vengono generate da **create_agent** i cui parametri sono il nome della classe di agente istanziata ed il nome dell'istanza, nella creazione degli agenti vengono anche istanziate le componenti dichiarate come **agent**.

Esempio 4.4.1
La definizione dello *stato* dell'agente corrisponde alla definizione del suo comportamento; in questo esempio vengono create due classi di agente relative alle due istanze di architettura astratta dell'esempio 4.3.2.

La prima classe definisce un *robot* che si muove attraverso un percorso rettilineo ed in grado di trasportare del materiale. Il percorso è contrassegnato ai due estremi da elementi di fine-corsa, quando li raggiunge arresta il suo moto; se è scarico carica il materiale, mentre se è carico lo scarica, quindi ruota su se stesso di 180 gradi e riparte in direzione dell'estremo opposto del percorso.
Durante il moto del robot un ostacolo può invaderne il percorso, in questo caso l'ostacolo viene percepito e se la distanza di impatto con il robot è inferiore al valore di soglia mantenuto nelle sue credenze si arresta, altrimenti procede verso il fine-corsa.
Il robot viene spinto da un propulsore elettrico e se il materiale caricato è troppo pesante, il propulsore ha un sovraccarico di corrente rischiando di guastarsi. Inoltre la prolungata attività del robot favorisce un surriscaldamento del propulsore aumentando la probabilità che si verifichino dei guasti.
Al fine di preservare il buon funzionamento del robot, all'architettura bdi_2 dell'esempio 4.3.2, viene associata una classe di agente *monitor*: questa controlla la corrente assorbita dal propulsore e nel caso venga superata la soglia mantenuta nelle credenze del monitor, invia al robot un messaggio contenente un'informazione di pericolo, la cui ricezione può arrestarne il moto.
Il monitor controlla anche la temperatura sprigionata dal propulsore e se questa supera il livello di guardia, attiva il meccanismo di raffreddamento oppure invia il messaggio di pericolo.

La componente *plan_component* della classe *robot* viene istanziata con cinque

piani: i primi due relativi alla comparsa di un ostacolo lungo il percorso, gli altri due alla percezione del fine-corsa e l'ultimo relativo alla percezione di un messaggio di pericolo da parte di un agente di classe monitor.
La componente *plans_component* della classe *monitor* viene invece istanziata con tre piani relativi al manifestarsi di un sovraccarico di corrente ed all'eccesivo aumento di temperatura.
In entrambe le classi le credenze contengono informazioni necessarie alla valutazione delle condizioni dei piani, all'invio ed alla ricezione dei messaggi.

I piani, come descritto nell'esempio 4.3.1 hanno la forma di una tripla in cui il primo elemento è un evento trigger, il secondo una condizione ed il terzo una lista di azioni corrispondenti al corpo del piano.

I piani relativi alla percezione di un ostacolo hanno come evento trigger la stringa "*ostacolo*"; il loro evento di attivazione è una coppia il cui primo elemento corrisponde alla stringa "*ostacolo*" ed il secondo è il valore della distanza del robot dall'ostacolo.
La condizione di tali piani consiste nel verificare che la distanza del *robot* dall'ostacolo sia inferiore alla soglia associata alla credenza *distanza*. In caso la condizione sia verificata vengono selezionati due piani: nel primo viene eseguita l'istruzione ($, stop) il cui effetto è quello di arrestare il robot, nel secondo vengono eseguite le istruzioni ($, dietrofront) e ($, riparti) il cui effetto è quello di fare ruotare il robot di 180 gradi su se stesso e farlo ripartire nella direzione da cui proviene.
I piani relativi alla percezione del fine-corsa hanno come evento trigger la stringa "$fine\_corsa$" e come condizioni le coppie (?,carico) e (?,scarico), la cui semantica corrisponde alla verifica del valore booleano associato rispettivamente alle credenze "*carico*" e "*scarico*".
Nel caso il robot sia carico la credenza relativa alla stringa "*carico*" viene aggiornata con il valore "$false$" attraverso l'istruzione (#, (carico, false)); il moto del robot viene arrestato, viene eseguita l'azione ($, scarica), quindi il robot esegue l'istruzione ($, dietrofront) e viene fatto ripartire dall'esecuzione dell'istruzione ($, riparti) ed infine la credenza relativa alla stringa "*scarico*" viene aggiornata con il valore "$true$". Se il robot è scarico la sequenza di operazioni relativa al corpo del piano è simile al caso precedente: la credenza "*scarico*" viene aggiornata con il valore "$false$", il materiale viene caricato, il robot viene fatto ripartire e la credenza "*carico*" viene aggiornata con il valore "$true$".
Il piano relativo alla percezione di un sovraccarico ha come evento trigger la stringa "*corrente*" e come condizione ($soglia\_max, s\_corrente$); condizione verificata solo se il valore di corrente percepito è superiore al valore associato alla credenza $s\_corrente$, in tal caso viene inviato al robot il messaggio di pericolo.
I piani relativi all'aumento della temperatura hanno come evento trigger la stringa

"$temperatura$" e, come condizione la coppia ($soglia\_max, s\_temp$), che consiste nella comparazione del valore di temperatura percepito con il valore di soglia mantenuto nella credenza s_temp. Se la condizione viene verificata viene attivato il sistema di raffreddamento del robot nel primo caso, mentre nel secondo l'invio del messaggio di pericolo. $\square$

Capitolo 5

Semantica del linguaggio

La semantica di un programma che descrive un sistema multi-agente è data dall'albero delle transizioni tra le configurazioni del sistema.
Una configurazione ha la forma di una n-upla:

$$[ENV,\ ETH,\ s_{a_1},\ \dots\ ,\ s_{a_n},\ s_{env}] \tag{5.1}$$

contenente informazioni riguardanti l'ambiente, l'etere e lo stato degli agenti del MAS.
In una configurazione:

- ENV è l'insieme degli eventi che si manifestano nell'ambiente. Solo l'agente ambiente inserisce eventi in ENV, gli altri agenti possono solo "leggere" in ENV ma non "scrivere".

- ETH è l'insieme dei messaggi, ordinati secondo l'ordine di arrivo, che rappresenta il mezzo di comunicazione tra gli agenti: l'etere.
  La forma dei messaggi scambiati è una tripla i cui elementi sono:

  $(mittente, destinatario, contenuto)$

  L'invio del messaggio M dall'agente A all'agente B consiste nell'inserimento della tripla (A,B,M) in ETH. L'agente B riceve M spostandolo da ETH in una propria componente definita nella architettura.

- s_{a_i} è una coppia

  $$\langle ex\_env,\ comps\rangle \tag{5.2}$$

  che rappresenta lo *stato* dell'agente i-esimo del MAS. In uno stato:

  - *ex_env* rappresenta l'ambiente di esecuzione dell'agente, dove si mantengono le associazioni tra variabili ed i valori a loro assegnati;

- *comps* è rappresenta il contenuto delle componenti che costituiscono l'architettura dell'agente.

- s_{env} è lo stato dell'agente env_agt, il cui scopo è la simulazione dell'ambiente del sistema. Il suo stato non differisce dallo stato degli altri agenti in quanto env_agt è definito tramite i concetti di architettura espressi nel capitolo 4.

5.1 Ambiente di esecuzione dell'agente

L'ambiente di esecuzione è la struttura formale che durante l'attività dell'agente, mantiene i valori associati alle variabili dell'engine. Tale struttura è una funzione la cui definizione formale verrà data in seguito.
Ad esempio, si consideri l'ambiente vuoto e in cui non esiste nessuna associazione tra variabili e valori. Si avrà quindi $ex\_env(x) = \perp$ per ogni variabile $x \in \mathcal{V}$. Si consideri inoltre il seguente segmento di programma:

decl x;
$x := 3$;
$p()$;
$\vdots$

L'esecuzione della dichiarazione di variabile **decl** x nell'ambiente e, associa alla variabile x il valore **NIL**, il che significa $ex\_env(x) =$ **NIL**, l'esecuzione dell'assegnazione di variabile $x := 3$ modifica e in modo che il valore associato a x sia 3: $ex\_env(x) = 3$. L'esecuzione della chiamata di procedura $p()$ ha l'effetto di generare un ambiente locale alla chiamata ex_env' tale che $ex\_env'(x) = \perp$ per ogni variabile $x \in \mathcal{V}$. Nell'ambiente ex_env' possono essere dichiarate variabili senza che siano in conflitto con la variabile x dichiarata in ex_env.
All'ambiente ex_env viene associato l'ambiente ex_env' estendendo il dominio di definizione di ex_env agli identificatori di chiamata di procedura, in questo esempio all'identificatore di chiamata di procedura p. In questo modo l'espressione $ex\_env(p)$ corrisponde all'ambiente locale ex_env'.
Se il corpo della procedura relativa alla chiamata $p()$ è costituito dal programma seguente:

{
  decl x;
  $x := 5$
{

l'esecuzione della dichiarazione **decl** x nell'ambiente locale ex_env' comporta l'aggiornamento di ex_env' con l'associazione del valore **NIL** alla variabile x, quindi

$ex\_env'(x) = \textbf{NIL}$, mentre l'assegnazione $x := 5$ aggiorna ex_env' in modo che $ex\_env'(x) = 5$.
Siccome l'ambiente locale ex_env' è raggiungibile dall'ambiente di esecuzione dell'agente ex_env, attraverso la sua applicazione all'identificatore di chiamata di procedura p, si ha che $ex\_env(p)(x) = ex\_env'(x) = 5$.
In questo modo l'ambiente di esecuzione dell'agente ex_env riesce a tener conto delle associazioni di valore relative alle variabili locali dichiarate nei corpi delle procedure, senza che si manifesti alcun conflitto tra di loro, infatti $ex\_env(x) = 3$ e $ex\_env(p)(x) = 5$.
Le procedure dell'engine possono essere eseguite in interleaving, quindi è necessario che l'insieme degli identificatori di chiamata di procedura id_proc sia un insieme isomorfo all'insieme dei naturali, in modo da poter sempre garantire un identificatore "fresco" per ogni chiamata eseguita. Ad esempio nell'eseguire il programma:

$$\begin{array}{l} \vdots \\ p(x) \parallel p(x) \\ \vdots \end{array}$$

non si creano ambiguità perché alla chiamata di procedura eseguita per prima viene assegnato l'identificatore p_1, mentre a quella eseguita per seconda viene assegnato l'identificatore p_2.

Un ambiente generico può essere rappresentato graficamente come l'albero di figura 5.1, in cui la radice rappresenta l'ambiente in cui sono definite le variabili globali dell'engine, le foglie sono i valori semantici assegnati alle variabili, i nodi non foglia rappresentano ambienti locali relativi a chiamate di procedura e gli archi rappresentano l'aggiornamento dell'ambiente in seguito all'esecuzione di una dichiarazione di variabile globale, dichiarazione di variabile in un ambiente locale ad una procedura, oppure ad una chiamata di procedura.
Il valore assegnato alla variabile x_n di figura 5.1 è rappresentato dall'espressione $e(x_n)$, mentre l'ambiente locale e_{1_n} è rappresentato dall'espressione $e(p_1)(p_{1_n})$ dove p_1 e p_{1_n} sono identificatori che rappresentano rispettivamente le chiamate di procedura $p_1()$ e $p_{1_n}()$.
L'n-esimo ambiente locale viene rappresentato dall'espressione $ex\_env(p_1)\dots(p_n)$ e la sequenza $p_1 \dots p_n$ viene chiamata **path**. Il path rappresenta un cammino all'interno dell'albero che rappresenta la funzione ambiente e viene utilizzato per individuare un ambiente locale ad una chiamata di procedura, l'ambiente in cui sono dichiarate le variabili x_1 e x_2 è individuato dal path nullo **NIL**: $ex\_env(\textbf{NIL})(x_2) = ex\_env(x_2)$.

L'ambiente di esecuzione è una funzione così definita:

$$ex\_env : \mathcal{V} \bigcup PATH \longrightarrow \mathbf{VAL} \bigcup PATH \bigcup ex\_env$$

in cui ad ogni identificatore di variabile viene associato **NIL**, oppure un valore, oppure un *path* e ad ogni *path* viene associato un ambiente locale all'esecuzione della procedura:

$$ex\_env(id) = \begin{cases} \{\mathbf{NIL}\} \bigcup \mathbf{VAL} \bigcup PATH & id \in \mathcal{V} \\ ex\_env & id \in PATH \\ \bot & altrimenti \end{cases} \tag{5.3}$$

dove $\mathcal{V}$ è l'insieme delle variabili definito nella sezione 4.1.1, $PATH$ l'insieme degli identificatori di chiamata di procedura e **VAL** l'insieme dei valori semantici del linguaggio.

