**Università degli Studi di Genova**
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Anno Accademico 2003/2004

Tesi di Laurea

# Coo-AgentSpeak: un linguaggio per agenti deliberativi e cooperativi

Candidato
**Daniela Demergasso**

Relatore
**Dott. Davide Ancona**

Correlatore
**Dott. Giorgio Delzanno**

**Dott.ssa Viviana Mascardi**

# Indice

# Introduzione

Negli ultimi anni, nel campo della Tecnologia dell'Informazione (Information Tecnology, IT), si è assistito all'affermarsi di sistemi basati sugli agenti. Molti credono che quello basato sugli agenti sia il più importante nuovo paradigma per lo sviluppo del software successivo a quello orientato agli oggetti.

I concetti tipici degli agenti intelligenti hanno trovato credito nell'ambito di diverse sotto-discipline della Tecnologia dell'Informazione tra cui reti di calcolatori, ingegneria del software, programmazione orientata agli oggetti, intelligenza artificiale, interazione uomo-macchina, sistemi distribuiti e concorrenti, sistemi mobili, telematica, lavoro cooperativo supportato dal calcolatore, sistemi di controllo e commercio elettronico.

I sistemi basati su agenti avranno un impatto consistente sulla società del futuro e questo fatto è accreditato dall'esistenza del programma per le Information Society Tecnologies della Commissione Europea.

Questo programma prevede che, in futuro piuttosto vicino, computer e reti saranno integrati nell'ambiente in cui operano quotidianamente per rendere accessibili una moltitudine di servizi e applicazioni attraverso l'uso di interfacce utente di facile utilizzo. Questa visione di *ambient intelligence* (intelligenza ambientale) mette l'utente al centro degli sviluppi futuri e mira alla realizzazione di una società basata sulla conoscenza a disposizione di tutti.

Secondo la nota definizione di N. R. Jennings, M. Wooldridge e K. Sycara in [39] si ha che

*Un agente è un sistema computazionale situato in un ambiente e capace di azioni flessibili e autonome per raggiungere i suoi obietti di progetto.*

In questo contesto, *essere situato* significa che l'agente riceve input sensoriali dal suo ambiente e che esso può compiere azioni che cambiano l'ambiente in diversi modi.

*Autonomo* significa che il sistema dovrebbe essere capace di agire senza l'intervento diretto di umani o di altri agenti e dovrebbe avere il controllo sulle proprie azioni e sul proprio stato interno. Altri usano questo con-

cetto in un senso più forte associandolo a sistemi che possono apprendere dall'esperienza.

Per *flessibile* si intende un sistema che è:

- *Reattivo.* Gli agenti sono capaci di percepire il loro ambiente e di rispondere in modo tempestivo ai cambiamenti che si verificano in esso.

- *Pro-attivo.* Gli agenti sono capaci di esibire un comportamento opportunistico diretto all'obiettivo (goal-directed) e di prendere l'iniziativa quando appropriato.

- *Sociale.* Gli agenti intelligenti sono capaci di interagire, al momento appropriato, con altri agenti o con umani al fine di risolvere i propri problemi e auitare gli altri nelle loro attività.

Spesso agli agenti vengono attribuite anche altre proprietà, come ad esempio:

- *mobilità*: la capacità di spostarsi fra i nodi di una rete;

- *apprendimento*: la capacità di imparare per migliorare le proprie prestazioni.

Spesso gli agenti sono modellati per mezzo di concetti cognitivi basati su stati mentali come credenze (beliefs), conoscenze (knowledge), desideri (desires), intenzioni (intentions). L'uso di questi concetti rende più semplice la descrizione di un'entità complessa quale un agente è: se l'entità complessa fosse descritta in virtù delle proprie componenti e delle dipendenze funzionali che intercorrono tra loro, la sua descrizione risulterebbe solitamente più complicata rispetto ad una descrizione data tramite attitudini mentali.

Tra i vari modelli cognitivi usati per la descrizione di agenti uno dei più noti è il modello BDI (Belief-Desire-Intention), in cui:

- i *belief* corrispondono all'informazione che l'agente ha sul mondo. Può essere incompleta o non corretta.

- i *desire* rappresentano la situazione che l'agente vorrebbe raggiungere.

- le *intention* rappresentano i desideri che l'agente si è impegnato a realizzare.

La tecnologia ad agenti risulta essere una risposta promettente alle richieste dello sviluppo del software moderno.

Molti sistemi software, infatti, integrano componenti autonome. L'autonomia talvolta riduce la capacità di una componente di reagire ad eventi e di gestirli, tuttavia in molti casi fa sì che una componente possa agire in modo proattivo. Questo è il caso dei moderni sistemi di controllo per domini

fisici, in cui il controllo è proattivo, che sono implementati con un insieme di processi autonomi e cooperativi o con sistemi basati su calcolatori integrati interagenti con altri mediante sensori distribuiti in rete. L'integrazione di dispositivi mobili entro applicazioni e sistemi complessi e distribuiti può essere realizzata solo modellando questi dispositivi in temini di componenti software autonome. Le applicazioni Internet sono tipicamente formate da processi autonomi, eseguiti anche su nodi diversi, e cooperanti con altri processi seguendo un approccio dettato più dalla semplicità concettuale e dalla gestione decentrata che dalla reale necessità di avere attività concorrenti e autonome.

I sistemi computazionali sono situati. C'è una consapevolezza precisa su dove le componenti sono collocate ed eseguite e su quali componenti interagiscono esplicitamente. Applicazioni mobili ed applicazioni di "computazione pervasiva" (pervasive computing) riconoscono la necessità dell'applicazione di modellare esplicitamente le caratteristiche e i dati dell'ambiente piuttosto che modellarle implicitamente in termini di attributi interni di un oggetto.

La socialità nei moderni sistemi distribuiti rientra in diversi ambiti quali: la capacità delle componenti di supportare interazioni dinamiche, come l'interazione stabilita in esecuzione con una componente precedentemente sconosciuta; il livello di interazione più alto che supera lo schema client-server tradizionale; il rafforzamento di certe classi di regole che governano le interazioni. Tipicamente i sistemi di controllo per domini fisici critici non possono essere fermati e di solito non possono neanche essere rimossi dall'ambiente in cui sono integrati. Ciò nononstante, questi sistemi vanno aggiornati continuamente e l'ambiente in cui vivono cambia frequentemente con l'aggiunta di nuove componenti hardware e software. Per questi sistemi la gestione dell'apertura e la capacità di riorganizzare l'interazione automaticamente sono punti cruciali come lo è la capacità di una componente di entrare in un nuovo contesto di esecuzione rispettando le regole che guidano il funzionamento dell'intero sistema.
In riferimento ai sistemi di persavise computing, la mancanza di risorse, energia, o semplicemente l'irraggiungibilità nella comunicazione possono far si che i nodi vadano e vengano in modo impredicibile richiedendo la ristrutturazione degli schemi di comunicazione. Queste problematiche sono ulteriormente aggravate nell'ambito di reti mobili e sistemi P2P, in cui l'interazione deve essere fruibile e controllabile a prescindere dalla mancanza di strutture essenziali e di dinamiche di connettività. Considerazioni simili si possono fare per computazioni distribuite aperte e basate su Internet.

Si vede quindi come attualmente una sempre più vasta maggioranza di scenari per i moderni sistemi distribuiti sia intrinsecamente incline ad essere sviluppata in termini di sistemi multi agente (Multi-Agent Systems, MAS) e come questi moderni sistemi distribuiti siano di fatto già dei MAS.

La ricerca sui MAS è incentrata proprio sullo studio del comportamento

di una collezione di agenti autonomi, eterogenei, eventualmente preesistenti, che mirano a risolvere un dato problema e/o che competono per massimizzare il vantaggio individuale. Tale comportamento è determinato dalla capacità degli agenti di interagire e comunicare.

La progettazione e realizzazione di un MAS può essere fatta mediante diversi sistemi o linguaggi di specifica. I vari linguaggi proposti in letteratura hanno solitamente caratteristiche molto diverse: anche linguaggi fondati sullo stesso modello dichiarativo, ad esempio sul modello BDI, non rappresentano, o non implementano, concetti e meccanismi in modo uniforme.

Un'esempio di questa difformità tra linguaggi è dato dal meccanismo di gestione del caso in cui a fronte di un evento verificatosi all'interno del MAS, l'agente non trovi la conoscenza procedurale necessaria per gestirlo. Alcuni linguaggi ignorano il problema scartando semplicemente l'evento, altri risolvono il problema ad hoc disponendo di procedure alternative da eseguire per default, altri ancora rinviano la gestione dell'evento ad un secondo momento in cui la conoscenza procedurale per esso potrebbe essersi resa disponibile.

Questa eterogeneità è in parte dovuta alla scarsa attenzione riservata alla nozione di *agente cooperativo* che, invence, meriterebbe un ruolo di maggior rilievo nell'ambito dell'architettura BDI. La *cooperazione* è la capacità di un agente di aiutare gli altri agenti a realizzare i propri desideri.

La tesi si colloca nell'ambito della ricerca sui linguaggi BDI e si propone di implementare un interprete per un linguaggio esistente. Tale linguaggio, Coo-AgentSpeak [4], estende le funzionalità di AgentSpeak(L), uno dei primi linguaggi di tipo BDI proposti in letteratura, con la capacità di aggiornare dinamicamente la libreria dei piani in seguito a scambio cooperativo di piani.

Questa capacità è introdotta integrando in AgentSpeak(L) i concetti caratterizzanti del modello *Cooperative BDI (Coo-BDI)* [3]. La proposta Coo-BDI estende il modello BDI fornendo per esso un meccanismo generale ed efficiente per la promozione della cooperazione tra agenti, consentendo agli agenti di scambiare i propri piani.

Le potenzialità applicative del modello Coo-BDI si possono cercare in quei domini in cui gli agenti devono *apprendere* o hanno risorse limitate quali:

*Elaboratori portatili personali (Personal Digital Assistants, PDAs).* Nei PDA le risorse fisiche sono solitamente limitate e affinché la tecnologia del PDA sia utilizzabile in modo efficace è necessario caricare e collegare il codice in modo dinamico. L'approccio seguito in Coo-AgentSpeak consente di effettuare queste azioni di ricerca di piani all'esterno e di estendere dinamicamente la libreria dei piani, attività non previste nell'approccio BDI tradizionale, e permette all'agente di

scartare i piani esterni dopo il loro utilizzo: questo può essere utile in quanto la risorsa spazio nei PDA è limitata.

*Agenti auto-riparanti (Self-repairing agents).* Un agente auto-riparante è un agente posto in un ambiente software che cambia dinamicamente e capace di identificare porzioni del proprio codice che possano essere aggiornate per assicurare il corretto funzionamento nell'ambiente in evoluzione. Quando l'agente trova pezzi di codice obsoleto lo rimpiazza con codice nuovo fornito da un agente server senza bisogno di fermare la propria attività o l'intero sistema. Coo-AgentSpeak consente di rimpiazzare dinamicamente piani locali con altri piani esterni e questo meccanismo può essere usato per modellare un agente auto-riparante.

*Maggiordomi digitali (Digital butlers).* Un maggiordomo digitale è un agente che assiste l'utente in diversi compiti quali la gestione dell'agenda, il filtraggio della posta in arrivo, il recupero di informazioni interessanti dal web. Una caratteristica tipica di un maggiordomo digitale è la capacità di adattare dinamicamente il suo comportamento alle necessità dell'utente. Questa abilità si ottiene cooperando sia maggiordomi digitali con più esperienza, sia con l'utente assistito. L'utente può istruire l'agente, attraverso l'uso di un'interfaccia utente, mostrando la giusta sequenza di azioni da completare in determinate sistuazioni. L'agente può trattare questa sequenza come un piano recuperato esternamente che viene aggiunto alla sua libreria dei piani esattamente come per i piani recuperati da agenti suoi pari.

La novità di Coo-AgentSpeak rispetto alle proposte esistenti in letteratura risiede nella sua aderenza al linguaggio proposto originariamente da A. S. Rao [61], del quale eredita pertanto i vantaggi ed il potere espressivo, ed alla sua introduzione di costrutti nuovi, quali la gestione delle forze illocutorie e la possibilità di estendere la libreria dei piani tramite cooperazione con altri agenti.

Alcuni di tali costrutti si ritrovano in linguaggi esistenti, ma la loro presenza contemporanea in un unico linguaggio rappresenta un passo avanti rispetto allo stato dell'arte.

Un altro aspetto innovativo di Coo-AgentSpeak riguarda gli aspetti di ingegnerizzazione del software orientato agli agenti. Tipicamente i linguaggi orientati agli agenti si concentravano sulla programmazione di un agente individuale autonomo, lasciando all'utente la progettazione dello sviluppo del sistema multi-agente in cui gli agenti interagivano. Attualmente nella costruzione dei sistemi multi-agente ci si concentra maggiormente sugli aspetti sociali, quali gruppi e ruoli, piuttosto che sullo sviluppo di agenti individuali e di loro aspetti cognitivi. In quest'ottica Coo-AgentSpeak risulta essere uno strumento a supporto della programmazione di *team (squadre)* di

agenti in cui ogni agente è programmato in un linguaggio di programmazione orientato agli agenti.

L'interprete per Coo-AgentSpeak oggetto di questo lavoro di tesi è stato realizzato estendendo *Jason* (***J**ava-based **a**gentSpeak interpreter used with **S**ACI for multi-agent distribution **o**ver the **n**et*), l'interprete per il linguaggio AgentSpeak(L), con il meccanismo per la cooperazione di Coo-BDI.

La tesi si articola in sei capitoli.

**Capitolo 1.** Questo capitolo è costituito da una rassegna su alcuni dei più noti linguaggi di specifica per agenti.

Per ognuno dei linguaggi si sono descritti: la logica su cui è fondato (prevalentemente quella BDI); la sintassi; la semantica operazionale, talvolta fornita mediante descrizione informale, altre volte mediante l'uso di sistemi transazionali; eventuali implementazioni realizzate e loro applicazione; eventuali estensioni esistenti.

I linguaggi trattati sono dMARS, 3APL, JACK, AgentTalk, JAM e AgentSpeak(L), per quanto rigurda quelli basati su una logica prevalentemente BDI, seguiti da ConGolog, basato sul Situation Calculus, Dribble, basato sulla logica dinamica e Agent-0, basato sulla logica modale.

Tutti questi linguaggi, escluso Dribble, hanno un interprete o un compilatore eseguibile.

**Capitolo 2.** Contiene la descrizione di AgentSpeak(L) con particolare attenzione alla descrizione della semantica operazionale mediante un sistema transazionale. Presenta inoltre alcune estensioni del linguaggio utili per la realizzazione di Coo-AgentSpeak.

**Capitolo 3.** Comprende una descrizione delle componenti fondamentali di un linguaggio di comunicazione per agenti (ACL) e una breve rassegna dei principali ACL esistenti in letteratura.

**Capitolo 4.** Al suo interno si fa una breve analisi di come il problema dell'assenza di conoscenza procedurale è affrontato nei linguaggi di specifica trattati nel Capitolo 1 e si descrive l'approccio Coo-BDI come soluzione parziale ad esso.

**Capitolo 5.** Include l'analisi di alcune modifiche da apportare all'idea Coo-BDI per renderla integrabile in Jason ed una descrizione di Jason dal punto di vista sintattico e del suo funzionamento; accenna inoltre lo studio di fattibilità relativo ad una possibile implementazione descrivendo scostamenti e fornendo motivazioni.