5.2 Semantica di un agente

Il sistema multi-agente è composto da un agente che simula l'ambiente del sistema e da diversi agenti collocati in tale ambiente. Questi ultimi possono percepire gli eventi che si manifestano nell'ambiente ed eseguire azioni il cui effetto modifica lo stato dell'ambiente.
Ad esempio l'agente ping-pong definito nell'esempio 4.4.1 esegue azioni, quali:

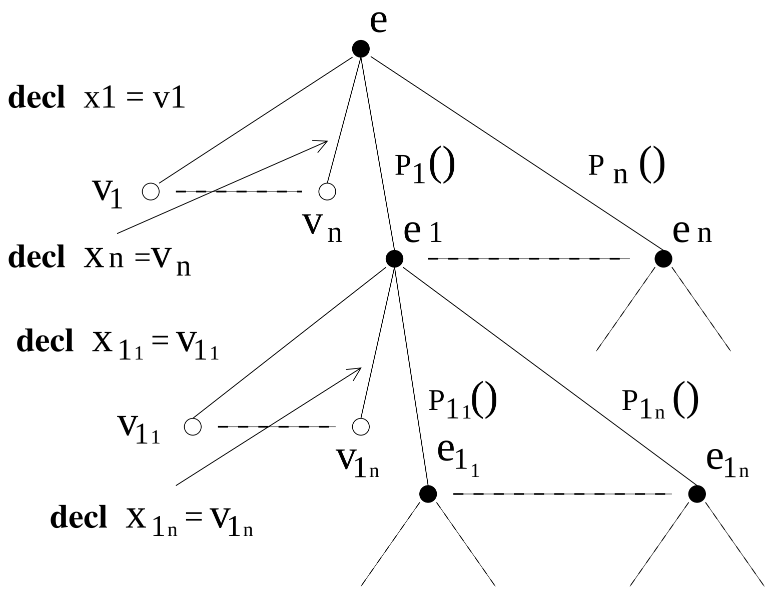


Figura 5.1: Albero rappresentante la funzione ambiente "e".

($, stop) e ($, riparti)

che modificano lo stato dell'ambiente del sistema.
Quando il robot ping-pong esegue l'azione "riparti", il motore del robot si accende e l'agente si muove in una data direzione. L'accensione del motore comporta un aumento di temperatura sia del motore che dell'ambiente circostante al robot.
L'agente che simula l'ambiente del sistema calcola gli effetti dell'accensione del motore sull'ambiente e quindi inserisce in ENV gli eventi relativi all'aumento della temperatura, in questo modo gli eventi possono essere percepiti dagli altri agenti del sistema; nell'esempio citato precedentemente l'aumento della temperatura viene percepito dall'agente che monitorizza il funzionamento del robot.

L'esecuzione dell'agente che simula l'ambiente è descritta dalla chiusura transitiva e riflessiva della relazione $\overset{env\_agt(a)}{\longmapsto}$, che consiste in una transizione tra configurazioni *configurazioni-agente_ ambiente* rappresentate dalla tripla:

$$(stato\_agente,\ ENV,\ ETH) \tag{5.4}$$

dove *stato_agente* è la coppia descritta in (5.2) ed ENV rappresenta l'ambiente in cui sono collocati gli agenti.
La relazione $\overset{env\_agt(a)}{\longmapsto}$ è descritta dalla meta-regola seguente:

$$\frac{(\langle env, comps\rangle, ENV, ETH) \overset{N\_I(env,a)}{\longmapsto} (\langle env', comps'\rangle, ENV', ETH')}{(\langle env, comps\rangle, ENV, ETH) \overset{env\_agt(a)}{\longmapsto} (\langle env', comps'\rangle, ENV', ETH')}$$

L'esecuzione degli altri agenti del sistema è descritta dalla chiusura transitiva e riflessiva della relazione $\overset{agt(a)}{\longmapsto}$, che consiste in una transizione tra *configurazioni-agente*, dove una *configurazione-agente* è rappresentata dalla coppia:

$$(stato\_agente,\ ETH) \tag{5.5}$$

in cui *stato_agente* è la coppia descritta in (5.2) ed ETH rappresenta l'etere.
La relazione $\overset{agt(a)}{\longmapsto}$ è descritta dalla meta-regola seguente:

$$\frac{(\langle env, comps\rangle, ETH) \overset{N\_I(env,a)}{\longmapsto} (\langle env', comps'\rangle, ETH')}{(\langle env, comps\rangle, ETH) \overset{agt(a)}{\longmapsto} (\langle env', comps'\rangle, ETH')}$$

in entrambe le meta-regole viene utilizzata la funzione N_I, che ha come argomenti il nome dell'agente che deve eseguire l'azione, l'ambiente di esecuzione locale in cui la esegue e restituisce: l'istruzione che deve essere eseguita, il nome dell'agente che la esegue (lo stesso passato come argomento), ed il path che identifica l'ambiente locale in cui viene eseguita.

5.2.1 La funzione di selezione delle istruzioni "N_I"

Il comportamento di un agente è determinato dall'architettura e dalle informazioni che istanziano le sue componenti e dall'engine, cioè il programma che controlla il flusso di accesso e modifica delle informazioni contenute nelle componenti.
Dato un engine, lo stato delle componenti, lo stato dell'etere e dell'ambiente in cui sono collocati gli agenti del MAS, il comportamento di un agente in un particolare istante dipende dall'istruzione dell'engine che deve essere eseguita.
Le istruzioni che possono essere eseguite sono quelle descritte nella sezione 4.2, quindi chiamate di procedura, istruzioni di accesso e modifica delle componenti, istruzioni di ricezione ed invio di messaggi, percezione di eventi, esecuzione di azioni.
La funzione "N_I" seleziona la prima istruzione dell'engine che deve essere eseguita da un particolare agente: ha come argomenti il nome dell'agente, ed il suo ambiente di esecuzione, come risultato: il nome dell'agente che esegue l'istruzione (lo stesso passato come input), l'istruzione dell'engine che deve essere eseguita ed il path che individua l'ambiente locale in cui deve essere eseguita:

$$N\_I: \ AGENT \times E \ \longrightarrow \ AGENT \times PATH \times INSTRUCTIONS$$

dove:

- $AGENT$ è l'insieme dei nomi di agente del MAS;
- E è l'insieme dei degli ambienti di calcolo degli agenti del MAS;
- $PATH$ è l'insieme dei path degli ambienti di esecuzione degli agenti del MAS;
- $INSTRUCTION$ è l'insieme delle istruzioni dell'engine degli agenti del MAS;

La semantica di N_I è definita in funzione degli operatori di concatenazione ";", di interleaving "||", di scelta non deterministica "|", **if then else endif** e **while do endwhile**.

Se, ad esempio, l'engine dell'agente a consiste nella concatenazione di istruzioni (definite nella sezione 4.2)

$I_1; I_2$

la funzione N_I applicata ad a e env_a ha come risultato $[a, \mathbf{NIL}, I_1]$ dove env_a è l'ambiente locale dell'agente a, **NIL** è il path vuoto ed I_1 è una delle istruzioni definite nella sezione 4.2.

Se I_1 corrisponde ad una chiamata di procedura la funzione N_I ne tiene conto e quando seleziona l'istruzione I_{1_1} del corpo della procedura ha come risultato la tripla $[a, p_{I_1}, I_{1_1}]$, dove p_{I_1} è il path relativo all'ambiente locale generato dalla chiamata.
Trattandosi di una concatenazione N_I, seleziona I_2 solo quando l'esecuzione di I_1 è terminata, nel caso di una chiamata di procedura, solo quando l'esecuzione del corpo di I_1 è terminata.
Nel caso dell'interleaving di istruzioni

$$I_1 \parallel I_2$$

N_I seleziona non deterministicamente I_1 oppure I_2. Se l'istruzione selezionata è una chiamata di procedura, ad esempio I_1, alla chiamata successiva, N_I selezionerà non deterministicamente una istruzione del corpo di I_1 oppure I_2, nel caso l'istruzione scelta sia del corpo di I_1 il risultato di N_I sarà $[a, p_{I_1}, I_{1_1}]$, altrimenti $[a, \mathbf{NIL}, I_2]$.

Per quanto riguarda la scelta non deterministica di istruzioni

$$I_1 \mid I_2$$

N_I seleziona non deterministicamente I_1 oppure I_2. Se viene scelto I_1 alla chiamata successiva di N_I non sarà più possibile scegliere I_2 e viceversa.

Se la prossima istruzione dell'engine da eseguire è del tipo

if *cond* **then** I_1 **else** I_2 **endif**

N_I seleziona I_1 se l'espressione booleana *cond*, nell'ambiente di esecuzione dell'agente, viene verificata, altrimenti seleziona I_2.

Nel caso dell'istruzione

while *cond* **do** I_1 **endwhile**

la funzione N_I seleziona l'istruzione I_1 fino a quando l'espressione booleana *cond* viene verificata nell'ambiente di esecuzione dell'agente, quando *cond* non è più verificata N_I seleziona l'istruzione successiva.

5.2.2 Valutazione di variabili all'interno dell'ambiente di esecuzione dell'agente

Qualsiasi istruzione che legge oppure modifica il valore di una variabile necessita di strumenti per ottenere dall'ambiente il valore ad essa associato, oppure il cammino relativo all'ambiente locale in cui essa è stata dichiarata.

Ricerca di una variabile all'interno di un path

Sia ex_env l'ambiente di esecuzione di un agente e $path$ un cammino all'interno del grafo ad albero che definisce e, una variabile è definita in e se è stata dichiarata in tale ambiente oppure se è stata passata come parametro nella chiamata di procedura che ha generato tale ambiente locale.
Siccome tutte le variabili utilizzate in un programma devono essere dichiarate, se una variabile risulta indefinita in un ambiente locale, allora è una variabile globale dell'engine.
Ad esempio se la variabile x nell'ambiente locale individuato da $path$ risulta essere indefinita ($ex\_env(path)(x) \;=\; \perp$) è una variabile globale e deve essere necessariamente definita nella radice dell'albero ($ex\_env(x) \;=\; v$).

La relazione $\underset{ex\_env,path}{\overset{s\_var}{\longmapsto}}$ ha come parametri l'ambiente di esecuzione dell'agente ex_env ed un $path$ che identifica un ambiente locale ad una chiamata di procedura; tale relazione associa un identificatore di variabile al path relativo all'ambiente locale in cui è stata dichiarata la variabile.
Se la variabile è stata dichiarata nell'ambiente relativo al path fornito come parametro, allora $\underset{ex\_env,path}{\overset{s\_var}{\longmapsto}}$ ha come risultato il path stesso:

$$\frac{ex\_env(path)(x) \;=\; v}{x \underset{ex\_env,path}{\overset{s\_var}{\longmapsto}} path} \quad var(x), \quad value(v)$$

Se la variabile è stata passata per riferimento in una chiamata di procedura, allora $\underset{ex\_env,path}{\overset{s\_var}{\longmapsto}}$ ha come risultato il path relativo all'ambiente locale in cui è stata dichiarata la variabile:

$$\frac{ex\_env(path)(x) \;=\; path'}{x \underset{ex\_env,path}{\overset{s\_var}{\longmapsto}} path'} \quad var(x), \quad \neg value(path'), \quad path' \neq \perp$$

passarla per valore, corrisponde invece a dichiararla nell'ambiente locale.

Nel caso in cui la variabile risultasse indefinita nell'ambiente relativo a $path$, la relazione $\underset{ex\_env,path}{\overset{s\_var}{\longmapsto}}$ ha come risultato il path nullo **NIL** indicante che la variabile è dichiarata globale per l'engine:

$$\frac{ex\_env(path)(x) \;=\perp}{x \underset{ex\_env,path}{\overset{s\_var}{\longmapsto}} \mathbf{NIL}} \quad var(x)$$

Nelle tre meta-regola, ed in quelle che seguiranno, il predicato $var(\cdot)$ è vero se il suo argomento è una variabile e falso altrimenti, mentre il predicato $value(\cdot)$ è vero se il suo argomento è un valore semantico e falso altrimenti.

Valutazione di variabile

La relazione $\underset{ex\_env,path}{\overset{e\_var}{\longmapsto}}$ associa ad un identificatore di variabile il corrispondente valore nell'ambiente di esecuzione dell'agente, ha come parametri l'ambiente di esecuzione dell'agente ex_env ed un $path$ all'interno di ex_env.
Il risultato è ottenuto utilizzando la relazione $\underset{ex\_env,path}{\overset{s\_var}{\longmapsto}}$ per ottenere il path relativo all'ambiente in cui è stata dichiarata la variabile:

$$\frac{x \underset{e,path}{\overset{s\_var}{\longmapsto}} path' \qquad ex\_env(path')(x) \;=\; v}{x \underset{ex\_env,path}{\overset{e\_var}{\longmapsto}} v} \quad var(x), \quad value(v)$$

5.2.3 Valutazione di espressioni

La relazione $\underset{ex\_env,path}{\overset{e\_exp}{\longmapsto}}$ associa ad un'espressione costituita dalle operazioni definite in 4.1.1 il corrispondente valore semantico.
Le operazioni definite in 4.1.1 sono sintatticamente distinte: ad esempio, il simbolo di operazione **inshlist**, la cui semantica informale è quella di aggiungere un elemento in testa ad una lista, è rappresentato da un simbolo diverso da quello per aggiungere un elemento ad uno stack oppure ad una tupla, questa mancanza di overloading permette una valutazione corretta delle espressioni tipate senza la necessità di attribuire tipi alle variabili durante la loro dichiarazione.
La valutazione delle espressioni avviene attraverso le seguenti meta-regole:

$$\frac{x \underset{ex\_env,path}{\overset{e\_var}{\longmapsto}} v}{x \underset{ex\_env,path}{\overset{e\_exp}{\longmapsto}} v} \quad var(x)$$

Se in ex_env alla variabile x viene associato il valore v l'espressione costituita dalla variabile x viene valutata in v.

$$\frac{exp \underset{ex\_env,path}{\overset{e\_l\_exp}{\longmapsto}} v}{exp \underset{ex\_env,path}{\overset{e\_exp}{\longmapsto}} v}$$

La relazione $\overset{e\_l\_exp}{\underset{ex\_env,path}{\longmapsto}}$ associa il valore v all'espressione exp solo se è una espressione di tipo lista "ben formata", ad esempio l'espressione

instlist(**put_stack**(**e_s**, a),b)

non è ben formata in quanto il primo argomento dell'operazione **instlist** deve essere una lista, mentre nell'esempio è uno stack.
Un esempio di espressione lista ben formata è :

instlist(**e_l**, **put_stack**(**e_s**, a))

in cui l'espressione rappresenta una lista il cui unico elemento è uno stack.
Le espressioni di altro tipo, possono essere valutate nello stesso modo utilizzando le relazioni $\overset{e\_s\_exp}{\underset{ex\_env,path}{\longmapsto}}$, $\overset{e\_q\_exp}{\underset{ex\_env,path}{\longmapsto}}$, $\overset{e\_set\_exp}{\underset{ex\_env,path}{\longmapsto}}$, $\overset{e\_t\_exp}{\underset{ex\_env,path}{\longmapsto}}$, rispettivamente per le espressioni di tipo **stack**, **queue**, **set** e **tupla**:

$$\frac{exp \overset{e\_s\_exp}{\underset{ex\_env,path}{\longmapsto}} v}{exp \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} v} \qquad \frac{exp \overset{e\_q\_exp}{\underset{ex\_env,path}{\longmapsto}} v}{exp \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} v} \qquad \frac{exp \overset{e\_set\_exp}{\underset{ex\_env,path}{\longmapsto}} v}{exp \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} v} \qquad \frac{exp \overset{e\_t\_exp}{\underset{ex\_env,path}{\longmapsto}} v}{exp \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} v}$$

5.2.4 Dichiarazione di variabile

L'esecuzione di una dichiarazione di variabile, ad esempio **decl** x, ha l'effetto di aggiornare l'ambiente di esecuzione dell'agente in modo che la variabile x sia definita e ad essa venga associato il valore **NIL**.

La semantica della dichiarazione viene formalizzata con la meta-regolaseguente:

$$\frac{}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,decl]}{\longmapsto} (\langle ex\_env', C\rangle, ETH)}$$

dove:

- a è il nome dell'agente che esegue la dichiarazione;
- $path$ individua l'ambiente locale in cui viene eseguita la dichiarazione;
- decl $\equiv$ **decl** x
- ex_env rappresenta l'ambiente prima della dichiarazione;
- $ex\_env' = e(path)[\mathbf{NIL}/x]$ rappresenta l'ambiente aggiornato in seguito alla dichiarazione;

Le variabili globali vengono dichiarate al momento della definizione dell'architettura astratta ed inizializzate nella definizione dell'architettura concreta. Compaiono nell'ambiente di esecuzione della configurazione iniziale.

5.2.5 Assegnazione di variabile

L'assegnazione di variabile modifica il valore associato ad una variabile assegnandogli il valore ottenuto dalla valutazione di una espressione.
Ad esempio, se tramite l'ambiente, alla variabile x è associato il valore v e l'espressione exp viene valutata in v_1, allora l'istruzione di assegnazione $x := exp$, aggiorna l'ambiente in modo che alla variabile x sia associato il valore v_1.

La meta-regola seguente caratterizza la semantica dell'istruzione:

$$\frac{exp \underset{ex\_env,path}{\overset{e\_exp}{\longmapsto}} v \quad x \underset{ex\_env,path}{\overset{s\_var}{\longmapsto}} path'}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,x:=exp]}{\longmapsto} (\langle ex\_env', C\rangle, ETH)}$$

dove:

- a è il nome dell'agente che esegue l'assegnazione di variabile;
- $path$ individua l'ambiente locale in cui viene eseguita la dichiarazione;
- ex_env rappresenta l'ambiente prima della assegnazione;
- $ex\_env' = ex\_env(path')\big[v/x\big]$ rappresenta l'ambiente aggiornato in seguito alla assegnazione.

5.2.6 Chiamata di procedura

L'esecuzione della chiamata di procedura, consiste nella creazione di un ambiente locale alla chiamata.
L'ambiente locale viene inizializzato con le variabili che costituiscono i parametri formali definiti nella procedura, ai quali vengono associati i valori ottenuti dalla valutazione dei parametri attuali corrispondenti.
Nella chiamata di procedura i parametri attuali possono assumere la forma di espressioni oppure di variabili ed in quest'ultimo caso variabili passate per valore oppure per riferimento.
Nel caso in cui i parametri attuali abbiano la forma di espressioni, vengono valutati tramite la relazione $\underset{ex\_env,path}{\overset{e\_exp}{\longmapsto}}$ e quindi associati, nell'ambiente locale relativo alla chiamata di procedura, ai parametri formali.
Se sono variabili passate per valore, i parametri formali vengono inizializzati con i valori associati alle variabili, mentre se sono variabili passate per riferimento i parametri formali vengono inizializzati con il $path$ che individua l'ambiente locale in cui tali variabili sono state dichiarate.

La semantica della chiamata di procedura è descritta dalla meta-regola seguente:

$$\frac{p_{a_1} \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} v_1 \ \cdots \ p_{a_n} \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} v_n \quad p_{a_{n+1}} \overset{s\_var}{\underset{ex\_env,path}{\longmapsto}} path_{n+1} \ \cdots \ p_{a_{n+m}} \overset{s\_var}{\underset{ex\_env,path}{\longmapsto}} path_{n+m}}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,call]}{\longmapsto} (\langle ex\_env', C\rangle, ETH)}$$

dove:

- a è il nome dell'agente che esegue la chiamata;
- $path$ è la sequenza di identificatori di chiamata di procedura che individua l'ambiente in cui viene eseguita la chiamata;
- $call = P(p_{a_1}, \ \ldots \ , \ p_{a_n}, !p_{a_{n+1}}, \ \ldots \ , \ !p_{a_m})$ è la chiamata di procedura in cui i parametri passati per riferimento sono contrassegnati dal simbolo "!".
- ex_env_P è la funzione totalmente indefinita che denota l'ambiente locale generato dalla chiamata di procedura;
- $e' = ex\_env(path)\Big[ex\_env_P[v_1/p_{f_1}, \ldots, v_n/p_{f_n}, path_{n+1}/p_{f_{n+1}}, \ldots, path_{n+m}/p_{f_{n+m}}]/P\Big]$ è l'ambiente di esecuzione dell'agente, in cui all'identificatore di chiamata di procedura P viene associato il nuovo ambiente locale inizializzato con i parametri formali $p_{f_1} \ \ldots \ p_{f_{n+m}}$, a cui sono stati associati i relativi valori.

5.2.7 Operazioni sulle componenti

Le componenti sono strutture dati, caratterizzate da un tipo, su cui sono definite le operazioni di inserimento, lettura e cancellazione di un elemento.

Ad una componente viene associato un tipo durante la fase di definizione dell'architettura concreta, questo tipo viene utilizzato per realizzare le operazioni di inserimento, estrazione oppure cancellazione di un elemento dalla struttura dati tipata; il predicato $is\_type(c, \ t)$ è soddisfatto se l'argomento t corrisponde al tipo della componente c, viene quindi utilizzato per individuare il tipo associato ad una componente.

La semantica delle operazioni di accesso e modifica delle componenti, oltre che essere funzione del tipo della componente dipende anche dalla modalità con cui viene eseguita una operazione.
Ad esempio se la componente è di tipo lista l'inserimento di un elemento può avvenire in testa oppure in coda alla lista, tale comportamento viene specificato dalle modalità **head** oppure **tail**, mentre per una componente di tipo tupla la

modalità assume il significato di *posizione* all'interno della tupla.

Le modalità che una istruzione può assumere sono:

- Componente di tipo lista:
  - **head**: per l'inserimento, la lettura e la cancellazione di un elemento in testa ad una componente di tipo lista;
  - **tail**: per l'inserimento, la lettura e la cancellazione di un elemento in coda ad una componente di tipo lista.
- Componente di tipo insieme:
  - *el*: per la cancellazione dell'elemento *el* dall'insieme;
  - **NIL**: per l'inserimento e la lettura di un elemento. **NIL** è la modalità nulla.
- Componente di tipo tupla:
  - *pos*: per l'inserimento, la lettura e la cancellazione di un elemento in posizione *pos*;

Lettura di un elemento di una componente

L'esecuzione dell'istruzione di lettura di un elemento da una componente modifica l'ambiente di esecuzione dell'agente: l'istruzione **get_comp**(c, m, x) denota la lettura di un elemento dalla componente c con modalità m e l'inserimento di tale elemento nella variabile x.

Il tipo t della componente c viene individuato dal predicato $is\_type(c,\ t)$ e per individuare il *path* relativo alla dichiarazione della variabie viene utilizzata la relazione $\underset{ex\_env,path}{\overset{s\_var}{\longmapsto}}$.
La relazione $c \underset{t,m}{\overset{\mathcal{G}}{\longmapsto}} v$ rappresenta la lettura, secondo la modalità m, dell'elemento v, dalla componente c di tipo t. Se la componente non è istanziata con alcun valore, l'esecuzione dell'istruzione non modifica l'ambiente, quindi non modifica la variabile.
Ad esempio se la componente è di tipo lista e tale componente è "vuota" il valore ottenuto dalla relazione $\underset{t,m}{\overset{\mathcal{G}}{\longmapsto}}$ è **NIL**.
La definizione della relazione $\underset{t,m}{\overset{\mathcal{G}}{\longmapsto}}$ viene tralasciata in quanto rappresenta l'operazione di lettura su struttre dati già conosciute.

Le meta-regole seguenti caratterizzano la semantica dell'istruzione di lettura:

$$\frac{is\_type(c,\ t) \quad x \underset{e,path}{\overset{s\_var}{\longmapsto}} path' \quad c \underset{t,m}{\overset{\mathcal{G}}{\longmapsto}} v}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,get]}{\longmapsto} (\langle ex\_env', C\rangle, ETH)} \quad v \neq \mathbf{NIL}$$

$$\frac{is\_type(c,\ t) \quad x \underset{e,path}{\overset{s\_var}{\longmapsto}} path' \quad c \underset{t,m}{\overset{\mathcal{G}}{\longmapsto}} \mathbf{NIL}}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,get]}{\longmapsto} (\langle ex\_env, C\rangle, ETH)}$$

dove:

- a è il nome dell'agente che esegue la lettura dalla componente;
- $path$ è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita l'istruzione di lettura;
- $get \equiv \mathbf{get\_comp}(c,\ m,\ x)$;
- $c \in C$ è la componente di tipo t da cui viene prelevato l'elemento e C è l'insieme delle componenti dello stato dell'agente;
- ex_env è l'ambiente di esecuzione dell'agente in cui viene eseguita la lettura;
- $ex\_env' = ex\_env(path')\big[v/x\big]$ è il nuovo ambiente di esecuzione dell'agente in cui alla variabile x viene associato l'elemento letto.