**Capitolo 6.** Descrive la realizzazione dell'implementazione di Coo-AgentSpeak attraverso le fasi di test e studio del funzionamento di Jason; la progettazione del meccanismo di recupero dei piani e la sua implementazione in Java.

Infine, nel Capitolo **Conclusioni e sviluppi futuri** oltre alle considerazioni sul lavoro svolto sono date indicazioni su possibili modifiche ed estensioni apportabili a Coo-AgentSpeak.

# Capitolo 1

# Linguaggi di specifica per agenti

La nozione secondo cui un agente è un'entità soggetta a credenze, desideri, ecc. è nota ed accettata da molti ricercatori.

Al fine di formalizzare sistemi intenzionali sono state sviluppate diverse logiche tra cui la teoria delle intenzioni di P. R. Cohen e H. J. Levesque [14] e la logica BDI di A. S. Rao e Georgeff [62]. Tuttavia, gli agenti intelligenti software non possono essere formalizzati usando linguaggi logici ad hoc: devono essere programmati usando linguaggi eseguibili come qualsiasi altro software. Per questo c'è bisogno di programmi che riescano a colmare il "gap" (divario) tra la teoria logica e le esigenze pratiche legate allo sviluppo degli agenti software.

Uno dei modelli che ha ottenuto più consenso come candidato per colmare questo divario è quello BDI [63] che verrà descritto in seguito.

I linguaggi trattati in questa rassegna sono principalmente basati sul modello BDI e sono in parte collegati al *Procedural Reasoning System (PRS)* [55].

PRS è stato sviluppato per la rappresentazione e l'uso di conoscenza procedurale proveniente da esperti al fine di soddisfare goal. Può essere considerato l'antenato di tutti i linguaggi e le architetture per il ragionamento pratico presentati in questo capitolo. La conoscenza procedurale equivale alla descrizione di collezioni di azioni strutturate da usare in situazioni specifiche. PRS supporta la definizione di sistemi real-time, attivi e intelligenti che fanno uso di conoscenza procedurale, quali programmi di diagnostica e controllori di sistema.

Il primo gruppo di linguaggi considerato nel capitolo è costituito da quei linguaggi che estendono PRS condividendone le caratteristiche principali sia per quanto riguarda i concetti fondamentali del linguaggio (belief, desideri/goal e intenzioni), sia per quanto riguarda il modo di operare dell'agente. I linguaggi in questo gruppo sono dMars che appartiene alla

seconda generazione di linguaggi per agenti basati su PRS, JACK che è di terza generazione e AgentTalk che viene considerato come un discendente di dMars anche se presenta alcune differenze per quanto riguarda i concetti di base. Nella seconda generazione rientra anche JAM che contempla i concetti base di PRS e supporta ragionamento al metalivello oltre a meccanismi di serializzazione utili per gestire la mobilità.

Il secondo gruppo di agenti considerato comprende 3APL e Dribble che differiscono da PRS in modo sostanziale pur condividendone alcuni concetti. Questi linguaggi prevedono la presenza di belief e goal, comuni a PRS, regole e azioni. Gli agenti operano seguendo una fase di ragionamento e una di esecuzione delle azioni definite mediante un sistema di transizione. Dribble usa la logica dinamica per provare le proprietà degli agenti.

Nella rassegna sono stati trattati altri due linguaggi che meritano un discorso a parte.

ConGolog è un linguaggio di programmazione concorrente che permette l'esecuzione di programmi ad alto livello. Si basa sul situation calculus. In esso non si parla di agenti, che sono presenti nella sua estensione CASL, ma di programmi la cui esecuzione è guidata da un sistema di transizione. ConGolog è stato incluso nella rassegna in quanto consente comunque di programmare agenti che hanno una conoscenza procedurale (simile ai piani) definita in modo dichiarativo.

Infine, Agent-0 è il primo linguaggio di programmazione agent-oriented ed ha quindi una certa valenza storica. La logica su cui si fonda è quella modale. I suoi concetti di base si discostano da quelli dei linguaggi precedenti, pur includendo i belief e un insieme di regole, mentre l'interprete opera seguendo due fasi: una di reazione e una di esecuzione. Le sue estensioni sono interessanti: la prima introduce l'uso di piani e la capacità di modificarli dinamicamente, la seconda integra il formato KQML con la sintassi per i messaggi.

I sistemi ed i linguaggi presentati sono stati scelti in quanto sono tutti linguaggi per i quali esiste un interprete o un compilatore che li rende eseguibili. Dribble è un'eccezione poiché al momento non ne esistono implementazioni ma è comunque interessante in quanto permette di creare e modificare piani dinamicamente, condividendo pertanto uno degli obiettivi principali di Coo-AgentSpeak.

Ogni linguaggio compreso in questa descrizione verrà esaminato tenendo conto di vari aspetti:

- la logica predominante su cui tali sistemi e linguaggi si basano;
- la descrizione sintattica del linguaggio e dei suoi concetti fondamentali;
- la semantica;
- le implementazioni;

- le estensioni.

I linguaggi dei primi due gruppi, legati a PRS e ai concetti BDI, sono presentati seguendo un ordine cronologico. Seguono le descrizioni di ConGolog ed Agent-0.

## 1.1 dMARS

dMars (distributed Multi-Agent Reasoning System) sviluppato da M. d'Inverno *et al.* nel 1998 [22] è una piattaforma per l'implementazione di agenti intelligenti che si basa sull'architettura PSR (Procedural Reasoning System) [56] sviluppata da M. P. Georgeff e A. L. Lansky [28]. PRS è un'architettura i cui concetti trovano fondamento nel Belief-Desire-Intention (BDI) model del practical reasoning sviluppato da M. Bratman *et al.* [10].

### 1.1.1 BDI model

Il modello BDI è caratterizzato dalla presenza di agenti razionali aventi risorse limitate e limitate capacità di capire e di conoscere ciò che avviene nell'ambiente in cui vivono. Per ogni agente queste risorse e capacità sono rappresentate dai seguenti concetti:

- *Belief* [1] *(o credenze)*: la conoscenza dell'agente riguardo il mondo.

- *Desire (o desideri)*: obiettivi da perseguire; sono simili ai goal [2] ma non è necessario che siano consistenti.

- *Intention (o intenzioni)* : piani per cui ci si è impegnati a procedere con l'esecuzione.

- *Plan (o piani)*: "ricette" che rappresentano la conoscenza procedurale dell'agente. I piani sono generalmente caratterizzati da un trigger che determina l'adozione del piano, una precondizione che deve essere soddisfatta dallo stato corrente affinché il piano sia applicabile, un corpo che consiste di azioni da portare a termine, una condizione invariante che deve essere soddisfatta durante tutta l'esecuzione del piano, un insieme di azioni da eseguire se il piano termina con successo e un insieme di azioni da eseguire se il piano fallisce. I piani sono statici: non possono cambiare né essere cambiati durante il ciclo di vita dell'agente. Tutti i sitemi BDI comprendono una *event queue* in cui sono memorizzati gli eventi esterni (percepiti dall'ambiente) e i subgoal interni (generati dall'agente stesso mentre tenta di soddisfare un goal principale). Belief, desire e intention sono le attitudini mentali (*mental attitudes o mental states*) dell'agente.

[1] al contrario dei termini desire e intention non viene tradotto con il termine corrispondente, credenza, per tenere la corrispondenza con l'acronimo BDI

[2] il termine non viene tradotto perché l'uso della parola italiana corrispondente, obiettivo, non sempre rispecchia a pieno il significato attribuibile a goal in certi contesti

Il tipico ciclo di esecuzione per un agente BDI è caratterizzato dai passi seguenti:

1. osservare il mondo e lo stato interno dell'agente, e aggiornare di conseguenza la *event queue*;
2. generare le nuove istanze di piano per cui il trigger event può essere associato ad un evento nella event queue (*relevant* plan instances) e per cui la precondizione è soddisfatta (*applicable* plan instances);
3. selezionare per l'esecuzione un'istanza dall'insieme delle istanze di piano applicabili;
4. mettere l'istanza scelta su di uno *stack delle intenzioni (intention stack)* nuovo o esistente, a seconda che l'evento sia un goal o un subgoal;
5. selezionare uno stack per l'intenzione, prendere l'istanza di piano al top dello stack e eseguire il prossimo passo di tale piano: se il passo è un'azione, la si porta a compimento, altrimenti, se è un subgoal, lo si inserisce nell'event queue.

Generalmente, al momento della sua formulazione, un piano è specificato solo parzialmente poiché i passi precisi da eseguire dipendono dallo stato dell'ambiente nel momento in cui questi passi sono eventualmente eseguiti. Un agente razionale agisce mandando in esecuzione le azioni che intende eseguire senza ragionamenti ulteriori a meno che non sia forzato a revisionare le sue intenzioni da cambiamenti nei suoi belief o desideri.

### PRS: il precursore di dMARS

Il Procedural Reasoning System (PRS) della SRI International è un framework per la costruzione di sistemi che ragionano in tempo reale che portano a termine compiti complessi in ambienti dinamici. PRS dipende dalla rappresentazione della conoscenza procedurale, che descrive come eseguire un insieme di azioni al fine portare a termine uno scopo. PRS fornisce un ambiente in cui questo tipo di conoscenza procedurale su azioni e goal è espressa e processata prontamente.

Tale framework è tale che in esso possono essere facilmente integrati comportamenti goal-directed e event-driven ed è progettato per operare in ambienti dinamici.

PRS prevede funzionalità che lo rendono un potente sitema per lo sviluppo di applicazioni real-time. Copie multiple di oggetti PRS, ognuno dei quali è riferito come un *agente*, possono essere eseguite in parallelo. Gli agenti operano in modo asincrono ma possono comunicare attraverso il passaggio di messaggi per risolvere problemi in modo distribuito e cooperativo.

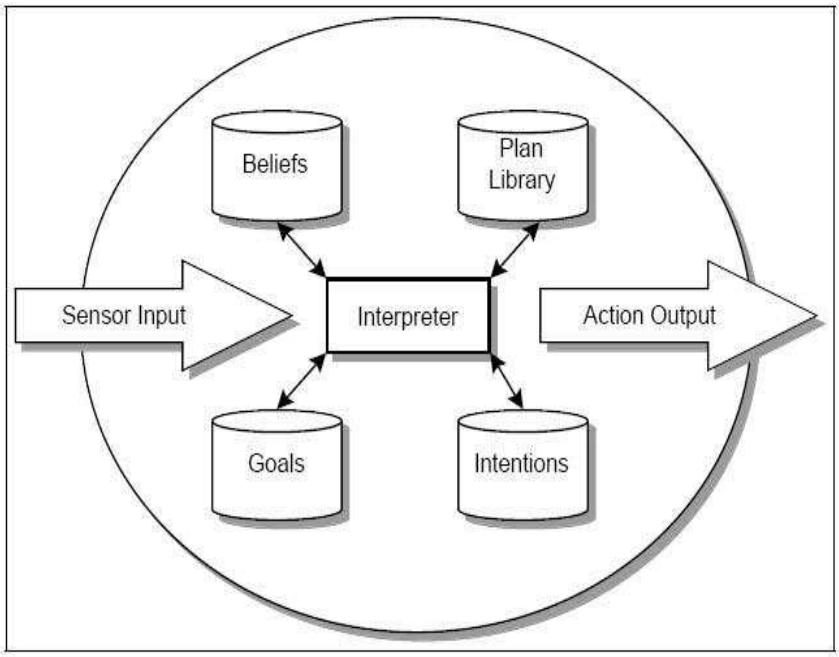



Figura 1.1: Architettura PRS

Altre funzionalità supportate da PRS sono ragionamento e pianificazione al metalivello. Queste funzionalità possono essere usate per implementare comportamenti di controllo e schedulazione complessi, come per le applicazioni individuali.

L'architettura PRS (Figura 1.1) è formata da un *Database* contenente i *Belief* correnti o i fatti relativi al mondo; un insieme dei *Goal* correnti da realizzare; un insieme di piani, detti *Acts* o *KAs* a seconda delle versioni dell'architettura, che descrivono come le sequenze di azioni e test possono essere portati a termine per perseguire certi goal o per reagire a una particolare situazione; e *Intention* contententi quei piani che sono stati scelti per un'eventuale esecuzione. Un *Interpreter* manipola le componenti, selezionando i piani appropriati per l'esecuzione basata su belief e goal del sistema, creando le intenzioni corrispondenti ed eseguendole.

L'interprete esegue l'intero sistema. In un particolare istante, si stabiliscono certi goal e occorrono certi eventi che alterano i belief nella base di dati del sistema. Questi cambiamenti nei goal e belief del sistema attivano varie Acts. Una o più di queste Acts applicabili saranno poi scelte e poste nel grafo delle intenzioni. Infine, PRS sceglie un compito (intenzione) dalla radice del grafo delle intenzioni ed esegue un passo di tale compito. Questo avrà come risultato il completamento di un'azione primitiva nel mondo, lo stabilire un nuovo sottogoal o la conclusione di alcuni nuovi belief, oppure

una modifica del grafo delle intenzioni stesso.

### 1.1.2 Il sistema dMARS

In dMARS i *belief* dell'agente corrispondono alla conoscenza che l'agente ha relativamente al mondo, che può essere incompleta o non corretta. I *desideri* (o goal) dell'agente corrispondono intuitivamente ai compiti ad esso assegnati e devono essere logicamente consistenti. Un'intuizione suggerita dai sistemi BDI è che un agente non possa esaudire tutti i propri desideri anche se sono consistenti. Si fissa quindi un sottoinsieme di desideri per la cui soddisfazione vengono impegnate delle risorse. Questi desideri scelti sono *intention* e l'agente tenta di esaudire una di queste intention finché crede che essa possa essere soddisfatta o finché sa di non poterla più soddisfare. Ogni agente ha una *plan library*, che è un insieme di piani, che specificano come un agente deve agire per seguire le proprie intention, ovvero la sua conoscenza procedurale.

Ogni piano contiene un *trigger*, o *invocation condition*, che specifica, solitamente in termini di eventi, le circostanze in cui il piano può essere considerato; un *context* o *pre-condition* che specifica le circostanze in cui l'esecuzione del piano può cominciare; una *maintenance condition* che caratterizza le circostanze che devono restare vere mentre il piano è in esecuzione; un *corpo* che definisce come procedere e che consiste di goal (o subgoal) e azioni primitive.

Tutto l'operato dell'agente dMARS è gestito dall'*interprete* che esegue continuamente questo ciclo:

- osserva il mondo e lo stato interno dell'agente e aggiorna la event queue in modo che rifletta gli eventi che sono stati osservati;
- genera nuovi possibili desideri, cercando i piani il cui trigger event è compatibile con un evento nella event queue;
- seleziona un piano per l'esecuzione da questo insieme di piani compatibili (un *intended means*);
- mette l'istanza del piano da eseguire su di un intention stack già esistente o nuovo a seconda che l'evento sia un subgoal o meno;
- seleziona un intention stack, prende il piano al top e esegue il prossimo passo di questo piano corrente: se il passo è un'azione la porta a termine, altrimenti, se è un subgoal mette tale subgoal nella event queue.

Questo è il modello base dell'esecuzione di un agente dMARS. La pianificazione fatta dagli agenti si basa sull'espansione dei subgoal context-sensitive che è differita fino al momento in cui il subgoal è scelto per l'esecuzione.

Si prosegue con una descrizione dei tipi base e delle componenti primitive del sistema in dMARS presentata usando il linguaggio Z [70]. Z è un linguaggio formale di specifica orientato ai modelli basato sulla teoria degli inisiemi e sulla logica del prim'ordine. Gli elementi chiave di una specifica in Z sono le definizioni dello *spazio degli stati* di un sistema e delle possibili *operazioni* che portano da uno stato all'altro. Un'estratto della notazione Z si trova in Appendice A.

I *belief* sono simili ai fatti PROLOG: sono essenzialmente letterali ground della logica del prim'ordine classica (formule atomiche positive o negative prive di variabili).

$$Belief == \{b : BeliefFormula \mid belvars\ b = \varnothing \bullet b\}$$
$$BeliefFormula ::= pos\langle\langle Atom\rangle\rangle \mid not\langle\langle Atom\rangle\rangle$$

I *goal* dell'agente sono specificati mediante un semplice linguaggio modale temporale con due connettivi unari in aggiunta a quelli della logica classica. Gli operatori sono "!" e "?" rispettivamente per "achieve" e "query". Quando un agente prevede il goal $!\varphi$ allora ha una sequenza di azioni che una volta portate a termine rendono $\varphi$ vera. Quando un agente prevede il goal $?\varphi$ allora ha una sequenza di azioni che una volta portate a termine portano l'agente a conoscenza della veridicità o meno di $\varphi$. Questi connettivi sono definiti in termini di *situation formulae* che sono espressioni che vanno valutate rispetto a un insieme di belief.

$$\begin{aligned} SituationFormula ::= {} & belform\langle\langle BeliefFormula\rangle\rangle \\ \mid {} & and\langle\langle SituationFormula \times SituationFormula\rangle\rangle \\ \mid {} & or\langle\langle SituationFormula \times SituationFormula\rangle\rangle \\ \mid {} & true \\ \mid {} & false \end{aligned}$$

Un goal è definito mediante una *temporal formula* che è una belief formula avente come prefisso un operatore achieve o una situation formula avente come prefisso un operatore query.

$$Goal ::= achieve\langle\langle BeliefFormula\rangle\rangle \mid query\langle\langle SituationFormula\rangle\rangle$$

Le *azioni* che possono essere eseguite dagli agenti sono classificabili in *esterne* (nel qual caso il dominio dell'azione è l'ambiente all'esterno dell'agente) o *interne* (nel qual caso il dominio dell'azione è l'agente stesso). Un'azione esterna è specificata come se fosse una chiamata di procedura o un'invocazione di metodo ed è contraddistinta da un simbolo di azione, preso entro un certo insieme, applicato ad una sequenza di termini. Le azioni interne aggiungono o rimuovono un belief dal database purché il belief non contenga variabili.

*ExtAction*

$name : ActionSym$

$terms : \text{seq}\ Term$

$$IntAction ::= add\langle\langle BeliefFormula\rangle\rangle \mid remove\langle\langle BeliefFormula\rangle\rangle$$

I *piani* sono adottati dagli agenti e una volta scelti determinano il comportamento dell'agente e fungono come *intention*. I piani hanno sei componenti:

- una *invocation condition* (o *trigger event*) che è la causa dell'adozione del piano. Ci sono quattro tipi di eventi ammessi come triggers: l'acquisizione di un nuovo belief; la rimozione di un belief; la ricezione di un messaggio; l'acquisizione di un nuovo goal.

$$\begin{aligned} TriggerEvent ::= {} & addbelevent\langle\langle Belief\rangle\rangle \\ \mid {} & rembelevent\langle\langle Belief\rangle\rangle \\ \mid {} & toldevent\langle\langle Atom\rangle\rangle \\ \mid {} & goalevent\langle\langle Goal\rangle\rangle \end{aligned}$$

- un *context* opzionale (situation formula) che definisce la precondizione del piano che va creduta vera affinché il piano sia eseguibile.
- il *plan body* che è un albero che rappresenta il flusso delle azioni da compiere. In ogni albero gli archi (rami) sono etichettati con goal (o subgoal) o azioni (sia interne sia esterne) e i nodi sono gli stati supportati. Un plan body può essere sia un *end tip* contenente uno stato si una *fork* contentente uno stato e un insieme non vuoto di rami ognuno dei quali porta ad un altro albero. Eseguire un piano con successo vuol dire attraversare un albero dalla radice ad un nodo foglia.

$$Body ::= End\langle\langle State\rangle\rangle \mid Fork\langle\langle \mathbb{P}_1(State \times Branch \times Body)\rangle\rangle$$

- una *maintenance condition* che deve essere vera affinché il piano continui ad essere eseguito.
- un insieme di *azioni interne* che vengono eseguite se il piano termina con successo.
- un insieme di *azioni interne* che vengono eseguite se il piano fallisce.

*Plan*

$$
\begin{aligned}
&inv : TriggerEvent \\
&context : optional[SituationFormula] \\
&body : Body \\
&maint : SituationFormula \\
&succ : \mathrm{seq}\, IntAction \\
&fail : \mathrm{seq}\, IntAction
\end{aligned}
$$

I piani senza body sono detti *primitive plans*.

Nell'esecuzione di un agente dMARS quando si seleziona un piano per un evento si genera per esso una istanza di piano (*plan instance*).

$$Status ::= active \mid suspended$$

*PlanInstance*

$$
\begin{aligned}
&origplan : Plan \\
&env : Substitution \\
&state : State \\
&nextbranches : \mathbb{P}\, Branch \\
&branch : optional[Branch] \\
&status : Status \\
&id : PlanInstanceId \\
\hline
&state \in PlanStates\ origplan \\
&state \in \mathrm{dom}\, End \Rightarrow nextbranches = \varnothing \\
&nextbranches \subseteq PlanNextBranches\ origplan\ state \\
&branch \subseteq nextbranches
\end{aligned}
$$

Un'istanza di piano contiene una copia del piano originale e, in aggiunta: l'*ambiente* del piano; lo stato corrente raggiunto dal piano; l'insieme di rami che possono essere attraversati a partire da tale stato; il ramo che si sta attraversando; un identificatore per identificare univocamente l'istanza di piano; e in fine lo *status* del piano (*active* se il piano è parte di un'intenzione, *suspended* se il piano è stato temporaneamente sospeso). Quando un ramo non può essere attraversato esso fallisce e viene rimosso dall'insieme dei rami possibili. Se il ramo che l'agente sta provando ad attraversare è definito e l'agente ha scelto il prossimo ramo da provare che è indefinito, questa scelta non può essere fatta. Si individuano l'insieme dei prossimi rami che si possono raggiungere da un dato stato in un piano e tutti gli stati. Si devono poter determinare anche il prossimo stato in un piano a partire dallo stato corrente e dal ramo attraversato e lo stato iniziale di un piano. Quando un piano è selezionato per primo, il suo stato corrente si riferisce allo stato

iniziale. Un piano è di *successo* quando arriva in uno stato finale, è detto di *fallimento* se non è in uno stato finale e non ci sono altri rami disponibili (ad esempio, tutti i rami sono stati provati ma nessuno ha portato al successo).

$$
\begin{aligned}
&InitialInstance == \{p : PlanInstance \mid \\
&\qquad p.state = PlanStartState\ p.origplan \land p.status = active\} \\
&SucceedInstance == \{p : PlanInstance \mid p.state \in (\mathrm{dom}\ End)\} \\
&FailedInstance == \{p : PlanInstance \mid \\
&\qquad p.state \notin (\mathrm{dom}\ End) \land p.nextbranches = \{\}\}
\end{aligned}
$$

Un'intenzione (*intention*) in dMars è una sequenza di istanze di piano. Se l'intenzione è creata in risposta ad un evento esterno essa contiene l'istanza di piano appena generata. Se tale piano, a sua volta, genera un evento esterno a cui l'agente risponde con un altro piano, il nuovo piano è concatenato all'intenzione. In questo modo il piano al top dello stack dell'intenzione è il piano che sarà eseguito per primo in ogni intenzione.

$$Intention == \mathrm{seq}\ PlanInstance$$

Un evento (*event*) è formato da un triggering event e, opzionalmente, da un identificatore di istanza di piano che identifica il piano che ha generato l'evento, un ambiente e un insieme di istanze di piano che potrebbero già essere fallite (e non potrebbero essere riprocessate). Un evento è esterno quando non è associato a un piano esistente per esso quando entra nella event queue. Un evento interno è un evento subgoal che si ha quando il ramo di un'intenzione in esecuzione è un achieve goal che non può essere soddisfatto immediatamente.

$$
\begin{array}{|l}
\hline
Event \\
trig : TriggerEvent \\
id : optional[PlanInstanceId] \\
env : optional[Substitution] \\
failures : optional[\mathbb{P}\ PlanInstance] \\
\hline
\end{array}
$$

### 1.1.3 Una semantica operazionale per gli agenti dMARS

In questa sezione si specifica l'operato dell'agente dMARS considerando l'agente e il suo stato, la generazione di piani rilevanti e applicabili, il modo in cui gli eventi sono processati, l'esecuzione delle intenzioni e, in fine, il completamento e il fallimento dei piani.

L'*agente* dMARS è formato da una plan library, dalle funzioni di selezione per intenzioni, eventi e piani, da un'altra funzione di selezione per le sostituzioni che permette di scegliere tra le possibili associazioni alternative,

e da una funzione per selezionare il prossimo ramo che va eseguito entro un piano.

*dMARSAgent*

$$
\begin{aligned}
&planlibrary : \mathbb{P}\ Plan \\
&intentionselect : \mathbb{P}_1\ Intention \rightarrow Intention \\
&planselect : \mathbb{P}_1\ Plan \rightarrow Plan \\
&eventselect : \mathrm{seq}_1\ Event \rightarrow Event \\
&substitutionselect : \mathbb{P}_1\ Substitution \rightarrow Substitution \\
&selectbranch : PlanInstance \nrightarrow Branch
\end{aligned}
$$

Lo *stato* dell'agente indica gli aspetti che cambiano nel tempo. I suoi componenti sono i belief dell'agente, le intenzioni, gli eventi ancora da processare. Un'operazione influenza lo stato dell'agente dMARS e non l'agente stesso. Inizialmente l'agente è provvisto di una event queue e degli insiemi di belief e intenzioni da cui saranno generate l'intenzione e l'azione successiva. Gli agenti percepiscono gli eventi esterni che vengono messi in coda.

*dMARSAgentState*

$$
\begin{aligned}
&dMARSAgent \\
&beliefs : \mathbb{P}\ Belief \\
&intentions : \mathbb{P}\ Intention \\
&events : \mathrm{seq}\ Event
\end{aligned}
$$

Un piano è *relevant* rispetto a un evento se esiste un *most general unifier* (mgu) che unifica il triggering event del piano e l'evento. Questo si fa mediante una funzione *genrelplans* che prende un evento $e$ e un insieme di piani $ps$ e ritorna un insieme di coppie (piano, sostituzione) tali che se si restituisce $(p,\sigma)$, allora $p$ è un piano rilevante in $ps$ per l'evento $e$ e $\sigma$ è l'mgu tra $e$ ed il triggering event di $p$. Se l'evento è un subgoal event e contiene un ambiente con sostituzione, questa va applicata al triggering event prima di generare i piani rilevanti.

$$
\begin{aligned}
&genrelplans : Event \rightarrow \mathbb{P}\ Plan \rightarrow \mathbb{P}(Plan \times Substitution) \\
&\forall\ e : Event;\ lib : \mathbb{P}\ Plan \bullet \\
&\quad undefined\ e.env \Rightarrow genrelplans\ e\ lib = \\
&\qquad \{p : lib;\ \sigma : Substitution \mid mguevents\ (e.trig, p.inv) = \sigma \bullet (p,\sigma)\} \wedge \\
&\quad defined\ e.env \Rightarrow genrelplans\ e\ lib = \\
&\qquad \{p : lib;\ \sigma : Substitution \mid \\
&\qquad\quad mguevents\ (ASTrigEvent\ (the\ e.env)\ e.trig, p.inv) = \sigma \bullet (p,\sigma)\}
\end{aligned}
$$

Un piano rilevante è *applicable* se il suo context è una conseguenza logica dei belief dell'agente. Il piano applicabile è generato mediante una funzione *genapplplans* che prende un insieme di piani e la sostituzione che li rende rilevanti, i belief correnti, e che, oltre a restituire il piano applicabile, aggiorna le sostituzioni.

$$
\begin{array}{l}
genapplplans : \mathbb{P}(Plan \times Substitution) \rightarrow \\
\qquad (\mathbb{P}\, BeliefFormula) \rightarrow \\
\qquad\qquad \mathbb{P}(Plan \times Substitution) \\
\hline
\forall\, relsubs : \mathbb{P}(Plan \times Substitution); \\
\qquad bels : \mathbb{P}\, BeliefFormula \bullet \\
genapplplans\ relsubs\ bels = \\
\quad \{rel : Plan;\ \sigma, \psi : Substitution \mid \\
\quad (rel, \sigma) \in relsubs \land \\
\quad dMarsLogCons(ASSitForm\ (\sigma \ddagger \psi)\ (the\ rel.context), bels) \bullet \\
\qquad\qquad\qquad\qquad (rel, \sigma \ddagger \psi)\}
\end{array}
$$

Gli eventi sono processati in due modi a seconda che la event queue sia vuota o meno.

Se la coda è *non vuota* si sceglie un evento, si determinano piani rilevanti e applicabili per esso e si sceglie un piano per cui generare un'istanza. Con un evento esterno si crea una nuova intenzione contenente la sola istanza di piano. Con un evento interno, l'istanza di piano è messa sopra allo stack dell'intenzione che ha generato tale evento (subgoal). Un'istanza di piano che fallisce non può essere scelta per un evento interno. L'istanza di piano nel suo stato iniziale è creata da una funzione che prende come argomenti un piano e una sostituzione. Se l'evento è esterno esso deve essere aggiornato in modo che includa l'identificatore della nuova istanza di piano.

Se la coda è *vuota* si processano le intenzioni. A questo punto non risultano intenzioni, piani e rami selezionati. Il primo passo da fare è quello di scegliere un'intenzione, identificare il piano per l'esecuzione, che è quello che si trova al top dello stack dell'intenzione in modo che sia active, e si sceglie il ramo entro il corpo del piano da eseguire.

$$
\begin{array}{l}
\textit{SelectIntention} \\
\hline
\Delta AgentIntExecutionOperationState \\
\hline
events = \langle\rangle \\
the\ selectedintention' = intentionselect\ intentions \\
the\ executingplan' = head(the\ selectedintention') \\
(the\ executingplan').status = active \\
(the\ executingbranch') = selectbranch\ (the\ executingplan') \\
\hline
\end{array}
$$

A seconda che il ramo scelto porti ad eseguire un'azione interna o esterna, un query goal o un achieve goal si devono sempre considerare i due casi per

cui o si raggiunge uno stato successivo nell'esecuzione del body, se il ramo porta ad un'esecuzione di successo, o si elimina il ramo se la sua esecuzione porta a un fallimento.

*BranchSucceed*

$$\Delta AgentIntExecutionOperationState$$

$$the\ executingplan' = AchieveBranch\ (the\ executingplan)$$

*BranchFail*

$$\Delta AgentIntExecutionOperationState$$

$$(the\ executingplan').nextbranches = (the\ executingplan).nextbranches \setminus executingbranch$$

Se si ha un'*azione esterna* la si esegue immediatamente. L'esecuzione è modellata da una funzione che prende un'istanza di piano entro cui si è scelta l'azione esterna in esecuzione e rende la sostituzione in corrispondenza della quale si ha successo che viene *composta* con le altre sostituzioni già applicate all'ambiente. Se l'azione non è compresa nel dominio di questa funzione si ha fallimento. Se si ha un fallimento non c'è cambio di stato e il ramo deve essere rimosso dai possibili rami per il piano in esecuzione.