Inserimento di un elemento in una componente

L'effetto dell'operazione d'inserimento di un elemento non modifica l'ambiente di esecuzione dell'agente come nel caso precedente, modifica invece la componente oggetto dell'inserimento: l'istruzione $\mathbf{put\_comp}(c,\ m,\ exp)$ valuta l'espressione exp, quindi la inserisce nella componente c con modalità m.

Come nel caso precedente il tipo t della componente c viene individuato dal predicato $is\_type(c,\ t)$, mentre l'inserimento del valore v ottenuto dalla valutazione di exp, avviene attraverso la relazione $v \underset{c,t,m}{\overset{\mathcal{P}}{\longmapsto}} c'$, in cui c è il nome della componente, t il tipo ed m la modalità di inserimento. Come nel caso precedente la relazione $\underset{c,t,m}{\overset{\mathcal{P}}{\longmapsto}}$ non viene definita.

La semantica dell'istruzione di inserimento è definita dalla meta-regola seguente:

$$\frac{is\_type(c,\ t) \qquad exp \underset{ex\_env,path}{\overset{e\_exp}{\longmapsto}} v \qquad v \underset{c,t,m}{\overset{\mathcal{P}}{\longmapsto}} c'}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,put]}{\longmapsto} (\langle ex\_env, C'\rangle, ETH)}$$

dove:

- a è il nome dell'agente che esegue l'inserimento nella componente;
- $path$ è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita l'istruzione di inserimento;
- $put \ \equiv$ **put_comp**$(c,\ m,\ exp)$;
- $C \ = \ \{\dots,\ c,\ \dots\}$ è l'insieme delle componenti dell'agente prima dell'inserimento dell'elemento, c è la componente di tipo t oggetto dell'istruzione di inserimento;
- $C' \ = \ \{\dots,\ c',\ \dots\}$ è il nuovo insieme di componenti dell'agente, ottenuto in seguito all'esecuzione dell'istruzione di inserimento. La componente c' è la componente c a cui è stato aggiunto v in accordo con la modalità specificata.

Eliminazione di un elemento da una componente

Anche in questo caso l'eliminazione di un elemento da una componente non modifica l'ambiente di esecuzione dell'agente. L'istruzione **del_comp**$(c,\ m)$ rimuove un elemento dalla componente c in accordo con la modalità m.

Il tipo t della componente c si ottiene dal predicato $is\_type(c,\ t)$ e la rimozione dell'elemento selezionato, in accordo con la modalità specificata, avviene tramite la relazione $\underset{t,m}{\overset{\mathcal{R}}{\longmapsto}}$.
In questo caso la modalità specifica una posizione all'interno della tupla se la componente è di tipo **tupla**, **head** oppure **tail** se è di tipo **list**, specifica un elemento dell'insieme se è di tipo **set**.

La semantica dell'istruzione di cancellazione di un elemento è definita dalla meta-regola seguente:

$$\frac{is\_type(c,\ t) \quad c \underset{t,m}{\overset{\mathcal{R}}{\longmapsto}} c'}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,rem]}{\longmapsto} (\langle ex\_env, C'\rangle, ETH)}$$

dove:

- a è il nome dell'agente che esegue la rimozione dell'elemento;
- $path$ è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita l'istruzione di eliminazione;
- $rem \equiv$ **del_comp**(c, m);
- $C = \{\ldots, c, \ldots\}$ è l'insieme delle componenti dell'agente prima dell'eliminazione dell'elemento selezionato attraverso la modalità specificata, c è la componente di tipo t oggetto dell'eliminazione;
- $C' = \{\ldots, c', \ldots\}$ è il nuovo insieme di componenti dell'agente, ottenuto in seguito all'eliminazione di un elemento da $c \in C$. La componente c' è c a cui è stato eliminato un elemento in accordo con la modalità specificata. Se prima di eseguire l'eliminazione la componente era vuota, c non subirà modifiche: $c' = c$.

5.2.8 Invio e ricezione di messaggi

Le istruzioni che permettono l'invio e la ricezione dei messaggi agiscono sull'etere (ETH) del sistema multi-agente aggiungendo e rimuovendo messaggi.
In questo modo gli agenti del sistema possono comunicare tra di loro, scambiarsi informazioni e richieste di servizi.
La struttura dati che costituisce ETH è un insieme di messaggi ordinato rispetto al loro ordine di arrivo, in modo che il primo messaggio inviato ad un particolare destinatario sia anche il primo ad essere ricevuto dall'agente cui è destinato.
ETH può essere, ad esempio, rappresentato da una lista in cui i messaggi vengono inseriti sempre in coda.
Quando un messaggio di un particolare destinatario deve essere ricevuto, la lista viene visitata scorrendola dalla testa verso la coda e quando viene trovato il messaggio riferito a tale destinatario, viene prelevato dalla lista.

Invio di un messaggio

L'invio di un messaggio avviene tramite l'istruzione **send**(r, c) che invia il messaggio contenuto in c al destinatario r.
L'effetto di questa istruzione è quello di aggiungere all'etere del sistema la tripla:

$$(mittente, destinatario, contenuto)$$

dove *mittente* è il nome dell'agente che invia il messaggio, *destinatario* è il nome dell'agente che lo riceve e *contenuto* rappresenta il contenuto del messaggio.

Il nome dell'agente che esegue l'invio del messaggio viene specificato nella *create_agent* che lo ha generato, mentre il nome del destinatario ed il contenuto del messaggio possono fare parte delle informazioni mantenute nelle componenti dell'agente e quindi essere espressi come valori, ad esempio come stringhe, oppure come variabili contenenti le opportune informazioni.
Un messaggio inviato a tutti gli agenti del sistema ha come destinatario la costante **ALL**.
Ad esempio l'istruzione:

send(**ALL**, "ciao mondo")

eseguita dall'agente "*A*", inserisce nell'etere il messaggio:

(A, **ALL**, "ciao mondo")

In questo modo ogni agente del sistema sarà in grado di ricevere il messaggio.

La meta-regola che definisce la semantica dell'istruzione che invia un messaggio è la seguente:

$$\frac{r \overset{e\_exp}{\underset{ex\_env.path}{\longmapsto}} receiver \quad c \overset{e\_exp}{\underset{ex\_env.path}{\longmapsto}} content}{(\langle ex\_env, C\rangle, ETH) \overset{[s,path,send]}{\longmapsto} (\langle ex\_env, C\rangle, ETH')}$$

dove:

- s è il nome dell'agente che invia il messaggio;
- *path* è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita l'istruzione che invia il messaggio;
- $send \equiv$ **send**(r, c);
- ETH è l'etere prima che il messaggio venga inviato;
- ETH' è l'etere aggiornato con il nuovo messaggio, inserito in ETH come ultimo spedito al destinatario r dall'agente s.

Ricezione asincrona di un messaggio

La ricezione asincrona di un messaggio avviene tramite l'istruzione **receive**(s, x) che associa alla variabile x il contenuto del messaggio inviato dal mittente s e rimuove il messaggio letto dall'etere.

Un agente può ricevere soltanto i messaggi contenuti nell'etere di cui è destinatario, tra questi può scegliere quelli che appartengono ad un particolare mittente, oppure utilizzando la costante **ANY** in sostituzione del nome del mittente, il messaggio selezionato prescinde dall'agente che lo ha inviato.

Questo tipo di ricezione viene detta asincrona perché se nell'etere non vi è nessun messaggio da parte del mittente specificato, con destinatario il nome dell'agente che esegue la ricezione, oppure con destinatario la costante **ALL**, l'istruzione associa la costante **NIL** alla variabile passata per valore:

$$\frac{s \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} s' \quad x \overset{s\_var}{\underset{ex\_env,path}{\longmapsto}} path'}{(\langle ex\_env, C\rangle, ETH) \overset{[r,path,receive]}{\longmapsto} (\langle ex\_env', C\rangle, ETH)} \quad (s',\ r,\ \cdot) \notin ETH \ \wedge\ (s',\ \mathbf{ALL},\ \cdot) \notin ETH$$

dove:

- $path$ è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita la ricezione del messaggio;
- r è il nome del destinatario del messaggio, oppure la costante **ALL**;
- s' è il nome del mittente del messaggio oppure la costante **ANY**;
- $receive \equiv \mathbf{receive}(s,\ x)$;
- ETH è l'etere del sistema;
- ex_env è l'ambiente di esecuzione dell'agente prima che venga eseguita l'istruzione di ricezione del messaggio;
- $ex\_env' \ = \ ex\_env(path')\big[\mathbf{NIL}/x\big]$ è l'ambiente di esecuzione dopo la ricezione del messaggio. Alla variabile x viene associato la costante **NIL**.

Se nell'etere è presente il messaggio desiderato tale messaggio viene letto, il suo contenuto associato alla variabile passata come argomento e quindi viene rimosso dall'etere, se il destinatario è diverso dalla costante **ALL**:

$$\frac{s \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} s' \quad x \overset{s\_var}{\underset{ex\_env,path}{\longmapsto}} path' \quad s' \overset{\mathcal{REC}}{\underset{r}{\longmapsto}} c}{(\langle ex\_env, C\rangle, ETH) \overset{[r,path,receive]}{\longmapsto} (\langle ex\_env', C\rangle, ETH')} \quad (s',\ r,\ \cdot) \in ETH \quad r \neq \mathbf{ALL}$$

Se il destinatario del messaggio consiste nella costante **ALL** il messaggio non viene rimosso in modo che possa essere ricevuto da tutti gli agenti del sistema:

$$\frac{s \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} s' \quad x \overset{s\_var}{\underset{ex\_env,path}{\longmapsto}} path' \quad s' \overset{\mathcal{REC}}{\underset{r}{\longmapsto}} c}{(\langle ex\_env, C\rangle, ETH) \overset{[r,path,receive]}{\longmapsto} (\langle ex\_env', C\rangle, ETH)} \quad (s',\ \mathbf{ALL},\ \cdot) \in ETH$$

Nelle meta-regole precedenti:

- *path* è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita la ricezione del messaggio;
- r è il nome del destinatario del messaggio, oppure la costante **ALL**;
- s' è il nome del mittente del messaggio oppure la costante **ANY**;
- $receive \equiv \textbf{receive}(s,\ x)$;
- ETH è l'etere se il messaggio non è stato rimosso;
- ex_env è l'ambiente di esecuzione dell'agente prima che venga eseguita l'istruzione di ricezione del messaggio;
- $ex\_env' = ex\_env(path')\big[content/x\big]$ è l'ambiente di esecuzione dopo la ricezione del messaggio. Alla variabile x viene associato il contenuto c del messaggio.
- ETH' è l'etere in cui è stato rimosso il messaggio letto in precedenza.

Ricezione sincrona di un messaggio

La ricezione sincrona di un messaggio differisce da quella asincrona quando nell'etere non è presente il messaggio desiderato: in tal caso l'istruzione di ricezione asincrona termina senza causare nessuna modifica nell'etere e nell'ambiente, mentre quella sincrona non termina ed attende l'arrivo del messaggio:

$$\frac{s \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} s' \quad x \overset{s\_var}{\underset{ex\_env,path}{\longmapsto}} path' \quad s' \overset{\mathcal{REC}}{\underset{r}{\longmapsto}} c}{(\langle ex\_env, C\rangle, ETH) \overset{[r,path,rec\_sync]}{\longmapsto} (\langle ex\_env', C\rangle, ETH')} \quad (s',\ r,\ \cdot) \in ETH \quad r \neq \textbf{ALL}$$

Anche in questo caso i messaggi con destinatario **ALL** non vengono rimossi dall'etere:

$$\frac{s \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} s' \quad x \overset{s\_var}{\underset{ex\_env,path}{\longmapsto}} path' \quad s' \overset{\mathcal{REC}}{\underset{r}{\longmapsto}} c}{(\langle ex\_env, C\rangle, ETH) \overset{[r,path,rec\_sync]}{\longmapsto} (\langle ex\_env', C\rangle, ETH)} \quad (s',\ \textbf{ALL},\ \cdot) \in ETH$$

Come nel caso precedente:

- *path* rappresenta la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita la ricezione del messaggio;
- r è il nome del destinatario del messaggio, oppure la costante **ALL**;
- s' è il nome del mittente del messaggio oppure la costante **ANY**;
- $receive \equiv \textbf{receive\_sync}(s,\ x)$;

- ETH è l'etere se il messaggio non viene rimosso;
- ex_env è l'ambiente di esecuzione dell'agente prima che venga eseguita l'istruzione di ricezione del messaggio;
- $ex\_env' = ex\_env(path')[content/x]$ è l'ambiente di esecuzione dopo la ricezione del messaggio. Alla variabile x viene associato il contenuto del messaggio.
- ETH' è l'etere in cui è stato rimosso il messaggio letto in precedenza.

5.2.9 Esecuzione di una azione

Le istruzioni viste fino a questo momento sono in grado di modificare lo stato interno dell'agente oppure l'etere del sistema multi-agente, ma non l'ambiente in cui sono collocati gli agenti.
Un agente può eseguire azioni il cui effetto si manifesta nell'ambiente del sistema, queste azioni sono eseguite tramite l'istruzione **exec**(a), in cui a è l'azione eseguita.

L'esecuzione di una azione consiste nell'inviare un messaggio con destinatario la costante speciale **ENV_AGT**, il quale può essere ricevuto soltanto dall'agente che costituisce l'ambiente.
Quando il messaggio viene ricevuto, l'agente ambiente lo processa e quindi inserisce i dati riguardanti gli effetti dell'esecuzione dell'azione nell'ambiente:

$$\frac{x \underset{ex\_env,path}{\overset{e\_exp}{\longmapsto}} action \quad \mathbf{send}(\mathbf{ENV\_AGT}, action)}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,exec]}{\longmapsto} (\langle ex\_env, C\rangle, ETH')}$$

dove:

- a è il nome dell'agente che esegue l'azione;
- $path$ è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita l'azione;
- $exec \equiv \mathbf{exec}(x)$;
- ETH è l'etere prima che avvenga l'inserimento del messaggio corrispondente all'esecuzione dell'azione $action$;
- ETH' è l'etere in cui è stato inserito il messaggio destinato all'agente ambiente.

Le azioni che vengono eseguite dall'agente non fanno parte del linguaggio che descrive l'architettura concreta, ma dipendono direttamente dal dominio applicativo caratterizzato dalle informazioni contenute nelle componenti dell'agente. Nell'esempio 4.4.1 le azioni eseguite dagli agenti ping-pong e homer sono: ($, stop), ($, dietrofront), (#(carico, false)) e cos'i via . Tali azioni vengono processate dall'engine dell'agente e quindi eseguite, alcune sono in grado di modificare solo lo stato interno dell'agente, altre invece hanno lo scopo di modificare l'ambiente del sistema e quindi l'engine le esegue tramite l'istruzione **exec**$(\cdot)$.

5.2.10 Percezione degli eventi

L'agente percepisce gli eventi che si manifestano nell'ambiente attraverso l'istruzione **percept**(x), la quale associa alla variabile x l'evento percepito.

L'insieme degli eventi ENV, in ogni istante dell'attività del sistema, contiene gli eventi che si manifestano, quindi se l'insieme contiene, ad esempio, n eventi è perché questi n eventi si stanno manifestando contemporaneamente.
Attraverso l'istruzione **percept**$(\cdot)$ l'agente percepisce un solo evento alla volta e tale percezione consiste nell'estrarre, senza cancellarlo, l'elemento da ENV:

$$\frac{x \overset{e\_exp}{\underset{e,path}{\longmapsto}} event}{(\langle ex\_env, C\rangle, ETH) \overset{[a,path,perc]}{\longmapsto} (\langle ex\_env', C\rangle, ETH)} \qquad event \in ENV$$

dove:

- a è il nome dell'agente che percepisce l'ambiente;
- $path$ è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita l'azione;
- $perc \equiv$ **percept**(x);
- $ex\_env' = e(path)\big[event/x\big]$ è l'ambiente di esecuzione dopo la percezione dell'evento: alla variabile x viene associato l'evento percepito.

5.2.11 Inserimento ed eliminazione degli eventi

L'agente che simula l'ambiente è il solo agente in grado di modificare direttamente i contenuto dell'ambiente ENV del sistema. Può inserire gli eventi che si manifestano come conseguenza dell'attività degli altri agenti e quando questi non hanno più ragione di esistere può eliminarli da ENV.

L'inserimento avviene tramite l'istruzione **put_event**(x) la quale inserisce l'evento x nell'insieme ENV:

$$\frac{x \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} event}{(\langle ex\_env, C\rangle, ENV) \overset{[a,path,p\_ev]}{\longmapsto} (\langle ex\_env, C\rangle, ENV')} \qquad ENV' = ENV \bigcup \{event\}$$

dove:

- a è il nome dell'agente che simula l'ambiente;
- $path$ è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguito l'inserimento del nuovo evento;
- $p\_ev \equiv$ **put_event**(x);
- ENV è l'insieme degli eventi che si manifestano nell'ambiente prima dell'inserimento dell'evento;
- ENV' è l'insieme degli eventi che si manifestano nell'ambiente dopo l'inserimento dell'evento;

Quando il manifestarsi di un evento viene meno, l'agente che simula l'ambiente è in grado di eliminarlo da ENV attraverso l'istruzione **remove_event**(x), in cui x è l'evento da rimuovere.
Ad esempio la scomparsa di un ostacolo dalla traiettoria del robot dell'esempio 4.4.1, comporta l'eliminazione dell'evento ("*ostacolo*", *distanza*) da ENV, dove "*ostacolo*" è la stringa che caratterizza la presenza di un ostacolo sul percorso del robot e *distanza* il valore relativo alla distanza del robot dall'ostacolo.

La semantica dell'istruzione di eliminazione di un evento da ENV è definita dalla meta-regola seguente:

$$\frac{x \overset{e\_exp}{\underset{ex\_env,path}{\longmapsto}} event}{(\langle ex\_env, C\rangle, ENV) \overset{[a,path,r\_ev]}{\longmapsto} (\langle ex\_env, C\rangle, ENV')} \qquad ENV' = ENV - \{event\}$$

dove:

- a è il nome dell'agente che simula l'ambiente;
- $path$ è la sequenza di identificatori di chiamate di procedura che individua l'ambiente in cui viene eseguita la rimozione dell'evento;
- $r\_ev \equiv$ **remove_event**(x);
- ENV è l'insieme degli eventi che si manifestano nell'ambiente prima della rimozione dell'evento;

- ENV' è l'insieme degli eventi che si manifestano nell'ambiente in seguito alla rimozione;

5.3 Semantica del MAS

La semantica del sistema multi-agente consiste nell'albero delle transizioni tra le configurazioni descritte nell'equazione 5.1.
La radice dell'albero rappresenta la configurazione iniziale del MAS, gli altri nodi sono le configurazioni relative agli aggiornamenti dello stato degli agenti, dell'etere oppure dell'ambiente. Gli archi rappresentano le transizioni delle configurazioni del MAS ottenute dalle transizioni tra gli stati degli agenti.

5.3.1 La configurazione iniziale del MAS

La configurazione iniziale del MAS rappresentata lo stato iniziale del sistema prima che l'esecuzione dell'engine cominci. Viene costruita in base alla definizione dell'architettura astratta, dell'architettura concreta, alle definizioni delle classi di agente ed alla creazione degli agenti.

La configurazione iniziale del MAS ha la forma di una n-upla:

$$[ENV_0,\ ETH_0,\ s_{0_{a_1}},\ \dots\ ,\ s_{0_{a_n}},\ s_{0_{env}}]$$

dove ENV_0, ETH_0, $s_{0_{a_1}} \dots$, $s_{0_{a_n}}$ e $s_{0_{env}}$ sono gli stati iniziali rispettivamente dell'ambiente, dell'etere, degli agenti del sistema ed infine dell'agente che simula l'ambiente.
Lo stato dell'ambiente e l'etere del sistema vengono inizializzati durante la fase di definizione della classe di agente.
Nell'esempio 4.4.1 ENV viene inizializzato con l'evento $(corrente, 2)$ e ETH con l'insieme vuoto di messaggi.
Lo stato di un agente (compreso quello dell'agente che simula l'ambiente) è una coppia il cui primo elemento è l'ambiente di esecuzione dell'agente ed il secondo l'insieme delle componenti (5.2). Quest'ultimo viene inizializzato sia nella fase di definizione della classe che nella creazione della sua istanza, mentre l'ambiente di esecuzione dell'agente viene inizializzato nella fase di definizione dell'architettura concreta, tramite l'inizializzazione delle variabili globali.
Nell'esempio 4.4.1 lo stato di ping-pong viene inizializzato con l'istanziazione della componente $plans\_component$ nella definizione della classe, con l'istanziazione delle componenti $beliefs\_component$ e $intentions\_component$ nella fase della creazione dell'agente; l'ambiente di esecuzione viene inizializzato nella creazione dell'architettura concreta dell'esempio 4.3.2.

5.3.2 Le transizioni del MAS

Il sistema multi-agente esegue una transizione tra configurazioni se uno dei suoi agenti esegue una transizione tramite la relazione $\overset{agt(a_i)}{\longmapsto}$:

$$\frac{(s_{a_i}, ETH) \overset{agt(a_i)}{\longmapsto} (s'_{a_i}, ETH')}{[ENV,\ ETH,\ s_{a_1},\ \ldots,\ s_{a_i},\ \ldots,\ s_{a_n},\ s_{env}] \overset{mas}{\longmapsto} [ENV,\ ETH',\ s_{a_1},\ \ldots,\ s'_{a_i},\ \ldots,\ s_{a_n},\ s_{env}]}$$

oppure se l'agente che simula l'ambiente esegue la transizione $\overset{env\_agt(a)}{\longmapsto}$:

$$\frac{(s_{env}, ENV, ETH) \overset{env\_agt(a)}{\longmapsto} (s'_{env}, ENV, ETH)}{[ENV,\ ETH,\ s_{a_1},\ \ldots\ s_{a_n}, s_{env}] \overset{mas}{\longmapsto} [ENV',\ ETH',\ s_{a_1},\ \ldots\ s_{a_n}, s'_{env}]}$$

Un'agente transisce da una *configurazione_agente* ad un'altra quando esegue un'istruzione dell'engine, l'effetto dell'istruzione modifica lo stato del MAS e quindi comporta una transizione ad una nuova configurazione.

Capitolo 6

Conclusioni

Questa tesi è articolata in due parti. Nella prima parte viene data una panoramica sulle diverse architetture per agenti reattivi, deliberativi e ibridi, quindi viene affrontata una ampia rassegna di linguaggi di specifica per sistemi multi-agente, che si conclude con un confronto tra di essi.
Nella seconda parte viene descritto un linguaggio in grado di specificare sistemi multi-agente eterogenei, cioè composti da agenti basati su architetture arbitrarie ed ognuno con un comportamento specifico, adeguato al proprio ruolo all'interno del sistema.
Il linguaggio presentato è dotato di una struttura che permette la distinzione tra architettura astratta e concreta.
Una architettura astratta descrive le caratteristiche dell'architettura che non dipendono dall'implementazione delle strutture dati utilizzate, attraverso la definizione delle sue componenti e delle procedure (non completamente specificate) che ne definiscono il funzionamento. Una architettura concreta completa le procedure dell'architettura astratta basando la loro definizione su dettagli implementativi e specifica come vengono implementate le componenti. In questo modo possono essere sviluppate diverse implementazioni della stessa architettura.
Definita una architettura concreta, alcune delle sue componenti possono essere istanziate con opportune informazioni (il programma dell'agente) che riguardano il dominio applicativo di una particolare classe di agenti. È in questa fase che, data una particolare architettura, un agente acquisisce le informazioni per svolgere una attività piuttosto che un'altra. Ad esempio fissata una architettura, le sue componenti possono essere istanziate in modo che l'agente sia in grado di pilotare un aereo oppure cercare informazioni di un certo tipo, navigando all'interno di internet.
Da ogni singola classe di agenti viene poi creata una istanza che rappresenta il singolo agente del sistema. In questa fase le rimanenti componenti dell'architettura vengono inizializzate con le informazioni relative allo stato iniziale dell'agente.

Lo stato iniziale evolverà durante l'attività dell'agente, sia in funzione degli eventi che si manifesteranno nell'ambiente in cui è collocato, sia in funzione dei messaggi che l'agente riceverà.

6.1 Osservazioni sul linguaggio proposto nella tesi

Il linguaggio proposto è in grado di specificare agenti che rispondono ai requisiti (non necessariamente a tutti) che caratterizzano la definizione debole di agente. Questi requisiti sono giá stati commentati nella sezione e sono:

- autonomia;
- reattività;
- proattività;
- capacità sociale.