Se si ha un'*azione interna* si modifica il data base in accordo con essa. Se questa azione porta ad un cambiamento nel data base si aggiunge un evento alla event queue e si ci muove nello stato successivo dell'esecuzione del piano.

Nel caso di *query goal*, se l'ambiente applicato al goal può essere unificato con l'insieme dei belief mediante i most general unifiers (mgu) si sceglie uno di questi. Tale sostituzione è composta con le sostituzioni dell'ambiente e si raggiunge lo stato successivo. Se non ci sono unificazioni possibili il ramo fallisce.

Con un *achieve goal*, il ramo ha successo se il goal può essere unificato con i belief e quindi il resto del piano in esecuzione è unificato come nel caso precedente. Se il goal non può essere unificato, l'evento achieve goal è posticipato, il piano in esecuzione è sospeso modificandone il parametro status e l'insieme delle istanze provate è ridefinito in modo da contenere l'insieme vuoto. L'identificatore del nuovo evento interno è settato per indicare il piano correntemente in esecuzione. Una volta che un achieve goal è rinviato, il ciclo di esecuzione può ripartire, in caso contrario ulteriori operazioni sono portate a termine come segue.

Un ramo di successo porta ad un nuovo stato che può essere uno stato non finale, nel qual caso si procede con l'esecuzione di un altro ramo, o uno stato finale, nel qual caso il piano ha *successo*. Nell'ultimo caso, le sostituzioni dell'ambiente sono applicate alle condizioni che hanno portato

al successo per dare una sequenza di azioni interne ground. Poi, il database è aggiornato per completare queste azioni ground una alla volta sull'insieme di belief corrente in modo da ottenere il nuovo insieme di belief.

Se un piano ha successo si considerano altri due casi. Se ci sono più piani nell'intenzione, l'ambiente con le sostituzioni corrente viene aggiornato per includere le sostituzioni appropriate per tutti i piani realizzati e l'ambiente del piano successivo nello stack. L'istanza di piano terminato con successo è poi rimossa dal top dell'intenzione scelta poiché il nuovo piano in esecuzione, che viene riattivato, è il secondo nello stack originale. Anche l'evento interno che ha generato il piano completato viene rimosso. Se non ci sono altri piani, l'intenzione ha successo e può essere rimossa così come l'evento esterno che l'ha generata.

Infine, se un piano fallisce ma restano dei rami, questi possono essere provati. Se non ci sono ulteriori alternative, il piano *fallisce*. Quando questo è il solo piano sullo stack, l'intenzione fallisce completamente, altrimenti le sostituzioni dell'ambiente sono applicate alle condizioni di fallimento del piano e le azioni ground interne di fallimento sono completate. Se non è il solo piano sullo stack, deve esistere nell'event queue un goal event per cui tale piano è stato adottato; questo evento viene trovato e aggiornato per tenere memoria dell'istanza di piano fallita in modo che non sia riprovata. Lo status del secondo piano sullo stack dell'intenzione resta sospeso.

### 1.1.4 Implementazioni

Un'implementazione di dMARS è stata realizzata dall'organizzazione AAII (Australian Artificial Intelligence Institute, Melbourne, Victoria - Australia) [11, 27]. Questa implementazione comprende editor grafici, un compilatore e un interprete per un linguaggio di programmazione logica goal-oriented, un certo numero di librerie run-time (che includono un database per la conoscenza in-memory, un multi-threading package e un sottosistema di comunicazione). È stata sviluppata in C++ e gira su varie piattaforme Unix e su Windows/NT. dMARS è usato sia in ambito di ricerca, sia nella produzione di sistemi per l'automazione industriale e per la simulazione di sistemi commerciali e di controllo del traffico aereo. Di seguito alcuni utilizzatori del sistema dMARS

- Space Shuttle Malfunction Handling (*NASA*)
- Space Station Command and Control (*USA*)
- Combat Simulation (*Air Operations Division of Defence Science and Technology Organization (DSTO-AOD), Surveillance Systems Division of Defence Science and Technology Organization (DSTO-SSD), Maritime Operations Division of Defence Science and Technology Organization (DSTO-MOD)*)

- Situation Awareness (*Land Operations Division of Defence Science and Technology Organization (DSTO-LOD)*)

- Communications Management (*Communication Division of Defence Science and Technology Organization (DSTO-Comms)*)

- Air Traffic Control (*AirServices, Thomson Airsys*)

- Network Management Prototype (*Telstra*)

- Supply Chain Management (*Daimler Chrysler*)

- Resource & Logistics Management (*Daimler Chrysler*)

- Resource Data Analysis (*Broken Hill Proprietary Company (BHP)*)

- Process Control (*Hazelwood Power*)

### 1.1.5 Estensioni

Il modello BDI descritto e istanziato come in dMARS, è l'ispirazione per altri modelli e sistemi quali AgentSpeak(L) e JACK.

## 1.2 3APL

Il linguaggio di programmazione per agenti 3APL è stato sviluppato da M. d'Inverno *et al.* nel 1998 [21]. Esso supporta la progettazione e la costruzione di agenti intelligenti per lo sviluppo di sistemi complessi attraverso un insieme di concetti intuitivi quali belief, goal e piani. 3APL usa un insieme di primitive per programmare agenti formato da insiemi di belief, goal e regole di ragionamento (reasoning rules). I belief rappresentano le questioni di cui l'agente deve occuparsi, i goal indicano cosa si vuole perseguire e come farlo, quindi sono usati per rappresentare sia achievement goal sia piani. Le regole di ragionamento pratico forniscono all'agente la capacità di trovare un piano adatto per soddisfare un goal, la capacità di creare nuovi goal per trattare una situazione particolare e la capacità di usare le regole per revisionare un piano.

L'architettura per 3APL è basata sul *think-act cycle* ed è divisa in due parti: la prima parte corrisponde a una fase di ragionamento pratico e usa le regole per esso, la seconda corrisponde a una fase di esecuzione in cui l'agente compie delle azioni.

In seguito si introducono, usando il linguaggio Z introdotto in Appendice A, i tipi base usati nel modello 3APL che comprendono belief, azioni, goal e regole di ragionamento; si definiscono gli agenti 3APL e si descrivono le loro operazioni.

### 1.2.1 Il sistema in 3APL

I *belief* possono essere: un atomo, una negazione di atomo, un congiunzione di due belief o l'implicazione di un belief da un altro belief. Un *atomo* è un simbolo di predicato, preso nell'insieme *PredSym* avente come argomento una sequenza di termini (anche vuota).

*Atom*

$head : PredSym$

$terms : \text{seq}\, FOTerm$

$$Belief ::= pos\langle\langle Atom\rangle\rangle \mid not\langle\langle Atom\rangle\rangle \mid and\langle\langle Belief \times Belief\rangle\rangle \mid imply\langle\langle Belief \times Belief\rangle\rangle \mid \mathbf{false} \mid \mathbf{true}$$

Le *azioni* sono rappresentate da simboli di azioni specificati in modo analogo agli atomi, prendendo i simboli di azione nell'insieme *ActionSym*. Ci sono delle funzioni ausiliarie definite per restituire l'insieme delle variabili in un atomo, un belief, o un'azione e che sono usate nello specificare le operazioni degli agenti.

*Action*

$name : ActionSym$

$terms : \text{seq}\, FOTerm$

I goal sono usati per rappresentare *sia i goal sia i piani* che servono per soddisfare i goal stessi dell'agente. I goal sono costruiti da costrutti base, come azioni, e da costrutti di programmazione imperativa regolari, come composizioni sequenziali e scelte non-deterministiche. I goal 3APL sono caratterizzati come *goal-to-do*, che sono attitudini mentali corrispondenti a piani d'azione per raggiungere un certo stato, o *goal-to-be*, che sono attitudini mentali corrispondenti allo stato desiderato da un agente.

Prima di descrivere formalmente i goal, si introducono i *contexts*, che sono goal aventi una caratteristica extra detta 'holes' che opera come *segnaposto* entro la struttura dei goal. Più precisamente, un *context* è sia un'azione di base (indicata con *bac*), un query goal, un achieve goal, la composizione sequenziale di due contexts, la scelta non-deterministica tra due contexts, una goal variable, sia un "□" che rappresenta un posto entro un contesto che può contenere un altro contesto. Nella definizione di seguito si usa l'insieme delle goal variable *GVar* per illustrare come si compie il processo detto di *goal revision* nel seguito.

$$Context ::= bac\langle\langle Action\rangle\rangle \mid query\langle\langle Belief\rangle\rangle \mid achieve\langle\langle Atom\rangle\rangle \mid seqcomp\langle\langle Context \times Context\rangle\rangle \mid choice\langle\langle Context \times Context\rangle\rangle \mid goalvar\langle\langle GVar\rangle\rangle \mid \square$$

Il □, che funge da segnaposto entro un contesto, è diverso da una goal variable, perché anche se entrambi rappresentano un segnaposto, □ è una funzionalità usata per *specificare* 3APL, mentre le goal variable sono parte di 3APL stesso.

Un *goal* può essere definito come un context senza alcuna occorrenza di □. La funzione ausiliaria *squarecount* è usata per contare le occorrenze di □ in un context.

$$
\begin{array}{|l}
squarecount : Context \to \mathbb{N} \\
\hline
\forall\, a : Action;\ b : Belief;\ at : Atom;\ c_1, c_2 : Context;\ gv : GVar \bullet \\
\quad squarecount(bac\ a) = 0 \land squarecount(query\ b) = 0 \land \\
\quad squarecount\ (achieve\ at) = 0 \land \\
\quad squarecount\ (seqcomp(c_1, c_2)) = squarecount\ c_1 + squarecount\ c_2 \land \\
\quad squarecount\ (choice(c_1, c_2)) = squarecount\ c_1 + squarecount\ c_2 \land \\
\quad squarecount\ (goalvar\ gv) = 0 \land squarecount\ \Box = 1
\end{array}
$$

$$Goal == \{g : Context \mid squarecount\ g = 0\}$$

Un *front context* è un importante tipo di context usato per illustrare le proprietà significative di 3APL. I front contexts sono contexts con *precisamente una sola* occorrenza di □ *all'inizio* (*at the front*) del context. Informalmente, un elemento all'inizio di un context indica che un agente deve scegliere di eseguire prima tale elemento, di modo che se un □ all'inizio di un context viene rimpiazzato da un goal, allora questo goal può essere portato a compimento per primo, prima del resto di tutti gli altri goal.

$$
\begin{array}{|l}
frontcontext\_ : \mathbb{P}(Context) \\
\hline
\forall\, fc, fc_1 : Context;\ g : Goal \bullet frontcontext\ (fc) \Leftrightarrow \\
\qquad fc = \Box \lor \\
\qquad (fc = seqcomp(fc_1, g) \land frontcontext\ fc_1) \lor \\
\qquad (fc = choice(fc_1, g) \land frontcontext\ fc_1) \lor \\
\qquad (fc = choice(g, fc_1) \land frontcontext\ fc_1)
\end{array}
$$

$$FrontContext == \{fc : Context \mid frontcontext\ fc\}$$

$$\forall\, fc : FrontContext \bullet squarecount\ fc = 1$$

Un front context può essere definito formalmente sia come un quadrato singolo, sia come la composizione sequenziale di un front context con un goal, o come la scelta di un front context e un goal.

Un goal può contenere sia goal variable, sia variabili del prim'ordine. Sono definite tre funzioni che restituiscono, rispettivamente, l'insieme di tutte le variabili, le goal variable e le variabili del prim'ordine di un goal.

Un agente 3APL usa *regole di ragionamento pratico* non solo per pianificare in senso convenzionale, ma anche per *riflettere* sui suoi goal. La riflessione consente all'agente di riconsiderare uno dei suoi piani in una situazione in cui il piano fallisce rispetto al goal che sta tentando di soddisfare, o è già fallito, o quando può essere seguita una strategia migliore. Le regole di ragionamento pratico possono essere divise in quattro classi: *reactive-rules* che possono essere usate non solo per reagire alla situazione corrente, ma anche per creare nuovi goal; *plan-rules* che sono usate per trovare piani per gli achievement goal; *failure-rules* che sono usate per ripianificare quando i piani falliscono; *optimisation-rules* che possono rimpiazzare piani meno efficaci con piani ottimali.

$$PRType ::= reactive|failure|plan|optimisation$$

Una regola di ragionamento pratico è formata da una testa, opzionale, che è un goal, un body, anch'esso opzionale, che è un goal, una guardia che è un belief, e un tipo per definire il suo scopo.

Informalmente, una regola di ragionamento pratico con testa $g$, body $p$ e guardia $b$ stabilisce che se l'agente tenta di soddisfare il goal $g$ e si trova nella situazione $b$, allora si deve considerare se *rimpiazzare* $g$ con un piano $p$ che è il mezzo per soddisfarlo. Se si ha una *plan-rule*, il goal $g$ è della forma *achieve* $s$ dove $s$ è una formula semplice e la regola stabilisce che per soddisfare $g$ nella situazione $b$ si deve considerare il piano $p$. Se si ha una *failure-rule*, allora $g$ può essere un qualsiasi goal e la regola stabilisce che se $g$ fallisce nella situazione $b$ si deve cancellare $g$ e al suo posto adottare la strategia $p$ per trattare il fallimento. Infine, se si ha una *optimisation-rule* allora $g$ può essere un qualunque goal e la regola afferma che se $g$ non è efficiente nella situazione $b$ si può procedere eliminando $g$ e adottando al suo posto la strategia $p$. Formalmente, una *reactive-rule* ha la testa vuota e può essere applicata nel momento in cui la guardia è vera.

*PRrule*

$head, body : optional[Goal]$
$guard : Belief$
$type : PRType$

---

$head = \varnothing \Leftrightarrow type = reactive \wedge$
$thehead \in (\mathrm{ran}\, achieve) \wedge body \neq \varnothing \Leftrightarrow type = plan$

Le variabili che compaiono nella testa e nella guardia di una regola sono dette *variabili globali* della regola, e quelle che compaiono nel body ma non nella testa o nella guardia, sono dette *variabili locali*.

### 1.2.2 Agenti 3APL e operazioni

Un *agente* è caratterizzato specificando belief, goal, regole di ragionamento pratico e competenze (repertorio di azioni di base). La differenza principale tra queste componenti dell'agente è che i primi due insiemi sono aggiornati dinamicamente mentre gli ultimi due sono fissi e non cambiano. Quindi un agente è definito come un'entità che consiste dei due insiemi statici *expertise* e *rulebase*.

*Agent*

$$expertise : \mathbb{P}\, Action$$
$$rulebase : \mathbb{P}\, PRrule$$
$$\forall\, a : expertise \bullet actionvars\ a = \varnothing$$

I belief e i goal dell'agente sono memorizzati negli insiemi *beliefbase* e *goalbase* rispettivamente. Questi due insiemi concorrono a formare lo stato mentale dell'agente (o *agent state*) che viene aggiornato durante l'esecuzione dell'agente stesso. Le operazioni compiute durante l'esecuzione dell'agente possono cambiare solo il suo stato mentale e non le regole o le competenze.

*AgentState*

$$Agent$$
$$beliefbase : \mathbb{P}\, Belief$$
$$goalbase : \mathbb{P}\, Goal$$

Un agente 3APL specifica le competenze e un insieme di regole di ragionamento pratico, ma non specifica beliefbase o goalbase iniziali dell'agente. Il primo passo nell'operare di un agente è inizializzare lo stato mentale.

Il funzionamento di un agente è parametrizzato mediante due nozioni semantiche. La prima nozione indica che la semantica delle azioni di base è definita dalla funzione globale *execute*. Questa funzione specifica che un'azione di base è un *operatore di aggiornamento* sui belief dell'agente. *execute* è tale che dati due agenti capaci di compiere un'azione, si garantisce che essi fanno la stessa cosa se eseguono tale azione.

$$execute : Action \times \mathbb{P}\, Belief \nrightarrow \mathbb{P}\, Belief$$

La seconda nozione indica l'esistenza di una relazione di conseguenza logica $LogCon\_$ che determina quali implicazioni l'agente è autorizzato a derivare dai suoi belief. Questa relazione è tra due insiemi di belief tali che il primo insieme di belief implica tutti i belief del secondo insieme ed è una relazione globale. Essa fa in modo che tutti gli agenti traggano conclusioni dai propri belief nella stessa maniera, garantisce cioè una quantità minima di consistenza globale.

$$| \quad LogCon\_ : \mathbb{P}(\mathbb{P}\, Belief \times \mathbb{P}\, Belief)$$

Le regole di ragionamento pratico forniscono agli agenti 3APL delle *capacità riflessive*. Le regole possono essere usate per pianificare, revisionare e creare goal. L'applicazione di una regola $r$ a un goal $g$ consiste nel rimpiazzare un subgoal $g'$ che corrisponde alla testa della regola $r$ con il body della regola $r$, se la testa della regola è non vuota. Se il body della regola è vuoto, il subgoal è eliminato. L'applicazione consiste in una sostituzione che è applicata all'intero goal risultante. Nel caso in cui la testa di una regola sia vuota, solo la guardia della regola deve essere derivabile dai belief dell'agente e un nuovo goal viene aggiunto all'insieme dei goal dell'agente.

Si suppone che $g'$ sia un subgoal di un goal $g$ che appare all'inizio del goal $g$, e $g'$ sia compatibile con la testa della regola $r$. Il compito è trovare un front context $fc$ tale che se il subgoal $g'$ è inserito al posto di $\Box$, il goal risultante è identico a $g$. Applicare $r$ è come *aggiornare* il $fc$ con il corpo della regola. C'è differenza tra inserire e aggiornare. Inserire un goal all'inizio di un context significa sostituire il goal con il $\Box$ nel front context, mentre aggiornare un context con un goal significa rimpiazzare il $\Box$ con tale goal e anche completare la scelta fatta (perseguire un subgoal in un goal scelto significa completare il ramo in cui il subgoal appare nel goal scelto).

Una regola è *applicabile* se la testa unifica un (sub)goal dell'agente $e$ la guardia della regola segue dai belief dell'agente. Se la regola non ha testa la si può applicare semplicemente se la guardia segue dalla beliefbase.

In base a questa definizione si può specificare la regola di applicazione. Se la testa della regola è non vuota, applicare la regola è come rimpiazzare un subgoal con il corpo della regola. Altrimenti, semplicemente si aggiunge il corpo della regola alla goalbase dell'agente. Si deve fare attenzione all'interferenza di variabili presenti nelle regole con variabili presenti nei goal. Per questa ragione tutte le variabili che compaiono nelle regole applicate sono ridenominate con variabili che non compaiono nel goal target.

L'*esecuzione di un goal* è specificata mediante i passi di computazione che un agente può compiere su un goal. Un passo di computazione corrisponde a una semplice azione dell'agente che può essere un'azione di base o anche una query sui belief dell'agente.

L'agente è autorizzato ad eseguire solo azioni di base o query che compaiono all'inizio di un goal. Il front context è utile per trovare un'azione o una query che l'agente possa eseguire. Se c'è un front context $fc$ in cui possono essere inserite in $\Box$ un'azione di base o una query, e che risulta come goal dell'agente, l'agente deve eseguite tale azione o query. Dopo l'esecuzione del goal, il goal necessita di aggiornamento, e tale aggiornamento si fa rimuovendo $\Box$ dal front context.

Eseguire un'azione di base è come cambiare la beliefbase dell'agente in accordo con la funzione *execute*. Le query sono goal per verificare se certe

condizioni seguono dalla beliefbase dell'agente. Le variabili libere nella condizione della query possono essere usate per recuperare dati dalla beliefbase. I valori recuperati sono memorizzati in una sostituzione $\vartheta$. Una query può essere eseguita solo se è una conseguenza della beliefbase, altrimenti non accade nulla. L'esecuzione di un goal è definita come la disgiunzione di queste due funzioni.

$$ExecuteGoal == ExecuteBasicAction \vee ExecuteQueryGoal$$

**Semantica**

La semantica operazionale del linguaggio 3APL è definita formalmente usando un sistema di transizione. Si definisce lo stato di un agente 3APL come una quadrupla $\langle \Pi, \sigma, \Gamma, \vartheta \rangle$ dove $\Pi$ è l'insieme dei goal, $\sigma$ è l'insieme dei belief, $\Gamma$ è l'insieme delle regole di ragionamento pratico e $\vartheta$ è una sostituzione formata dalle associazioni tra variabili che compaiono nei belief e nei goal dell'agente.

Le associazioni tra variabili $\vartheta$ sono passate da uno stato al successivo e possono essere aggiornate o meno a seconda del tipo di regola di transizione usata per determinare il passaggio di stato. L'inisieme completo delle regole di transizione si trova in [30].

### 1.2.3 Implementazioni

Ci sono due implementazioni dell'interprete per 3APL:

- **3APL Platform release** in Java [15]. Questa piattaforma è uno strumento sperimentale, progettata per supportare sviluppo, implementazione ed esecuzione degli agenti 3APL. Fornisce un'interfaccia grafica che consente all'utente di sviluppare ed eseguire gli agenti 3APL usando diverse funzionalità, quali un editor che evidenzia la sintassi con i colori e vari strumenti per il debug. La piattaforma consente la comunicazione tra agenti. Essa può girare su diverse macchine connesse in una rete in modo che agenti ospitati su piattaforme 3APL possano comunicare con altri. 3APL Platform è stata testata su sistemi Windows 98, Windows NT e Windows XP, su sistemi Linux e Unix (Solaris). È scritta in Java 2 SDK 1.4 e usa il Prolog engine di JIProlog che è anch'esso scritto in Java.

- **3APL Interpreter** prototipo scritto in *Haskell* e sviluppato da D. Leijen [16]. È stato creato per sperimentare nuove potenzialità di 3APL e offre funzionalità interessanti, in quanto

  - supporta pianificazione e riflessione, ovvero esegue ed esamina i goal dinamicamente;

- supporta componenti esterne arbitrarie, scritte in qualsiasi linguaggio;
- usa agenti multipli, concorrenti e comunicanti.

Poiché è un prototipo non implementa pienamente la specifica di 3APL, non ha un ambiente grafico, come la release Java, ma un'interfaccia a linea di comando, e supporta solo una versione di Prolog essenziale (priva di floating point, string, library, ecc.). Attualmente è compatibile solo con sistemi Windows.

I programmi 3APL scritti per le due versioni hanno delle differenze sintattiche anche se minime.

Non risultano esserci applicazioni reali sviluppate in 3APL.

### 1.2.4 Estensioni

Ci sono varie proposte per ricerche future. Una riguarda l'estensione del linguaggio per agenti con comunicazione e funzionalità di tipo multi-agent [32]. Un'altra area di ricerca si occupa di costruire un metodo per la progettazione degli agenti basato su una semantica denotazionale e sulla corrispondente prova teorica. Si vogliono migliorare comprensione e utilizzo delle regole di ragionamento pratico, dando loro, inoltre, una classificazione, un'ordine di priorità, eventualmente dinamico, e un modo per assegnare tale priorità [30].

## 1.3 JACK

JACK$^{TM}$ Intelligent Agents è un sistema ad agenti di terza generazione ed è stato sviluppato nel 1999 da Agent Oriented Software Group (AOS Group) [12], lo stesso gruppo che ha lavorato sui sistemi delle generazioni precedenti, PRS e dMARS.

JACK è un ambiente di sviluppo *Agent Oriented* costruito sul linguaggio di programmazione Java e integrato con esso. Oltre alle componenti dell'ambiente di sviluppo Java offre specifiche estensioni per implementare il comportamento degli agenti. In definitiva, JACK è stato sviluppato in modo da fornire estensioni agent-oriented al linguaggio Java. Gli agenti JACK sono quindi *agenti intelligenti* che ragionano in accordo con il modello BDI di cui si è parlato brevemente in precedenza.

Un agente JACK ragiona rispondendo a stimoli pro-attivi (è goal directed) e reattivi (è event driven). Ogni agente ha

- un insieme di conoscenze relative al mondo (è un data set);
- un insieme di eventi a cui dovrà reagire;

- un insieme di goal che desidera soddisfare (possono essere richieste di un agente esterno, conseguenze di un evento, o un cambiamento nei propri belief);
- un insieme di piani che descrivono come gestire goal o eventi.

JACK è costituito dalle seguenti tre componenti chiave:

**JACK Agent Language (JAL)** che è il linguaggio di programmazione usato per descrivere il sistema software agent-oriented. JAL è un super-insieme di Java che ne copre l'intera sintassi estendendola con costrutti che rappresentano le funzionalità agent-oriented. Una sua sintetica descrizione viene fornita nella Sezione 1.3.1.

**JACK Agent Compiler** che pre-processa i files sorgente scritti in JAL e li converte in Java puro. Questo codice sorgente Java può essere poi compilato in codice per la Java virtual machine.

**JACK Agent Kernel** è il runtime engine per i programmi scritti in JAL. Fornisce un insieme di classi che forniscono ai programmi JAL la loro funzionalità agent-oriented. Molte di queste classi "girano dietro la scena" e implementano l'infrastruttura sottostante e le funzionalità richieste dagli agenti, mentre altre sono usate esplicitamente nei programmi JAL, ereditate e richiamate per provvedere alle richieste degli agenti con le loro uniche funzionalità.

### 1.3.1 Il JACK Agent Language (JAL)

Il JACK Agent Language estende la sintassi Java per permettere al programmatore di sviluppare le componenti che sono necessarie per definire gli agenti BDI e il loro comportamento. Ognuno di questi elementi è implementato con una classe Java che eredita certe proprietà fondamentali da una classe base che viene estesa per poter ottenere quanto serve nel caso specifico. Queste classi base sono definite nel kernel e sono tali da avere una propria sintassi che non è presente in Java.

I costrutti tipici di JAL sono raggruppati nelle seguenti categorie:

- Classes (tipi);
- Declarations (#-dichiarazioni);
- Reasoning Method Statements (@-statements).

Inoltre ogni classe JAL fornisce dei normali attributi e metodi Java che possono essere utili nei programmi JACK. Di seguito si vedono meglio le varie categorie.

**JACK Agent Language Classes** definiscono le unità funzionali entro JACK. Sono implementate come classi Java con metodi privati, per la gestione delle loro proprietà agent-oriented, oltre a quelli base. Le classi, o tipi, tipiche di JACK sono:

**Agent** che modella la principale entità per il ragionamento in JACK;

**Capability** che aggrega le componenti funzionali di cui l'agente può far uso (eventi, piani, insieme dei belief e altre capacità);

**Event** che modella messaggi e avvenimenti cui gli agenti devono saper far fronte. Gli eventi possono arrivare dall'esterno mediante messaggi da altri agenti, o dall'interno come conseguenza di azioni proprie dell'agente o in risposta a suoi goal interni;

**Plan** che modella la procedura che un agente segue per gestire un dato evento. Tutte le azioni che un agente intraprende sono prescritte e descritte dai piani dell'agente stesso;

**BeliefSet** che modella la conoscenza dell'agente come belief che seguono la semantica data entro un mondo chiuso (in cui si ha "onniscienza") o un mondo aperto (in cui esiste il concetto di sconosciuto). L'insieme dei belief rappresenta i belief dell'agente come tuple che sono relazioni nella logica del prim'ordine e devono essere mantenute logicamente consistenti.

Ogni classe JAL offre attributi e metodi che possono essere usati dai programmi.

**JACK Agent Language Declarations (#-declarations)** definiscono proprietà agent-oriented di un tipo JAL o relazioni e dipendenze esistenti tra tipi JAL. Sono usate per specificare relazioni tra classi in un programma JACK, ad esempio quali piani sono usati dall'agente e quali eventi l'agente gestisce.

**Reasoning Method Statements (@-Statements)** sono istruzioni specifiche di JAL che compaiono nei *reasoning methods*. Descrivono le azioni che l'agente può compiere per seguire un comportamento. Passi, come notificare eventi, mandare messaggi ad altri agenti o attendere finché una specifica condizione diventi vera, sono espressi usando i Reasoning Method Statements.

Elenchi e descrizioni dettagliati di attributi e metodi delle JAL Classes, delle JAL Declarations e dei Reasoning Method Statements si trovano all'indirizzo `http://www.agent-software.com/shared/resources/index.html`

### Esecuzione

Il supporto run-time, come anticipato sopra, è dato dal JACK Agent Kernel. Esso si occupa di:

- gestione automatica della concorrenza tra compiti che vanno eseguiti in parallelo (le *intention* nella tecnologia BDI);
- guidare comportamento di default dell'agente in reazione agli eventi, ai fallimenti delle azioni e dei compiti (task), e così via;
- fornire un'infrastruttura per le applicazioni multi-agente che sia leggera e nativa ed abbia performance di alto livello per le comunicazioni usando un semplice naming scheme.

Quando si istanzia un agente nel sistema, esso resta in attesa di ricevere un goal o di reagire ad un evento. Quando riceve un evento, o un goal, l'agente comincia la gestione dell'evento e, se questo non è già stato gestito, cerca il piano adatto per la sua gestione. A seconda del tipo di evento, l'agente esegue il piano o i piani. La gestione degli eventi può essere sincrona o asincrona a seconda di come l'evento viene notificato. Nell'esecuzione del piano sono coinvolte interazioni con relazioni nel beliefset dell'agente o altre strutture dati Java. Il piano in esecuzione può a sua volta iniziare altri subtask (nuove intenzioni), che ne possono iniziare altri e così via. I piani possono terminare con successo o fallire. In certe circostanze, se il piano fallisce, l'agente può provare un altro piano.

Entro un solo processo si possono avere molteplici agenti e ogni agente può avere diverse *task queue* (coincidenti con gli intention stacks in BDI). Una task queue è generata quando l'agente riceve un evento asincrono e questa contiente i passi (o task) che sono richiesti perché l'agente gestisca l'evento. Questi passi sono specificati in un piano. L'esecuzione di un task può risultare in eventi notificati successivamente. Se l'evento è notificato in modo sincrono, i task risultanti sono aggiunti in testa alla task queue che ha generato tale evento. Se l'evento è notificato in modo asincrono, si genera una nuova task queue.

Un agente 'circola' entro le task queue secondo le indicazioni del task manager. Questo è gestito dal kernel entro il thread di esecuzione di JACK. Questo non impedisce agli altri thread nel programma Java di chiamare i metodi degli agenti.

Ci possono anche essere agenti in processi distinti e su macchine diverse che comunicano con ogni altro.