L'autonomia di un agente consiste nel potere operare senza l'intervento di entità esterne e di avere il controllo sul proprio stato interno.
La capacità di un agente specificato tramite il linguaggio proposto, di operare senza l'intervento di entità esterne, dipende dalla definizione del proprio programma. La creazione dell'agente comporta l'attivazione dell'engine, pertanto l'attività dell'agente viene determinata dall'applicazione dell'engine al programma ed allo stato corrente. Il robot dell'esempio 4.4.1 se al momento della sua creazione si trova di fronte ad un elemento di fine corsa, comincia la sua attività caricando oppure scaricando il materiale, senza necessità di intervento da parte di alcuna entità esterna.
Nel linguaggio proposto lo stato dell'agente è rappresentato dall'informazione mantenuta all'interno delle componenti che, nell'architettura astratta, sono definite come **agent**. Il controllo dello stato interno dell'agente avviene tramite le istruzioni di accesso e modifica di tali componenti ed è determinato dall'applicazione dell'engine al programma: l'engine attraverso l'uso delle istruzioni primitive fornisce le procedure per leggere e modificare il contenuto delle componenti, il programma determina quali componenti e in che modo vengono modificate.
Nell'esempio 4.3.2 lo stato interno dell'agente è rappresentato dalle credenze e dalle intenzioni ed il programma, dai piani. L'engine fornisce le procedure, *get_belief* e *put_belief* per la lettura e la modifica delle credenze, ed il programma fornisce i piani che determinano quando una credenza deve esser letta e quando deve essere modificata.

La reattività dell'agente consiste nella capacità di percepire gli eventi che si manifestano nell'ambiente in cui è collocato e di reagire a tali eventi, eventualmente modificando il proprio comportamento.
Nel linguaggio proposto la percezione dell'ambiente avviene tramite apposite istruzioni primitive, la capacità di reagire a tali eventi dipende dal modo in cui queste operazioni vengono impiegate.
L'impiego di tali istruzioni è determinato dalle procedure che costituiscono l'engine, le quali applicate al programma determinano in quale modo l'agente reagisce. Anche in questo caso, come per l'autonomia, la capacità reattiva dell'agente viene agevolmente caratterizzata grazie alla possiblità di accedere direttamente alle informazioni che definiscono lo stato ed il programma dell'agente, tramite istruzioni primitive del linguaggio basate su una semantica che ne caratterizza a pieno il significato. Nell'esempio 4.4.1 il robot percepisce l'ostacolo che invade il percorso, e quindi reagisce modificando il proprio comportamento.

La proatività è la capcità di assumere un comportamento dipendente, non solo dagli eventi percepiti, ma anche dagli obiettivi che l'agente deve perseguire.
L'architettura dell'agente può essere definita in modo tale che una componente sia dedicata alla rappresentazione esplicita degli obiettivi, in tal caso le informazioni che istanziano la componente definiscono gli obiettivi che l'agente deve perseguire. In questo caso la capacità proattiva dell'agente nasce dal modo in cui gli obiettivi vengono manipolati; l'engine fornisce le operazioni di accesso e modifica agli obiettivi ed il programma dell'agente controlla tali accessi e modifiche.
Se gli obiettivi non sono rappresentati esplicitamente, l'architettura fornisce attraverso l'engine le operazioni per "interpretare" il programma dell'agente e la capacità di essere proattivo emerge dalle interazioni con le altre entità collocate nell'ambiente.
Nell'esempio 4.4.1 gli obiettivi del robot non sono rappresentati esplicitamente, essi emergono dall'interazione del robot con i fine corsa e dalle credenze che ha sullo stato dell'ambiente, in questo caso gli obiettivi sono: arrivare ad un fine corsa; caricare oppure scaricare il materiale; raggiungere il fine corsa opposto.

La capacità sociale dell'agente consiste nell'interazione tra gli agenti del sistema non soltanto attraverso gli eventi che possono manifestarsi nell'ambiente, ma anche attraverso uno scambio di messaggi.
Il linguaggio sviluppato in questa tesi ha un insieme di istruzioni primitive realizzate allo scopo di inviare e ricevere messaggi.
Ogni agente del sistema può decidere a quale altro agente inviare messaggi oppure di inviare messaggi "broadcast" per l'intero sistema. Un agente che riceve messaggi può decidere quale messaggio ricevere tra quelli di cui è destinatario, scegliendo un particolare mittente.

Anche in questo caso il comportamento dell'agente nell'inviare oppure ricevere messaggi è funzione dell'applicazione dell'engine al programma ed allo stato corrente dell'agente.

6.2 Confornto con altri linguaggi

Alcuni linguaggi descritti nel capitolo 2 non sono in grado di specificare agenti con architetture arbitrarie. Ad esempio dMARS e AgentSpeak(L) specificano agenti con architetture BDI, quindi sistemi multi-agente composti esclusivamente da agenti BDI.
In questo modo la realizzazione di un agente è necessariamente vincolata ad un particolare tipo di architettura, il che rende la specifica dell'agente più problematica e meno "naturale". Al contrario, come accade nel linguaggio proposto in questa tesi, la possibilità di descrivere una architettura ad hoc per il ruolo specifico di un agente, facilita la realizzazione corretta sia dell'agente che dell'intero sistema multi-agente.
Alcuni linguaggi descritti nel capitolo 2, quali ad esempio DESIRE, Concurrent MetateM e ConGolog sono in grado di specificare sistemi eterogeneni, i cui agenti non sono tutti basati su di una unica architettura. Questi linguaggi non hanno però una struttura che permette la definizione esplicita dell'architettura, non hanno quindi istruzioni specializzate nel manipolare gli elementi che la costituiscono. Anche in questo caso la semantica di tali linguaggi non è in grado di evidenziare il ruolo dell'architettura nella definizione del comportamento dell'agente, come invece accade nel linguaggio descritto in questa tesi.
Nel paragrafo 2.10.5 del capitolo 2 viene evidenziata la capacità dei linguaggi basati su qualità mentali, quali ad esempio di AGENT 0 e dei linguaggi descritti nelle sezioni 2.7 e 2.8, di descrivere in modo significativo il comportamento dell'agente. Questa capacità è dovuta alla distinzione tra le varie categorie mentali utilizzate nei linguaggi ed alla presenza di operazioni primitive, specifiche nell'accedere alle informazioni mantenute in tali categorie.
Il linguaggio proposto è in grado di descrivere il comportamento dell'agente in modo altrettanto significativo sia grazie alla distinzione tra le componenti dell'architettura, quindi dei ruoli associati alle informazioni in esse contenute, sia alla presenza di operazioni specifiche per la manipolazione dei dati che contengono.

Una ulteriore differenza tra il linguaggio proposto nella tesi ed i linguaggi trattati nella rassegna ed , riguarda la mancanza, in questi ultimi, di una netta distinzione tra i concetti di:

- *stato*;

- *programma*;
- *engine*.

Nel linguaggio proposto i tre concetti assumono un ruolo fondamentale nel determinare il comportamento di un agente.
Essi determinano un modello di riferimento, in cui viene caratterizzato il ruolo dell'architettura e dell'informazione che istanzia le sue componenti nel descrivere il comportamento dell'agente.
Questo fa sì che il linguaggio sia in grado di specificare il comportamento di un agente attraverso la descrizione delle operazioni, eseguite dall'engine, che manipolano il contenuto delle componenti: cioè il programma e lo stato corrente dell'agente.

Lo scopo del linguaggio non riguarda soltanto la specifica di un agente, ma la specifica di un sistema di agenti in grado di comunicare tra loro e di percepire l'ambiente in cui sono collocati.
A questo proposito nel linguaggio vengono esplicitamente definiti l'*etere*, l'*ambiente*, un insieme di istruzioni primitive per l'invio e la ricezione dei messaggi da/per l'etere ed un insieme di istruzioni primitive per la percezione e l'inserimento degli eventi nell'ambiente. Inoltre viene definito un sistema di transizioni in grado di descrivere l'evoluzione dell'intero sistema in funzione dell'esecuzione di una istruzione dell'engine di un singolo agente.
In altri linguaggi in grado di specificare sistemi con più agenti che interagiscono tra loro e con un'ambiente, le strutture per la ricezione e l'invio dei messaggi spesso si sovrappongono alle strutture per la percezione degli eventi. Ad esempio negli agenti dMARS e AgentSpeak(L) oppure Concurrent MetateM, la percezione degli eventi viene trattata nello stesso modo della ricezione dei messaggi, senza avere istruzioni specifiche per trattare gli uni o gli altri. Queste caratteristiche rendono la progettazione degli agenti che costituiscono il sistema meno modulare e quindi di più difficile realizzazione.

6.3 Sviluppi futuri

La simulazione dell'attività di un sistema multi-agente realizzato attraverso il linguaggio proposto, è basata su un sistema di transizione tra configurazioni del MAS.
Nel modello di transizioni del MAS, proposto in 5, una *configurazione_agente* evolve in conseguenza dell'esecuzione di una istruzione dell'engine. La "granularità

fine" considerata in questo modello permette la definizione di ulteriori modelli con "granularità meno fine":

- Un agente evolve non più in conseguenza dell'esecuzione di una istruzione, ma al termine dell'esecuzione di un intero ciclo dell'engine. In questo modello vengono tralasciati i dettagli relativi all'esecuzione di ogni singola istruzione; l'aggiornamento dell'ambiente di esecuzione dell'agente e delle informazioni contenute nelle componenti dell'architettura viene preso in considerazione soltanto al termine dell'esecuzione del ciclo.

- un ulteriore sistema di transizioni potrebbe essere basato su di un modello sincrono, cioè un modello che simula la transizione di una configurazione del MAS a quella successiva se tutti gli agenti del MAS hanno compiuto una transizione. In questo modo si avrebbe una visione del comportamento del sistema senza dover necessariamente valutare in dettaglio l'attività di ogni singolo agente.

Una direzione di ricerca sicuramente interessante è lo studio dei diversi modelli sopra proposti, utili per studiare le proprietà del sistema multi-agente in funzione della granularità scelta, tenendo conto dell'attività degli agenti al livello di dettaglio desiderato.

Un'altra possibile direzione di sviluppo di questo lavoro, si basa sul vantaggio che si oterrebbe nell'osservazione dell'attività del sistema multi-agente se quest'ultimo fosse eseguibile. In questo caso si potrebbe eseguire l'engine degli agenti del sistema per simularne l'attività cioè le interazione tra di loro e con l'ambiente.
Un modo per rendere animabile il programma che definisce un sistema multi-agente, potrebbe essere quello di tradurre le istruzioni primitive del linguaggio in un altro linguaggio eseguibile in grado di modellare sia il concetto di stato dell'agente che la capacità di interazione con le altre entità del sistema.
Un'altra soluzione potrebbe essere quella di implementare in un linguaggio eseguibile, le regole che definiscono le transizioni tra configurazioni di agenti. In questo modo data una rappresentazione, nel linguaggio eseguibile scelto, della configurazione iniziale associata al programma, la simulazione dell'attività del sistema si otterrebbe eseguendo le regole di transizione tra rappresentazioni di configurazioni.
In entrambe le proposte, la scelta del linguaggio eseguibile da utilizzare potrebbe essere $\mathcal{E}_{hhf}$ [Del97], in quanto i vantaggi offerti nell'ambito della specifica dei sistemi multi-agente eterogenei sono già stati verificati in [Are99]. $\mathcal{E}_{hhf}$ è un linguaggio appartenente al paradigma della programmazione logica lineare in cui la nozione di stato di un programma ha una corrispondenza naturale con quella di stato dell'agente. Per questi motivi $\mathcal{E}_{hhf}$ è già stato utilizzato per modellare sia

aspetti di concorrenza tra agenti che aspetti riguardanti l'aggiornamento del loro stato [BDM+99a] e [BDM+99b].

Appendice A

Sintassi formale del linguaggio

Nella b.n.f. del linguaggio proposto, il simbolo * indica la ripetizione di zero o più volte della categoria sintattica a cui è applicato. Mentre nella definizione della categoria sintattica PROG il simbolo "ε" rappresenta il programma costituito da nessuna istruzione.

PROGRAM ::= ABSTRACT_ARCHITECTURE;
  CONCRETE_ARCHITECTURE; MAS.

ABSTRACT_ARCHITECTURE ::=
  abstract_architecture {ARCH_NAME}
  {ABS_COMPONENTS; PROCS; ENGINE}

CONCRETE_ARCHITECTURE ::=
  architecture {ARCH_NAME} is a {ARCH_NAME}
  {COMPONENTS; INIT_GLOBAL_VARS; PROCS }

MAS ::=
  class_agent {ARCH_NAME, CLASS_AGENT_NAME}
  {INIT_COMPONENTS; INIT_ENV_ETH; AGENTS}

ABS_COMPONENTS ::= components{DEF_ABS_COMPS*}

DEF_ABS_COMPS ::= COMP_NAME;

PROCS ::= {DEF_PROC*}

DEF_PROC ::= PROCEDURE_NAME(PARAMS*) {PROG}

PARAMS ::= VARS | !VARS | EXP

PROG ::=
ε | INSTRUCTION |
INSTRUCTION ";" PROG |
INSTRUCTION "‖" PROG |
INSTRUCTION "|" PROG |
if BOOL **then** PROG **else** PROG **endif**
while BOOL **do** PROG **endwhile**

INSTRUCTION ::=
get_comp(COMP_NAME, MODE, VAR_NAME) |
put_comp(COMP_NAME, MODE, EXP) |
del_comp(COMP_NAME, MODE) |
send(RECEIVER, CONTENT) |
receive(SENDER, CONTENT) |
receive_sync(SENDER, CONTENT) |
put_event(EVENT) |
remove_event(EVENT) |
percept(EVENT) |
decl VARS |
VARS := EXP

MODE ::= **NIL**| **head**| **tail**| POS | VARS | EXP

POS ::= NAT

EXP ::=
EXP_LIST |
EXP_STACK |
EXP_QUEUE |
EXP_SET |
EXP_TUPLA |
gethlist(EXP_LIST) |
gettlist(EXP_LIST) |
top_stack(EXP_STACK) |
getqueue(EXP_QUEUE) |
getset(EXP_SET) |
puttupla(EXP_TUPLA, POS, EXP) |
BOOL |

INT |
REAL |
STRING |
NIL|
VARS

EXP_LIST ::=
e_l |
inshlist(EXP_LIST, EXP) |
instlist(EXP_LIST, EXP) |
delhlist(EXP_LIST) |
deltlist(EXP_LIST)

EXP_STACK ::=
e_s |
put_stack(EXP_STACK, EXP) |
pop_stack(EXP_STACK)

EXP_QUEUE ::=
e_q |
insqueue(EXP_QUEUE, EXP) |
delqueue(EXP_QUEUE)

EXP_SET ::=
∅ |
insert(EXP_SET, EXP) |
removeset(EXP_SET, EXP)

EXP_TUPLA ::=
e_t(NAT) |
gettupla(EXP_TUPLA, POS)

COMPONENTS ::= components{DEF_COMP*}

DEF_COMP ::= COMP_NAME : TYPE;

ENGINE ::= engine{DEF_GLOBAL_VAR*
PROCEDURE_NAME(PARAMS*)*};

DEF_GLOBAL_VAR ::= **decl** VARS;

INIT_COMPONENTS ::= {INIT_COMP*};

INIT_COMP ::= init_comp {DEF_INIT_COMP*};

DEF_INIT_COMP ::= COMP_NAME [VALUE*];

INIT_ENV_ETH ::= init_ENV[EVENT*]; init_ETH[MESSAGGES*]

AGENTS ::= DEF_AGENTS*

DEF_AGENTS ::=
create_agent(CLASS_AGENT_NAME, AGENT_NAME)
{DEF_INIT_COMP*}

INIT_GLOBAL_VARS ::= init_global_vars{INIT_VARS *}

INIT_VARS ::= VARS := EXP;

EVENT ::= (ALPHANUMERIC_STRING, VALUE)

TYPE ::= **int**| **real**| **string**| **list**| **stack**| **queue**| **set**| **tupla**

MESSAGGES ::= (SENDER, RECEIVER, CONTENT)

ARCH_NAME ::= ALPHANUMERIC_STRING

CLASS_AGENT_NAME ::= ALPHANUMERIC_STRING

COMP_NAME ::= ALPHANUMERIC_STRING

SENDER ::= any | ALPHANUMERIC_STRING

RECEIVER ::= all | ALPHANUMERIC_STRING

CONTENT ::= ALPHANUMERIC_STRING

AGENT_NAME ::= ALPHANUMERIC_STRING

PROCEDURE_NAME ::= ALPHANUMERIC_STRING
VARS ::= ALPHANUMERIC_STRING

Appendice B

Esempio di sistema multi-agente

abstract_architecture {bdi}
{
    components
    {
        class plans_component;
        agent beliefs_component;
        agent intentions_component
    };

    procedures
    {
        percept_event()
        {
            percept_event_from_extern()
        };

```
plan_triggering_()
{
 decl event_trigger;
 decl plan_trigger;
 decl plan;
 decl aux_plans;
 decl empty;

 aux_plans := e_s;
 is_empty_event(!empty);
 if empty then skip
 else
    get_event();
    get_event_trigger(!event_trigger);
    is_empty_plans_component(!empty);
    while not(empty) do
       get_plans_component(!plan);
       get_plan_trigger(plan, !plan_trigger);
       if plan_trigger = event_trigger then
          put_triggered_plans(!plan)
       else  skip endif;
       aux_plans := put_stack(aux_plan, plan);
       del_plans_component(plan);
       is_empty_plans_component(!empty)
    endwhile
    while not(aux_plans = e_s) do
       put_plans_component(top_stack(aux_plans));
       aux_plans := pop_stack(aux_plans)
    endwhile
 endif
};
```

```
plan_selection()
{
 decl plan;
 decl condition;
 decl res;

 is_empty_triggered_plans(!empty);
 while not(empty) do
    get_triggered_plans(!plan);
    get_cond(plan, !condition);
    test_cond(condition, !res);
    if res then put_selected_plans(plan)
    else del_triggered_plans(plan) endif
 endwhile
};

upgrade_intentions_component()
{
 decl plan;
 decl empty;

 is_empty_selected_plans(!empty);
 if not(empty) then
    select_plan(!plan);
    put_intentions_component(gettupla(plan,3))
 else skip
};
```

```
    exec_intention()
    {
     decl actions_list;
     decl action;
     decl empty;

     is_empty_intentions_component(!empty);
     if empty then skip
     else
        get_intentions_component(!actions_list);
        get_action(actions_list, !action);
        exec_action(action)
     endif
    };


  engine
  {
    decl event;
    decl selected_event;
    decl triggered_plans;
    decl scelted_plans;

    while true do
       percept_event();
       plan_triggering();
       plan_selection();
       upgrade_intentions_component();
       exec_intention()
    endwhile
  }
};
```

```
architecture {bdi_1} is a {bdi}
{
    components
    {
        plans_component: list;
        beliefs_component: list;
        intentions_component: set;
    };

    init_global_vars
    {
        event = e_q;
        selected_event = e_t(2);
        triggered_plans = e_l;
        selected_plans = e_l;
    };

    procedures
    {
   /* procedure sulla variabile globale event */
        percept_event_from_extern()
        {
            decl sender;
            decl e_1;
            decl e_2;

            (
                percept(e_1);
                event := insqueue(event, e_1)
            ) ||
            (
                get_beliefe_component("sender", !sender);
                receive(sender, e_2);
                event := insqueue(event, e_1)
            )
        }
```

```
get_event()
{
   selected_event := getqueue(event);
   event := delqueue(selected_event, event)
};

is_empty_event(res)
{
   if (event = e_q) or(event = NIL) then res := true
   else res := false endif
};

get_event_trigger(trigger)
{
   trigger := gettupla(selected_event, 1)
};

get_event_value(value)
{
   value := gettupla(selected_event, 2)
};
/* procedure sulla variabile globale triggered_plans */
put_triggered_plans(plan)
{
   triggered_plans := inshlist(triggered_plans, plan)
};

get_triggered_plans(plan)
{
   plan := gethlist(triggered_plans)
};

del_triggered_plans(plan)
{
   triggered_plans := delhlist(triggered_plans)
};
```

```
is_empty_triggered_plans(res)
{
  if (triggered_plans = e_l) or(triggered_plans = NIL) then
    res := true
  else res := false endif
};

/* procedure sulla variabile globale selected_plans */
put_selected_plans(plan)
{
  selected_plans := inshlist(selected_plans, plan)
};

isempty_selected_plans(res)
{
  if (selected_plans = e_l) or (selected_plans = NIL) then res := true
  else res := false endif
};

select_plan(plan)
{
  plan := gethlist(selected_plans)
};

/* procedure sui piani */
get_plan_trigger(plan, plan_trigger);
{
  plan_trigger := gettupla(plan, 1)
};

get_cond(plan, condition);
{
  condition := gettupla(plan, 2)
};
```

```
test_cond(condition, res);
{
  decl data;
  decl event_value;

  get_belief(gettupla(condition, 2), !data);
  case gettupla(condition, 1)
    soglia_min: get_event_value(!event_value);
                if event_value < data then  res := true
                else res := false endif
    soglia_max: get_event_value(!event_value);
                if event_value > data then  res := true
                else res := false endif
    ?: if data then  res := true
       else res := false endif
  endcase
};
/* procedure sulla componente plans_component */
get_plans_component(plan)
{
  get_comp(plans_component, head, plan)
};

put_plans_component(plan)
{
  put_comp(plans_component, head, plan)
};

del_plans_component(plan)
{
  del_comp(plans_component, head)
};

is_empty_plans_component(res)
{
  decl plan;

  get_comp(plans_component, head, plan);
  if plan = e_t(3) then res := false
  else res:= true endif
};
```

```
/* procedure sulla componente beliefe_component */
  get_belief(id, data)
  {
    decl bel;
    decl aux_bel;
    decl empty;

    aux_bel := e_s;
    is_empty_beliefe_component(!empty);
    while not(id = gettupla(bel, 1)) andnot(empty) do
      get_comp(beliefe_component, head, bel);
      if id = gettupla(bel, 1) then
        data := gettupla(bel, 2)
      else
        del_comp(beliefe_component, head);
        aux_bel := put_stack(aux_bel, bel);
        is_empty_beliefe_component(!empty)
      endif
    endwhile;
    if empty then data := NIL else skip endif;
    while not(aux_bel = e_s) do
      put_comp(beliefe_component, head, top_stack(aux_bel));
      aux_bel:= pop_stack(aux_bel)
    endwhile
  };
```

```
put_beliefe_component(belief)
{
  decl id;
  decl bel;
  decl aux_bel;
  decl empty;

  aux_bel := e_s;
  id := gettupla(belief,1);
  is_empty_beliefe_component(!empty);
  if not(empty)
  then get_comp(beliefe_component, h, bel);
    while not( id = gettupla(bel,1)) and not(empty) do
      aux_bel:= put_stack(aux_bel, bel);
      del_comp(beliefe_component, h);
      is_empty_beliefe_component(!empty);
      if empty then skip
      else get_comp(beliefe_component, h, bel) endif
    endwhile;
    if empty then put_comp(beliefe_component, h, belief)
    else del_comp(beliefe_component, h);
      put_comp(beliefe_component, h, belief)
    endif
  else put_comp(beliefe_component, h, belief)
  endif;
  while not(aux_bel = e_s) do
    put_comp(beliefe_component, h, top_stack(aux_bel));
    aux_bel:= pop_stack(aux_bel);
  endwhile;
}

isempty_beliefe_component(res)
{
  decl belief;

  get_comp(beliefe_component, head, belief);
  if belief = e_t(3) then res := true
  else res:= false endif
}
```

```
/* procedure sulla componente intentions_component */
   put_intentions_component(actions_list)
   {
     put_comp(intentions_component, NIL, actions_list)
   }

   get_intentions_component(actions_list)
   {
     get_comp(intentions_component, NIL, actions_list)
   }

   isempty_intentions_component(res)
   {
     decl intention;

     get_comp(intrentions_component, head, intrention);
     if intrention = e_lthen res := true
     else res:= false endif
   }

   get_action(actions_list, action);
   {
     decl new_plan;

     if not(actions_list = e_l)          action := gethlist(actions_list);
     del_comp(intentions_component, actions_list);
     new_actions_list := delhlist(actions_list);
     put_comp(intentions_component, new_actions_list, ;)
   }
```

```
      exec_action(action);
      {
        decl receiver;

        case gettupla(action, 1)
          $: exec(gettupla(action,2));
          send: get_belief("receiver", !receiver);
                send(receiver, gettupla(action,2));
          #: put_beliefe_component(gettupla(action,2))
        endcase
      }
    }
};



architecture {bdi_2} is a {bdi}
{
    components
    {
      plans_component: list;
      beliefs_component: list;
      intentions_component: set;
    };

    init_global_vars
    {
      event = e_t(2);
      selected_event = e_t(2);
      triggered_plans = e_l;
      selected_plans = ∅;
    }
    procedures
    {
   /* procedure sulla variabile globale event */
      percept_event_from_extern()
      {
        percept(event)
      }
```

```
get_event()
{
   selected_event := event
}

is_empty_event(res)
{
   if (event = e_t(2)) or(event = NIL) then res := true
   else res := false endif
}
get_event_trigger(trigger)
{
   trigger := gettupla(selected_event, 1)
}
get_event_value(value)
{
   value := gettupla(selected_event, 2)
}
/* procedure sulla variabile globale triggered_plans */
put_triggered_plans(plan)
{
   triggered_plans := inshlist(triggered_plans, plan)
}

get_triggered_plans(plan)
{
   plan := gethlist(triggered_plans)
}

del_triggered_plans(plan)
{
   triggered_plans := delhlist(triggered_plans)
}

is_empty_triggered_plans(res)
{
   if (triggered_plans = e_l) or(triggered_plans = NIL) then
      res := true
   else res := false endif
}
```

```
/* procedure sulla variabile globale selected_plans */
   put_selected_plans(plan)
   {
      selected_plans := insert(selected_plans, plan)
   }

   isempty_selected_plans(res)
   {
      if (selected_plans = ∅) or (selected_plans = NIL) then res := true
      else res := false endif
   }

   select_plan(plan)
   {
      plan := getset(selected_plans)
   }

/* procedure sui piani */
   get_plan_trigger(plan, plan_trigger);
   {
      plan_trigger := gettupla(plan, 1)
   }

   get_cond(plan, condition);
   {
      condition := gettupla(plan, 2)
   }
```

```
test_cond(condition, res);
{
  decl data;
  decl event_value;

  get_belief(gettupla(condition, 2), !data);
  case gettupla(condition, 1)
    soglia_min: get_event_value(!event_value);
                if event_value < data then  res := true
                else res := false endif
    soglia_max: get_event_value(!event_value);
                if event_value > data then  res := true
                else res := false endif
    ?: if data then  res := true
       else res := false endif
  endcase
}
/* procedure sulla componente plans_component */
get_plans_component(plan)
{
  get_comp(plans_component, head, plan);
}

put_plans_component(plan)
{
  put_comp(plans_component, head, plan);
}

del_plans_component(plan)
{
  del_comp(plans_component, head)
}

is_empty_plans_component(res)
{
  decl plan;

  get_comp(plans_component, head, plan);
  if plan = e_t(3) then res := false
  else res:= trueendif
}
```

```
/* procedure sulla componente beliefe_component */
   get_belief(id, data)
   {
      decl bel;
      decl aux_bel;
      decl empty;

      aux_bel := e_s;
      is_empty_beliefe_component(!empty);
      while not(id = gettupla(bel, 1)) andnot(empty) do
         get_comp(beliefe_component, head, bel);
         if id = gettupla(bel, 1) then
            data := gettupla(bel, 2);
         else
            del_comp(beliefe_component, head);
            aux_bel := put_stack(aux_bel, bel);
            is_empty_beliefe_component(!empty)
         endif
      endwhile
      if empty then data := NIL else skip endif;
      while not(aux_bel = e_s) do
         put_comp(beliefe_component, head, top_stack(aux_bel));
         aux_bel:= pop_stack(aux_bel);
      endwhile
   }
```

```
put_beliefe_component(belief)
{
  decl id;
  decl bel;
  decl aux_bel;
  decl empty;

  aux_bel := e_s;
  id := gettupla(belief,1);
  is_empty_beliefe_component(!empty);
  if not(empty)
  then get_comp(beliefe_component, h, bel);
    while not( id = gettupla(bel,1)) and not(empty) do
      aux_bel:= put_stack(aux_bel, bel);
      del_comp(beliefe_component, h);
      is_empty_beliefe_component(!empty);
      if empty then skip
      else get_comp(beliefe_component, h, bel) endif
    endwhile;
    if empty then put_comp(beliefe_component, h, belief)
    else del_comp(beliefe_component, h);
      put_comp(beliefe_component, h, belief)
    endif
  else put_comp(beliefe_component, h, belief)
  endif;
  while not(aux_bel = e_s) do
    put_comp(beliefe_component, h, top_stack(aux_bel));
    aux_bel:= pop_stack(aux_bel);
  endwhile;
}

isempty_beliefe_component(res)
{
  decl belief;

  get_comp(beliefe_component, head, belief);
  if belief = e_t(3) then res := false
  else res:= trueendif
}
```

```
/* procedure sulla componente intentions_component */
   put_intentions_component(actions_list)
   {
      put_comp(intentions_component, , actions_list)
   }

   get_intentions_component(actions_list)
   {
      get_comp(intentions_component, , actions_list)
   }

   isempty_intrentions_component(res)
   {
      decl intention;

      get_comp(intrentions_component, head, intrention);
      if intrention = e_lthen res := true
      else res:= false endif
   }

   get_action(actions_list, action);
   {
      decl new_plan;

      if not(actions_list = e_l)
      action := gethlist(actions_list);
      del_comp(intentions_component, actions_list);
      new_actions_list := delhlist(actions_list);
      put_comp(intentions_component, new_actions_list, ;)
   }
```

```
        exec_action(action);
        {
          decl receiver;

          case gettupla(action, 1)
            $: exec(gettupla(action,2));
            send: get_belief("receiver", !receiver);
                  send(receiver, gettupla(action,2));
            #: put_beliefe_component(gettupla(action,2))
          endcase
        }
      }
};
```

```
MAS[1]
{
   class_agent {bdi_1}, {robot}
   {
    init_comp plans_component
     [
       (ostacolo, (soglia_min, distanza), ⟨($, stop)⟩),

       (ostacolo, (soglia_min, distanza), ⟨($, dietrofront),
            ($, riparti)⟩),

       (fine_corsa, (?, carico), ⟨(#, (carico, false)),
            ($, stop), ($, scarica), ($, dietrofront),
            ($, riparti), (#, (scarico, true))⟩),

       (fine_corsa, (?, scarico), ⟨(#, (scarico, false)),
            ($, stop), ($, scarica), ($, dietrofront),
            ($, riparti), (#, (carico, true))⟩)

       (pericolo, (?, stop_periclo), ⟨($, stop)⟩)
     ];
```