### Sviluppo di un'applicazione con JACK

Per costruire un'applicazione in JACK si devono innanzi tutto individuare le componenti distribuite del sistema e la ripartizione delle funzionalità. A tal fine vanno considerati un certo numero di vincoli, quali l'esistenza di sistemi ereditari o di una specifica infrastruttura di comunicazione. Quindi, una volta che si sono identificate le funzionalità che l'agente deve fornire e si è scelto il modello BDI per la realizzazione dell'applicazione si compiono, non necessariamente in ordine, queste due azioni:

- si identificano le classi elementari che servono per manipolare le risorse usate dall'agente. Tali risorse potrebbero essere esterne (basi di dati relazionali, Internet, braccia di un robot, una GUI e così via) così come interne (specifiche strutture dati matematiche per rappresentare informazioni finanziarie o spaziali);
- si identificano quegli elementi che costituiscono gli stati mentali dell'agente. Questi comprendono il trovare:
  - quali eventi esterni guidano l'agente (includendo messaggi da altri agenti);
  - quali goal l'agente può fissare per sé;
  - quali belief influenzano l'adozione di piani;
  - le procedure (che in termini BDI sono i piani) richieste per portare a compimento dei task, soddisfare dei goal e reagire a degli eventi nei vari contesti possibili.

Di seguito si mostra come il codice JACK sia una estensione di Java stratificata mediante un esempio [12].

L'esempio mostra agenti che fanno "ping" su altri agenti come scambio di messaggi vuoti. Il messaggio è rappresentato come un evento originato da un agente e ricevuto da un altro.

L'evento è dichiarato come segue

```
event PingEvent extends MessageEvent {
   int value;
   #posted as
   ping(int value)
   {
       this.value = value;
   }
}
```

Per la creazione della classe, al posto della parola chiave di Java si usa la parola chiave `event` e l'evento è dichiarato come un `MessageEvent`, che indica che esso può essere mandato a un altro agente. La dichiarazione `#posted as` indica come viene generato l'evento (in questo caso, invocando il metodo Java, `ping()` con un parametro intero).

Si mostra un agente con un singolo piano che gestisce un singolo evento

```
agent PingAgent extends Agent{
   #handles event PingEvent;
   #uses plan PingPlan;
   #posts event PingEvent pev;
```

```
   void ping(String other)
   {
      send(other, pev.ping(1));
   }
}
```

Quando viene chiamato il suo metodo `ping(String other)` esso notifica l'evento `PingEvent` con valore 1 all'agente `other`. Se quest'ultimo è un altro `PingAgent` sarà invocato il piano `BouncingPlan`, verrà mandato indietro un `PingEvent` con valore 2 e così via all'infinito.

L'applicazione può instanziare tutti gli agenti `PingAgent` che desidera. I nomi degli agenti e i loro indirizzi sulla rete sono determinati dal meccanismo di comunicazione in uso.

Si mostra poi un piano che gestisce la notifica dell'evento e risponde al suo mittente 'rimbalzando' l'evento indietro. Questo esempio non è sensibile al contesto (non ci sono restrizioni sullo stato dei belief dell'agente per la sua applicabilità).

```
plan BouncingPlan extends Plan {
   #handles event PingEvent ev;
   #sends event PingEvent per;

   body()
   {
      @send ( ev.from, pev.ping(ev.value + 1) );
      //reply to the sender of the event
   }
}
```

### 1.3.2 Semantica

Il JACK Agent Language ha una semantica informale che supporta il modello di esecuzione Belief Desire Intention la cui descrizione si trova nel JACK$^{TM}$ Intelligent Agents Agent Manual[1].

### 1.3.3 Implementazioni

JACK è implementato interamente in Java. Il AOS Group fornisce JACK che supporta le seguenti piattaforme: Windows 9x, Millenium edition (Me), NT, 2000, XP; Solaris 2.6, Solaris 7 (2.7), Solaris 8 and Solaris 9; Linux i386 (libc5 and libc6) e Mac OS X. JACK è Java puro e può girare su ogni piattaforma Java-based. JACK memorizza tutti i file per i programmi e i dati come testo normale, consentendo di usare strumenti standard per la gestione delle configurazioni e delle versioni.

Attualmente si è alla versione JACK 4.2 che comprende il JACK Compiler, il JACK Teams, il JACK Development Environment, un potente strumento per la progettazione, uno sviluppatore grafico di piani, un visualizzatore del tracciato dei piani in esecuzione, un ambiente run-time, un editor di supporto esterno. JACK offre eccellenti funzionalità per la sicurezza dovute al fatto di usare la piattaforma Java. Inoltre gli agenti JACK gestiscono i loro dati privatamente e pertanto questi non possono essere acceduti da altri processi senza un'autorizzazione che si ha solo se prevista dal progettista del sistema.

Al fine di promuovere l'uso delle tecnologie basate su agenti intelligenti l'AOS Group fornisce delle speciali licenze accademiche per JACK alle università. Informazioni relative a distribuzione e installazione di JACK sono disponibili agli indirizzi `http://www.agent-software.com` o `http://www.agent-software.co.uk`.

Una delle applicazioni più significative di JACK è un veicolo aereo, privo di controllo umano, (Unmanned Aerial Vehicle, UAV) guidato da un agente intelligente software realizzato in JACK che risiede a bordo e dirige il pilota automatico dell'aereo durante la missione.

### 1.3.4 Estensioni

Si prevede l'uscita della release **JACK v5.0** a cui verranno aggiunti: uno strumento per il debug che consenta la visualizzazione del run-time, un visualizzatore del tracciato dell'esecuzione che usi design diagrams, la generazione di report e una piattaforma che supporti un maggior numero di font.

## 1.4 AgentTalk

*AgentTalk* è un semplice linguaggio per la programmazione di sistemi agent-based. È stato sviluppato da M. Winikoff nel 2001 [76]. AgentTalk è un prototipo per l'interpretazione di AgentSpeak(L) che ha lo scopo il testare concetti agent-oriented e come obiettivi

1. essere estensibile;
2. essere semplice;
3. catturare l'essenza di sistemi BDI implementati quali JACK e dMARS.

e quindi non è stato concepito per essere una piattaforma per agenti utilizzabile realmente.

Il progetto base di AgentTalk ha molto in comune con *AgentSpeak(L)*. Nel paradigma BDI gli agenti hanno Belief, Desire e Intention. In AgentTalk la corrispondenza con i concetti base del BDI si è persa: gli eventi

rappresentano i goal (comunque, restano differenze significative tra essi), i piani che un agente sta eseguendo correntemente corrispondono (in modo flessibile) alle sue intenzioni. Per scrivere un sistema agent-oriented in AgentTalk si costruisce un *plan file* contenente la sequenza dei piani che si useranno (attualmente tutti gli agenti hanno lo stesso insieme di piani simultaneamente), poi si manda in esecuzione il sistema creando gli agenti, scrivendo i loro belief e mandando loro gli eventi. Gli eventi a cui un agente risponderà sono implicitamente definiti dai piani che possiede.

Un *piano* è formato da un *evento* da cui è attivato, da una sequenza di *context condition* che devono essere verificate rispetto ai belief dell'agente affinché il piano sia applicato, e un *plan body* formato dalla sequenza delle azione da eseguire. Nel momento in cui un agente rileva un evento esso cerca i piani che possono essere attivati da quell'evento e poi prova ogni piano, seguendo l'ordine con cui i piani sono definiti nel plan file. Per ogni piano si valutano le context condition e poi si esegue il plan body. Se tali valutazioni falliscono si prova il piano applicabile successivo per l'evento. Una volta che il plan body è completato i piani alternativi sono scaricati.

I sistemi ad agenti in AgentTalk sono concorrenti e quindi si deve tenere conto degli stati di esecuzione.

Nel caso in cui un evento non possa essere associato ad alcun piano applicabile, tale evento viene scartato.

### 1.4.1 Il linguaggio AgentTalk

L'implementazione corrente di AgentTalk è Scheme based (usa l'implementazione DrScheme [60, 50]) e per rappresentare i piani usa S-expressions [65]. I programmi AgentTalk sono conformi alla seguente grammatica BNF in cui gli elementi **bold** sono letterali.

⟨ plan file ⟩ ::= ⟨ plan ⟩⟨ plan file ⟩|⟨ plan ⟩
⟨ plan ⟩ ::= **(**⟨ event ⟩**(**⟨ belief ⟩**)(**⟨ plan body ⟩**))**
⟨ event ⟩ ::= *any scheme S-expression*
⟨ belief ⟩ ::= empty |⟨ belief ⟩⟨ belief ⟩
⟨ belief ⟩ ::= *any scheme S-expression*
⟨ variable ⟩ ::= **?**name
⟨ plan body ⟩ ::= empty |⟨ action ⟩⟨ plan body ⟩

| | |
|---|---|
| ⟨ action ⟩ ::= **(scheme** ⟨ scheme expression ⟩**)** | Evaluates the *scheme expression* |
| \| **(eval** ⟨ variable ⟩⟨ scheme expression ⟩**)** | Evaluates the *scheme expression* and binds the result to the *variable* |
| \| **fail** | Fail |
| \| **(assert** ⟨ belief ⟩ **)** | Adds the belief to the agent's belief |
| \| **(retract** ⟨ belief ⟩ **)** | Retracts *all* belief which match with the given belief |
| \| **(message** *any scheme expression***)** | Displays the object (without evaluating it) |

| | |
|---|---|
| \| **(send** *agent-name* ⟨ event ⟩ **)** | Sends the event to the named agent as a new top-level event |
| \| **(post** ⟨ event ⟩ **)** | Posts the event as a new top-level event to this agent. Same as (**send** *this-agent's-name* ⟨ event ⟩ ) |
| \| ⟨ event ⟩ | Sends the event to this agent as a sub-plan of the current plan. |

### 1.4.2 Semantica

Un agente in esecuzione manipola una struttura, che rappresenta un'intenzione, la quale è formata da una sequenza di alternative. Ogni alternativa contiene una sequenza di piani che consiste di una sequenza di azioni. Un esempio di intenzione è `[ALT((q;r;x), (r;x)), ALT([q;er])]`.

L'esecuzione segue la seguenti equazioni:

```
exec([ALT(act;acts, alts), rest]) = exec1(act, [alt(alts), rest])
exec([ALT([], alts), rest]) = exec(rest)
exec([]) = success
exec1(fail,[ALT(alt,alts), rest]) = exec([ALT(alts), rest])
exec1(fail,[ALT(), rest]) = exec1(fail, [rest])
exec1(fail,[]) = fail
exec1(action,X) = exec1(fail, X), if action fails
exec1(action,alts) = exec(alts), if action succeeds
exec1(event,alts) = exec([ALT(get(event)), alts])
```

Supponendo che per l'evento `event` si abbiano i piani

```
(event1 context1 body1)
(event2 context2 body2)
(event3 context3 body3)
```

si unifica `event` con ognuno degli `event1`, `event2` e `event3`; si controlla se ognuna delle context condition è vera. Le clausole che passano tutti i tests (unificando l'evento con l'evento, e controllando le context condition) sono collocate in una lista di alternative e aggiunte davanti all'intenzione.

```
get-alternative-plans(event)
= map(get-body-part,filter(check(event),
map(cp-with-unique-vars,*program*)))

check(event,clause)
= unify(head(clause),event) and valid(get-context(clause))
```

### 1.4.3 Implementazioni

L'implementazione di AgentTalk è fatta usando DrScheme. Può girare sotto Windows, Unix e MacOS.

L'interfaccia grafica di AgentTalk consente di

1. caricare e, eventualmente, ricaricare un plan file e scriverlo con un built-in editor;

2. aggiungere agenti, inviare loro eventi e inserire/eliminare belief;

3. fare il tracing del sistema
   - mandando in esecuzione gli agenti,
   - richiedendo il sommario degli agenti e del loro stato,
   - eseguendo l'agente un singolo passo alla volta.

L'editor built-in consente di editare i plan file senza ricorrere a un editor esterno. Questo editor al salvataggio del file lo ricarica automaticamente in AgentTalk.

### 1.4.4 Estensioni

Sono previste estensioni volte al miglioramento di vari aspetti.

- Assegnare staticamente delle priorità sia a piani sia a eventi.

- Modificare l'esecuzione dei metodi in modo che le context condition siano controllate quando il piano è scelto per l'esecuzione e non prima. Inoltre, escludere i piani possibili solo quando falliscono realmente e non quando fallisce la context condition. L'algoritmo di esecuzione per un evento dovrebbe essere il seguente:

  1. collocare i piani che si associano all'evento in una lista;
  2. prendere un piano;
  3. se la context condition ha successo si prova ad eseguire questo piano, altrimenti se ne prende un'altro. Se nessun piano ha una context condition che viene soddisfatta, allora fallisce;
  4. se il piano selezionato fallisce lo si rimuove dalla lista delle possibili alternative e si riparte dal passo 2.

- Aggiungere nuovi costrutti al linguaggio.

- Separare i piani tra i diversi agenti.

- Inserire plan packages e user interface per assegnare plan packages agli agenti.

- Adeguare sintassi e parser in modo da rendere l'uso più semplice e la segnalazione degli errori sintattici più corretta.
- Migliorare il controllo sugli errori.
- Aggiungere la possibilità di cancellare agenti (o consentire di mostrare solo alcuni agenti).
- Aggingere event tracing option.
- Migliorare l'efficienza:
  1. non tentare l'unificazione con tutte le clausole;
  2. non tentare l'unificazione con tutti i belief;
  3. ridurre l'ammontare dell'avvicendamento (rotating).
- Portare AgentTalk su altri linguaggi/piattaforme/implementazioni (Kawa (Java), Prolog, Lygon, LispMe).

## 1.5 JAM

JAM è un'architettura per agenti intelligenti sviluppato da M. J. Huber (2001) [35, 34] dell'Intelligent Reasoning Sistems (IRS) [33]. JAM combina gli aspetti migliori delle molteplici teorie leading-edge e dei framework per agenti intelligenti. JAM ha subito molte influenze da parte di:

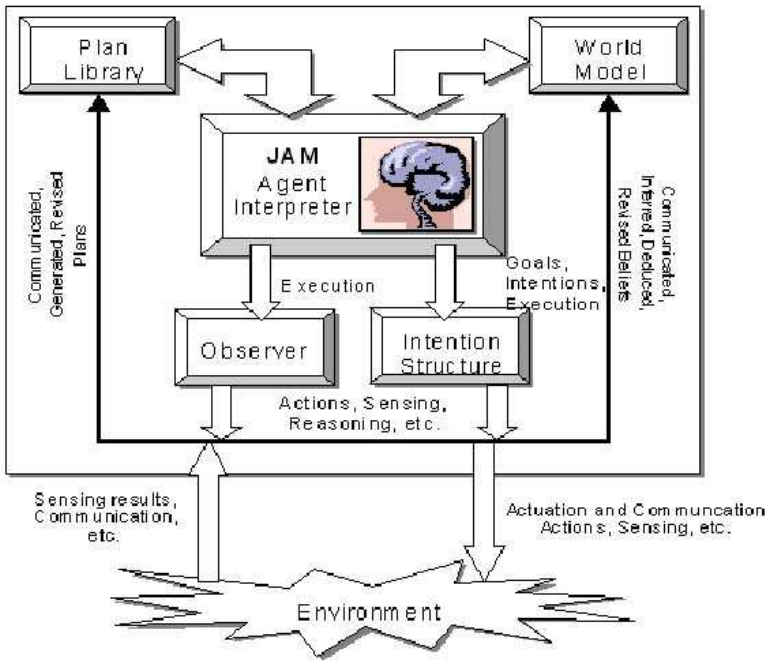
- teorie BDI;
- architetture per agenti intelligenti basate sul PRS;
- UMPRS, implementazione di PRS realizzata dall'Università del Michigan [44];
- PRS-CL della SRI International [55];
- ACT plan interlingua della SRI International [57];
- rappresentazioni Structured Circuit Semantics (SCS) di Lee e Durfee [44];
- aspetti di mobilità da Agent TCL [29], Agents for Remote Action (ARA), Aglets e altri.

JAM fornisce rappresentazioni procedurali e di piani ricche ed estese, ragionamento basato sull'utilità e un metalivello su goal multipli e simultanei, comportamento goal-driven e data-driven che sorge da una amalgama delle caratteristiche dei prodotti indicati in precedenza. L'architettura JAM fornisce anche una funzione primitiva agentGo che usa il meccanismo di serializzazione degli oggetti Java per fornire caratteristiche di mobilità.

### Panoramica su JAM

Ogni agente JAM è composto di cinque elementi principali:

- il *modello del mondo (world model)* è un database che rappresenta i belief dell'agente;
- la *libreria dei piani (plan library)* è una collezione di piani che l'agente può usare per poter raggiungere i propri goal;
- l'*interprete (interpreter)* è la "mente" dell'agente che ragiona su cosa l'agente potrebbe fare e dove e come farlo;
- la *struttura per le intenzioni (intention structure)* è un modello interno dei goal correnti dell'agente e tiene traccia dell'impegno di, e del progresso nel, portare a compimento questi goal;
- l' *osservatore (observer)* è una procedura dichiarativa leggera e specificata dall'utente che l'agente esegue in interleaving con i passi del piano (in aggiunta al ragionamento eseguito dall'interprete JAM) per eseguire funzionalità estranee allo scopo del normale ragionamento di JAM basato su piani o goal.




In JAM, cambiare il world model o porre nuovi goal genera ragionamenti mirati alla ricerca di piani che possono essere applicati alla situazione corrente. L'interprete di JAM sceglie un piano dalla sua lista di piani applicabili basandosi sul proprio metalivello di ragionamento e sulla massima

utilità, si propone di fare questo e quindi esegue la prima intenzione trovata con utilità più alta.

#### Checkpointing e mobilità

Gli agenti JAM facilitano la costruzione di applicazioni richiedenti mobilità mediante l'uso di funzionalità di *checkpointing*. Sono state implementate funzionalità per catturare lo stato a runtime di un agente JAM nel corso dell'esecuzione e funzionalità per ripristinare in seguito questo stato catturato nel corso dell'esecuzione. Un uso di queste funzionalità è salvare periodicamente lo stato dell'agente in modo da poterlo ripristinare nel caso in cui l'agente fallisse inaspettatamente. Questo consente di costruire applicazioni robuste che possano essere riavviate e recuperate da terminazioni altrimenti catastrofiche. Un altro uso della funzionalità di checkpointing implementa la mobilità dell'agente, per cui l'agente crea un checkpoint e ripristina la propria esecuzione su un computer differente. Un terzo possibile uso di tale funzionalità è clonare un agente creando un checkpoint e riprendere l'esecuzione da uno stato di esecuzione senza terminare l'agente originale. In tutti i casi, la funzione di ripristino di base è fornita da una semplice classe Java. Politiche di mobilità specifiche per l'applicazione si possono ottenere estendendo questa classe di ripristino e, se necessario, si possono realizzare altre funzionalità simili.

La mobilità dell'agente è stata realizzata implementando una funzione primitiva *agentGo*. Questa consente al programmatore dell'agente di specificare un computer e una porta: quando il piano contenente la funzione è eseguito, l'agente si trasferisce sulla nuova macchina e termina la propria "esistenza" sul computer originario. Sulla macchina destinazione, l'agente JAM riprende l'esecuzione, guidato da goal e piani preesistenti, dal punto in cui si era sospeso sulla macchina iniziale. Il movimento tra i computer diventa trasparente nel senso che tale attività non è gestita in modo diverso da ogni altra attività.

### 1.5.1 Architettura JAM e suo funzionamento

L'*interprete* JAM è responsabile della selezione ed esecuzione dei piani in base a intenzioni, piani, goal, e belief circa la situazione corrente. All'interprete è associata la struttura per le intenzioni, uno stack run-time di goal con e senza piani istanziati. Un agente JAM può avere un gran numero di piani alternativi per soddisfare ogni singolo goal e l'interprete ragiona su tutte le possibili combinazioni di piani, goal e associazioni di variabili prima di scegliere la migliore alternativa per la situazione particolare data. L'agente controlla tutti i piani che possono essere applicati a un goal per vedere se essi sono rilevanti per la situazione corrente. Questi piani che sono applicabili sono raccolti in quella che è detta Applicable Plan List (APL). Per

ogni piano instanziato in APL è determinato un valore di utilità, in modo che l'interprete possa secegiere per l'esecuzione il piano istanziato con la più alta utilità per il gaol corrente, se non ci sono piani di metalivello disponibili entro la APL. L'interprete JAM si comporta in modo tale che se il goal con l'intenzione nuova ha la priorità più alta tra tutti i goal con intenzioni, è eseguito il nuovo piano per il goal, altrimenti, se l'utilità più alta continua ad essere di una intenzione precedente, l'interprete esegue il piano di tale intenzione.

Se la generazione di una APL si risolve in più entrate, l'agente avvia un ragionamento al metalivello per decidere quale degli elementi in APL indirizzare alla struttura per le intenzioni. L'agente può prendere questa decisione in diversi modi tali da consentirgli di eseguire il ragionamento al metalivello sul proprio ragionamento al metalivello. Il ragionamento al metalivello finisce quando l'interprete genera un APL vuoto, cosa che indica che l'agente non ha strumenti a metalivelli più alti per decidere tra le alternative.

La distribuzione di JAM fornisce delle azioni primitive implementate per semplificare il ragionamento al metalivello. Queste servono per scegliere un'intenzione da una APL, trovare il goal correntemente perseguito, trovare il piano correntemente eseguito, estrarre attributi e valori di attributi da un piano, e indirizzare un'intenzione nella struttura per le intenzioni.

Un agente JAM ha un comportamento top-down che è determinato mediante la specifica di goal al top-level. La sintassi per un *goal* è la seguente:

```
goal_type goal_name parameter1 ...  parameterN
          <:UTILITY expression>;
```

Il goal_name è un'etichetta che identifica la relazione per il goal, i parametri sono gli argomenti della relazione per il goal. La parola chiave *:UTILITY* e l'espressione sono opzionali e forniscono un'opportunità per specificare sia un valore numerico, fissato, sia un calcolo di utilità complesso, in modo arbitrario. L'utilità del goal è combinata con quella di un piano istanziato per calcolare l'utilità totale dell'intenzione. Un agente JAM cambierà dinamicamente tra goal alternativi secondo il cambiamento dell'utilità dell'intenzione visto che si persegue sempre l'intenzione con utilità più alta. Il goal_type assume un valore tra *ACHIEVE*, *PERFORM*, *MAINTAIN* e *QUERY*, ognuno con la propria semantica che qui non viene descritta.

Inizialmente, al momento dell'invocazione, all'agente sono dati uno o più goal top-level. Questi *goal top-level* sono specificati mediante la parola chiave *GOALS* seguita da una lista di goal specificati nella forma descritta in precedenza. Questa lista di goal può aumentare durante l'esecuzione a causa della comunicazione con altri agenti, generata da ragionamenti interni da parte dell'agente, o a causa di altri mezzi.

I goal top-level sono *persistenti*. Essi infatti sono perseguiti finché sono soddisfatti da un'esecuzione di piano di successo o con altri mezzi in modo

opportuno, anche da altri agenti, o sono rimossi esplicitamente entro un piano. Se un piano per un goal top-level fallisce, l'agente ritira il suo impegno per quel goal rimuovendo l'intenzione per esso, ma lascia il goal nella struttura delle intenzioni per successivi tentativi di completamento del goal. In questo modo gli agenti hanno un livello di impegno per tutti i goal top-level e un livello forte specialmente per impegnarsi rispetto a quei goal che hanno intenzioni associate con loro.

Riguardo ai *subgoal*, questi sono goal che l'agente crea entro i piani durante la loro esecuzione. L'esecuzione dei subgoal è invocata entro i piani mediante le azioni `ACHIEVE`, `PERFORM`, `MAINTAIN` e `QUERY`. La generazione della APL e la gestione dell'intenzione per il subgoal è completata esattamente come per i goal top-level. L'esecuzione dei subgoal può essere eseguita a livelli di profondità arbitrari e può essere *ricorsiva*.

Una sostanziale differenza tra subgoal e goal top-level è che i subgoal non sono persistenti per default. Se un piano fallisce per un subgoal, l'interprete considera l'azione di esecuzione del subgoal come un fallimento.

Un *piano* JAM definisce una specifica procedurale per realizzare un goal, reagire a un evento, o seguire un comportamento, tenendo presente che un agente JAM è capace di entrambi i comportamenti goal-driven e data-driven.

La Struttura di base di un piano è mostrata di seguito:

```
PLAN: {
    GOAL: [goal specification]
        or
    CONCLUDE: [world model relation]
    NAME: [string]
    BODY: [procedure]
    <DOCUMENTATION: [string]>
    <PRECONDITION: [expression]>
    <CONTEXT: [expression]>
    <UTILITY: [numeric expression]>
    <FAILURE: [non-subgoaling procedure]>
    <EFFECTS: [non-subgoaling procedure]>
    <ATTRIBUTES: [string]>
}
```

I campi opzionali sono indicati tra i simboli "⟨" e "⟩". Inizialmente, al momento dell'invocazione dell'agente, l'agente ha uno o più piani. Sintatticamente i piani sono specificati con la parola chiave `PLANS` seguita da una lista di piani specificati come sopra. Questa lista di piani può aumentare nel corso dell'esecuzione per effetto della comunicazione con altri agenti, o per altre cause.

L'applicabilità di un piano è limitata da un goal particolare o da una conclusione data-driven. Ogni piano può essere ulteriormente forzato da una particolare *precondizione*, condizione che deve essere soddisfatta prima

della partenza dell'esecuzione del piano, e da un *contesto*, condizione che deve essere soddisfatta sia prima, sia durante l'esecuzione del piano. La procedura da usare per completare il goal è data nel *corpo* del piano, il quale può contenere azioni semplici e costrutti strutturati complessi.

Ogni piano può includere un calcolo di *utilità* implicito o esplicito, che è usato per influenzare la selezione di certe procedure su altre mediante il meccanismo di ragionamento al metalivello di default di JAM che è basato sull'utilità. L'utilità assume valori dello stesso tipo di quelli prevista per l'utilità nei goal. Un'altra componente opzionale è il campo *effects*, che è una procedura che l'interprete JAM esegue quando il piano è completato con successo. Questo campo è usato per l'aggiornamento del world model, o per altri costrutti procedurali diversi dall'esecuzione dei subgoal. La specifica della procedura che l'agente deve seguire quando un piano fallisce può essere rappresentata nella sezione opzionale *failure* del piano. Questa sezione è simile al campo "effects" in quanto contiene anch'essa una procedura che può contenere ogni componente di piano JAM escluse le esecuzioni di subgoal. Un uso tipico di questa sezione è quello di definire del codice di ripristino di uno stato consistente specifico per un particolare piano. Il campo opzionale *attributes* consente al programmatore di dare informazioni relative alle caratteristiche del piano su cui l'agente può ragionare durante l'esecuzione del piano e il ragionamento al metalivello. I campi *name* e *documentation* sono dei segnaposto rispettivamente per un identificatore unico e una documentazione testuale che potrebbero accompagnare il piano.

JAM fornisce diversi azioni e costrutti di programmazione. Le azioni sono definite come istruzioni su una linea singola e i costrutti sono istruzioni su linee multiple. Fornisce inoltre diversi costrutti di programmazione standard come le iterazioni, i salti condizionali, e l'assegnazione di valori alle variabili. I belief dell'agente JAM possono essere cambiati e controllati usando `ASSERT`, `FACT`, `RETRACT`, `RETRIEVE` e `UPDATE`. Prevede un'azione `PARALLEL`, per l'esecuzione simultanea di rami multipli all'interno dei piani usando thread Java separati mentre preserva l'esecuzione delle azioni in interleaving con il ragionamento dell'interprete. L'azione `WAIT` è usata per la sincronizzazione: fa in modo che l'esecuzione di un piano sia interrotta finché non si soddisfa un goal specifico o un'azione data non termina con successo.

Le funzionalità fornite da JAM possono essere estese dai programmatori definendo funzioni primitive in Java e usando i metodi di accesso al codice Java fornite dall'architettura JAM. La distribuzione di JAM fornisce anche delle azioni primitive predefinite che servono per il debug e la mobiltà.

Il *world model* contiene i fatti che rappresentano lo stato corrente del mondo così come è conosciuto dall'agente. Contiene informazioni quali variabili di stato, informazioni sensoriali, conclusioni da deduzione o inferenza, informazioni di modelling su altri agenti, ecc. Ogni entry del world model è

della forma

$$relation\_name\ argument1\ argument2\ ...\ argumentN;$$

Ordine, significato e tipaggio degli argomenti non sono vincolati e sono determinati dal programmatore dell'agente. Gli argomenti della relazione world model possono assumere questi tipi: stringhe, numeri in virgola mobile, numeri interi, e oggetti Java nativi. La specifica iniziale del world model è data creando un file di testo contenente la parola chiave `FACTS` seguita dalla lista di relazioni del world model. L'agente prima di iniziare l'esecuzione scandisce la specifica iniziale del world model e successivamente può fare asserzioni, cancellazioni e modifiche dinamicamente entro i piani.

L'*observer* è una procedura dichiarativa opzionale che l'interprete JAM esegue tra ogni azione in un piano. Tale procedura è praticamente un piano con solo il corpo e nessun'altra componente, specificato come segue:

```
OBSERVER: {
    [non-subgoaling procedure]
}
```

Il nome di tale procedura è legato alla sua funzione tipica di guardare gli eventi asincroni. Questa procedura che viene eseguita frequentemente non è pesante dal punto di vista computazionale. Per un programmatore di agenti l'observer rappresenta un "aggancio" architetturale che può essere usato per implementare in modo semplice funzionalità estranee alla tipica funzione di ragionamento, basato su goal o piani, di JAM.

### 1.5.2 Semantica

Nella documentazione reperita su JAM non si trova una descrizione formale della semantica.

### 1.5.3 Implementazioni

Esiste un'implementazione Java di JAM che è distribuita liberamente per uso non-commerciale. L'uso a fini commerciali di JAM è subordinato all'accettazione di una licenza con l'Intelligent Reasoning Systems. Le informazioni relative si trovano alla pagina `http://www.marcush.net/IRS/irs_downloads.html`.

Non si sono trovate informazioni relative all'esistenza di applicazioni reali realizzate in JAM.

### 1.5.4 Estensioni

L'architettura JAM non è completa in quanto dovrebbe integrare capacità quali la generazione e l'apprendimento di piani. Per quanto riguarda la

generazione JAM riesce a generare un piano dai suoi primi principi piuttosto che da una libreria di piani pre-programmata come fanno molte architetture BDI (come PRS-CL e UMPRS). La rappresentazione dei piani è stata estesa in modo da includere la rappresentazione dichiarativa di azioni primitive individuali e sono stati implementati algoritmi per l'ordine parziale sui piani basati sulla nuova rappresentazione. Si sono aggiunte a JAM delle abilità per la gestione della conversazione, un linguaggio compatibile FIPA e un protocollo come se fossero estensioni specifiche per le applicazioni ma non si è ancora deciso se queste capacità diventeranno parte integrante di JAM o se saranno fornite come un package supplementare.