[1] la sequenza di azioni dei piani $a_1, \ldots, a_n$ viene abbreviata con $\langle a_1, \ldots, a_n \rangle$, il cui significato è : **inshlist**(...**inshlist**(**e_l**, a_1), ... a_n). La tupla di dimensione n viene abbreviata con $(e_1, \ldots, e_n)$, il cui significato è : **puttupla**(...**puttupla**(**e_t**, e_1), ... e_n)

```
};

class_agent {bdi_2}, {monitor}
{
  init_comp plans_component
    [
     (corrente, (soglia_max, s_corrente), (send, pericolo)),

     (temperatura, (soglia_max, s_temp), ($, attiva_raffreddamento))

     (temperatura, (soglia_max, s_temp), (send, pericolo))
    ];

  e = e_t(2);
  azione = "";
  lista_piani = e_l;
};

  init_ENV [(corrente,2)];
  init_ETH [∅];

  create_agent(robot, ping-pong)
  {
    init_comp beliefs_component
      [
       (soglia_distanza, 10),
       (carico, false),
       (scarico, true),
       (sender, homer),
       (stop_pericolo, true)
      ];
      init_comp intentions_component [∅];
  };
```

```
create_agent(monitor, homer)
{
  init_comp beliefs_component
    [
     (soglia_corrente, 3),
     (soglia_temperatura, 5),
     (receiver, ping-pong)
    ];
  init_comp intentions_component [∅];
};
}
```

Bibliografia

[Are99] A. Aretti. Semantica di sistemi multi-agente in logica lineare. Master's thesis, DISI – Università di Genova, Genova, Italy, 1999.

[BDM+99a] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic programming & Multi-Agent Systems: a Synergic Combination for Applications and Semantics. In *The Logic Programming Paradigm. A 25-Year Perspective.* Springer-Verlag, 1999.

[BDM+99b] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Multi-Agent Systems Development as a Software Engineering Enterprise. In *Proc. of PADL*, San Antonio, Texas, January 1999. Springer Verlag. LNCS 1551.

[BEL95] Claire Beyssade, Patrice Enjalbert, and Claire Lefevre. Cooperating Logical Agents. In *Intelligent Agents II*, number 1037 in Lecture Notes in Artificial Intelligence, 1995.

[BKJT95] Frances Brazier, Barbara Dunin Keplcz, Nick R. Jennings, and Jan Treur. Formal Specification of Multi-Agent Systems: a Real-World Case. First International Conference on Multi-Agent Systems (ICMAS'95), pages 25–32, Gune 1995.

[Del97] G. Delzanno. *Logic & Object–Oriented Programming in Linear Logic.* PhD thesis, Università di Pisa, Dipartimento di Informatica, 1997.

[dKLW98] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMARS. In Rao and Wooldridge (eds.), editors, *Intelligent Agents IV*, number 1365 in Lecture Notes in Artificial Intelligence, pages 155–176. Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages, Springer-Verlag, 1998.

[Fis92] Michael Fisher. Normal Form for First-Order Temporal Formulae. Number 607 in Lecture Notes in Artificial Intelligence, Saratoga Springs, New York, June 1992. Proceeding of Eleventh International Conference on Automated Deduction (CADE), Springer-Verlag.

[Fis93] Michael Fisher. Towards a Semantics for Concurrent MetateM. In *Executable Modal and Temporal Logics*, Lecture Notes in Artificial Intelligence, pages 86–102. chambery, August 1993.

[GLL99] G. De Giacomo, Y. Lespérance, and H.J. Levesque. ConGolog, a concurrent programming language based on the situation calculus: foundations. Submitted, February 1999.

[HdBvdHjCM98] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John jules Ch. Meyer. Agent Programming Language. In M. J. Woldridge Munindar P. Singh, Anand Rao, editor, *Intelligent Agents IV*, number 1365 in Lecture Notes in Artificial Intelligence, pages 215–229, 1998.

[KG96] David Kinny and Michael Georgeff. Modelling and Design of Multi-Agent Systems. In *Intelligent Agents III*, number 1193 in Lecture Notes in Artificial Intelligence, 1996.

[Kra94] J. Kramer. Distributed Software Engineering. In *Proc. of the 16th International Conference on Software Engineering*, pages 253–266, Sorrento, Italy, 1994. IEEE Computer Society Press.

[KS96] R. Kowalsky and F. Sadri. Towards a Unified Agent Architecture that Combines Rationality with Reactivity. In *Proc. of International Workshop on Logic in Databases*, San Miniato, Italy, 1996. Springer-Verlag.

[LLFL+95] Y. Lespérance, H. Levesque, D. Marcu F. Lin, R. Reiter, , and R. Scherl. Foundations of a Logical Approach to Agent Programming. In *Intelligent Agents Volume II – Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, Lecture Notes in Artificial Intelligence, pages 331–346, 1995.

[MTF97] M. Mulder, J. Treur, and M. Fisher. Agent Modelling in Concurrent METATEM and DESIRE. In *Intelligent Agents IV*. Springer-Verlag, 1997. LNAI 1365.

[Mul97a] M. Mulder. Comparing Agents Models in Concurrent METATEM and DESIRE. Msc thesis, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1997.

[Mul97b] Jorg P. Muller. Control Architectures for Autonomous and Interacting Agents: A Survey. In Wayne Wobcke Laurence Cavedon, Anand Rao, editor, *Intelligent Agent Systems: Teoretical and Pratical Issues*, number 1209 in Lecture Notes in Artificial Intelligence, Cairn Australia, August 1997. Berlin Springer.

[Rao96] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away*, pages 42–55. Springer-Verlag, 1996. LNAI 1038.

[SdAMP96] M. Schroeder, I. de Almeida Mòra, and L. M. Pereira. A Deliberative and Reactive Diagnosis Agent Based on Logic Programming. In *Intelligent Agents III*, pages 293–307. Springer-Verlag, 1996. LNAI 1193.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial intelligence*, pages 51–92, 1993.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2), 1995.

[Woo97] M. Wooldridge. Agent-based Software Engineering. *IEE Proc. of Software Engineering*, 144(1), 1997.