## 1.6 Dribble

Dribble è un linguaggio di programmazione per agenti. È stato realizzato da B. van Riemsdijk, W. van der Hoek e J.-J. Ch. Meyer (2003) [73]. Esso costituisce una sintesi tra l'approccio dichiarativo e quello procedurale. In modo più specifico, Dribble si definisce combinando le caratteristiche dichiarative del linguaggio GOAL [31] con quelle procedurali del linguaggio 3APL. 3APL ha *funzionalità di pianificazione*, offre cioé i mezzi per creare e modificare piani durante l'esecuzione dell'agente. Gli agenti GOAL non hanno funzionalità che supportano la pianificazione ma offrono la possibilità di usare *goal dichiarativi* per selezionare le azioni.

L'idea base in Dribble è tale che un agente dovrebbe, tovandosi in un certo istante nel tempo, poter scegliere un piano per raggiungere un goal. Per fare questo l'agente ha a disposizione belief, goal e regole per scegliere piani, crearli e modificarli. La pianificazione così non è intesa come deliberazione e l'agente non ha modo di guardare avanti nel tempo per poter stabilire quale azione compiere. La deliberazione della programmazione dell'agente può comunque essere fatta in un metalinguaggio, usando 3APL o Dribble come linguaggio oggetto.

Le proprietà degli stati mentali degli agenti e degli agenti Dribble stessi sono descritti usando la *logica dinamica*. Questa consente di stabilire una corrispondenza tra le transizioni, definite mediante le azioni della logica, e le transizioni tra gli stati mentali, determinate dal sistema di transizione che definisce la semantica del linguaggio che si vedrà più avanti.

### 1.6.1 Logica dinamica

La logica dinamica (G. Weiss, 1999 [75, 49]) è usata in numerose aree dell'Intelligenza Artificiale Distribuita (Distributed Artificial Intelligence, DAI). In essa, al contrario di quanto avviene nella logica modale tradizionale, gli operatori di necessità e possibilità dipendono dal tipo di azioni disponibili. Si considerano programmi regolari e formule del linguaggio della logica dinamica definiti per mutua induzione.

Si considera un insieme di simboli di azioni atomiche $PA$. I programmi regolari sono definiti come segue:

- tutti i simboli di azioni atomiche in $PA$ sono programmi regolari;
- se $p$ e $q$ sono programmi regolari, allora $p;q$ è un programma regolare che indica *eseguire p e q in sequenza*;
- se $p$ e $q$ sono programmi regolari, allora $(p+q)$ è un programma regolare che indica *eseguire p oppure q*;
- se $p$ è un programma regolare, allora $p*$ è un programma regolare che indica *ripetere zero o più (ma comunque finite) iterazioni di p*;
- se $\varphi$ è una formula del linguaggio della logica dinamica, allora $\varphi?$ è un programma regolare che rappresenta *l'azione di controllare il valore di verità della formula* $\varphi$; ha successo se $\varphi$ risulta vero.

$(p + q)$ è la scelta non-deterministica. Per quanto riguarda $\varphi?$, se $\varphi$ è vero, quest'azione ha successo come una *noop (non azione)* e non influenza lo stato del mondo. Se $\varphi$ è falso, si ha fallimento e il ramo dell'azione di cui il programma fa parte termina senza successo, ottendo uno stato identico a quello raggiungibile se tale ramo non fosse esistito.

Le formule della logica dinamica sono definite come di seguito:

- tutte le formule proposizionali sono formule della logica dinamica;
- se $p$ è un programma regolare e $\varphi$ è una formula della logica dinamica, allora $[p]\varphi$ è una formula della logica dinamica che significa che *ogni qualvolta p termina, deve farlo in uno stato che soddisfi* $\varphi$;
- se $p$ è un programma regolare e $\varphi$ p una formula della logica dinamica, allora $\langle p\rangle\varphi$ è una formula della logica dinamica che significa che *è possibile eseguire p e fermarsi in uno stato che soddisfi* $\varphi$;
- se $\varphi$ e $\psi$ sono formule della logica dinamica, allora $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \Rightarrow \psi$ e $\neg\varphi$ sono formule della logica dinamica.

Dato l'insieme degli stati (o mondi) $S$, si ha la funzione interpretazione

$$h : S \times PROP \rightarrow \{\texttt{true,false}\}$$

che dice se una formula proposizionale appartenente a $PROP$ è vera o falsa in un mondo appartenente a $S$. Si definisce $\sigma \subseteq S \times PA \times S$ come relazione di transizione.

La semantica della logica dinamica è data rispetto a un modello $\langle S, \sigma, h\rangle$ che comprende un insieme $S$ di stati, una relazione di transizione $\sigma$ e una funzione d'interpretazione $h$.

Al fine di fornire la semantica del linguaggio si definisce prima una classe di relazioni di accessibilità ($\beta$ è un simbolo di azione atomica in $PA$; $p$ e $q$ sono programmi regolari; $r$,$s$ e $t$ sono elementi di $S$):

| | | |
|---|---|---|
| $s\ R_\beta\ t$ | sse | $\sigma(s, \beta, t)$ |
| $s\ R_{p;q}\ t$ | sse | esiste $r$ tale che $s\ R_p\ r$ e $r\ R_q\ t$ |
| $s\ R_{p+q}\ t$ | sse | $s\ R_p\ t$ or $s\ R_q\ t$ |
| $s\ R_{p*}\ t$ | sse | esiste $s_0, \ldots, s_n$ tale che $s = s_0$ e $t = s_n$ e per ogni $i, 0 \leq i < n, s_i\ R_p\ s_{i+1}$ |
| $s\ R_{\phi?}\ s$ | sse | $\langle S, \sigma, h\rangle \models_s \phi$ |

Nelle equivalenze seguenti $p$ varia tra programmi regolari e $\phi$ e $\psi$ sono formule della logica dinamica.

Se $\phi$ è una formula proposizionale, la sua semantica è data attraverso la funzione interpretazione $h$:

$$\langle S, \sigma, h\rangle \models_s \phi \quad \text{sse} \quad h(s, \phi) = true$$

La semantica delle formule $\langle p\rangle\varphi$ e $[p]\varphi$ è data da:

| | | |
|---|---|---|
| $\langle S, \sigma, h\rangle \models_s \langle p\rangle\phi$ | sse | esiste $t$ tale che $s\ R_p\ t$ e $\langle S, \sigma, h\rangle \models_t \phi$ |
| $\langle S, \sigma, h\rangle \models_s [p]\phi$ | sse | per ogni $t$, $s\ R_p\ t$ implica $\langle S, \sigma, h\rangle \models_t \phi$ |

La semantica di $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$ e $\neg\phi$ è data in modo standard.

### 1.6.2 Il linguaggio di programmazione Dribble

Ogni agente Dribble contiene due database: *beliefbase* e *goalbase* che sono entrambi insiemi di formule del linguaggio proposizionale $\mathcal{L}$. Il linguaggio $\mathcal{L}$ ha formule $\phi$ e i connettivi $\wedge$ e $\neg$ con il significato solito. L'insieme delle *formule per belief* e *goal* $\mathcal{L}_{BG}$, data la formula $\varphi$, è definito da:

- se $\phi \in \mathcal{L}$, allora $\mathbf{B}\phi, \mathbf{G}\phi \in \mathcal{L}_{BG}$
- se $\varphi_1, \varphi_2 \in \mathcal{L}_{BG}$, allora $\neg\varphi_1, \varphi_1 \wedge \varphi_2 \in \mathcal{L}_{BG}$.

Le formule con solo **B**-operatori sono anche dette "belief formulas" e costituiscono il linguaggio $\mathcal{L}_B$ avente come elemento tipico $\beta$. Analogamente il linguaggio $\mathcal{L}_G$ è quello con formule aventi solo **G**-opertatori.

Un *piano* in un agente Dribble è una sequenza di *elementi base*. Ci sono tre tipi di elementi base: azioni base, piani astratti e costrutti if-then-else. Le azioni base specificano le abilità con cui un agente può raggiungere un certo stato cambiando la propria belief base. Un piano astratto non può essere eseguito direttamente nel senso che esso aggiorna la base dell'agente. Esso serve da meccanismo astratto come le procedure nella programmazione imperativa. Un piano astratto può essere trasformato tramite regole di ragionamento in azioni base.

Formalmente, un piano è formato da un insieme BasicAction e da un insieme AbstractPlan. Si definiscono le ExecutableAction come BasicAction $\cup$ $\{\texttt{if}\beta\texttt{then}\pi_1\texttt{else}\pi_2\texttt{fi}\}$ ($\beta \in \mathcal{L}_B$ e $\pi_1, \pi_2 \in$ Plan). Il simbolo $E$ denota il piano vuoto.

- $E \in$ Plan;
- BasicAction $\cup$ AbstractPlan $\subseteq$ Plan;
- se $\pi_1, \pi_2 \in$ Plan e $\beta \in \mathcal{L}_B$ allora `if` $\beta$ `then` $\pi_1$ `else` $\pi_2$ `fi` $\in$ Plan;
- se $c \in$ ExecutableAction $\cup$ AbstractPlan $\cup \{E\}$ e $\pi \in$ Plan allora $c; \pi \in$ Plan.

Belief, goal e piani costituiscono lo *stato mentale* (o *mental state*) di un agente. I belief descrivono lo stato in cui si trova l'agente, i goal descrivono lo stato che l'agente vuole raggiungere. Le azioni base modificano i belief dell'agente. I goal sono aggiornati solo mediante l'aggiornamento di un belief, in quanto un agente elimina un goal se e solo se crede che tale goal sia stato raggiunto. Questo modo di vedere le cose stabilisce una *significativa relazione tra belief e goal*. Ogni goal dell'agente deve essere consistente così come deve esserlo la belief base.

Secondo la definizione formale, uno stato mentale di un agente Dribble è una tripla $\langle \sigma, \gamma, \pi \rangle$ dove $\sigma \subseteq \mathcal{L}$ sono i belief dell'agente e $\gamma \subseteq \mathcal{L}$ sono i goal dell'agente e $\pi \in$ Plan è il piano dell'agente. $\sigma$ e $\gamma$ sono tali che per ogni $\psi \in \gamma$ si ha:

- i belief dell'agente non implicano $\psi$ ($\sigma \not\models \psi$), e
- $\psi$ è consistente ($\psi \not\models \bot$), e
- $\sigma$ è consistente ($\sigma \not\models \bot$).

L'insieme dei possibili stati mentali è $\Sigma$ avente come elemento tipico $s$.

Il piano di un agente può essere cambiato mediante l'applicazione di regole o esecuzione di azioni eseguibili (azioni base o costrutti if-then-else). Ci sono due tipi di regole: *goal rules* e *Practical Reasoning (PR) rules*.

Formalmente, una *goal rule* $g$ è una coppia $\varphi \rightarrow \pi$ tale che $\varphi \in \mathcal{L}_{BG}$ e $\pi \in$ Plan. Una *PRrule* $\rho$ è una tripla $\pi_h|\beta \rightarrow \pi_b$ tale che $\beta \in \mathcal{L}_B$ e $\pi_h, \pi_b \in$ Plan.

Una goal rule indica che il piano $\pi$ può essere adottato se si verifica la condizione $\varphi$. Le goal rules sono usate per *scegliere* i piani per raggiungere un goal a partire da un certo stato. Una PR rule è letta come segue: se l'agente ha adottato il piano $\pi_h$ e se esso crede la formula $\beta$, allora il piano $\pi_h$ può essere sostituito con il piano $\pi_b$. Le PR rules possono essere usate per *creare* piani (spesso da piani astratti), per *modificare* piani e per *modellare comportamenti reattivi* (usando PR rules con testa vuota).

Un *agente* Dribble è una quadrupla $\langle \sigma_0, \gamma_0, \Gamma, \Delta \rangle$ dove $\langle \sigma_0, \gamma_0, E \rangle$ è lo stato mentale iniziale, $\Gamma$ è un insieme di goal rules e $\Delta$ è un insieme di PR rules.

Nello stato mentale iniziale il piano dell'agente è vuoto. Questo segue dall'idea per cui un agente deve cominciare ad agire perché ha certi goal. Un agente Dribble adotta piani per soddisfare questi goal secondo le goal rules. Durante l'esecuzione l'agente può facilmente adattare il suo piano seguendo le PR rules.

### 1.6.3 Semantica operazionale

La semantica delle formule per i belief e i goal è definita in modo tale che presi lo stato mentale $\langle \sigma, \gamma, \pi \rangle$, $\phi, \psi \in \mathcal{L}$ e $\varphi, \varphi_1, \varphi_2 \in \mathcal{L}_{BG}$ si ha:

- $\langle \sigma, \gamma, \pi \rangle \models \mathbf{B}\phi$ se e solo se $\sigma \models \phi$;
- $\langle \sigma, \gamma, \pi \rangle \models \mathbf{G}\psi$ se e solo se per alcuni $\psi' \in \gamma : \psi' \models \psi$ e $\sigma \not\models \psi$;
- $\langle \sigma, \gamma, \pi \rangle \models \neg\varphi$ se e solo se $\langle \sigma, \gamma, \pi \rangle \not\models \varphi$
- $\langle \sigma, \gamma, \pi \rangle \models \varphi_1 \wedge \varphi_2$ se e solo se $\langle \sigma, \gamma, \pi \rangle \models \varphi_1$ e $\langle \sigma, \gamma, \pi \rangle \models \varphi_2$

Una formula $\mathbf{B}\phi$ è vera in uno stato mentale se e solo se la belif base $\sigma$ modella $\phi$, in modo che l'agente creda tutte le conseguenze logiche dei suoi belief. La semantica di $\mathbf{G}\phi$ è definita in termini di goal separati, in contrasto con la sua definizione data in termini dell'intera goal base. Tutte le conseguenze logiche di un goal particolare sono goal ma solo se non sono creduti. Due goal distinti potrebbero essere inconsistenti e quindi potrebbero non essere combinati in un goal unico.

La semantica del linguaggio di programmazione è definita mediante *sistema di transizione*. Un sistema di transizione è formato da un insieme di regole di derivazione per ricavare transizioni per un agente. Una *transizione* è una trasformazione di uno stato mentale in un altro e corrisponde ad un singolo passo di computazione. Si descrivono, nel seguito, alcune regole di transizione (tralasciando quella per l'esecuzione del costrutto if-then-else).

Una *goal rule* è applicabile in uno stato mentale se l'antecedente della regola è vero in tale stato mentale. Può essere applicata solo se il piano dell'agente è vuoto. Questo segue dall'idea secondo cui un agente può selezionare un piano nuovo solo se ha finito di eseguire il suo vecchio piano. Nello stato mentale risultante, il piano diventa uguale alla conseguenza della goal rule. Quindi, presa la goal rule $g : \varphi \rightarrow \pi \in \Gamma$ la sua applicazione è formalizzata da:

$$\frac{\langle \sigma, \gamma, E \rangle \models \varphi}{\langle \sigma, \gamma, E \rangle \rightarrow_{\texttt{applyRule}(g)} \langle \sigma, \gamma, \pi \rangle}$$

Una *PR rule* è applicabile in uno stato mentale se il piano dell'agente è uguale alla testa della regola e la guardia della regola è vera in tale stato

mentale. Il risultato dell'applicazione della regola è che il piano nel corpo della regola viene adottato, rimpiazzando il piano che era uguale alla testa della regola. Infatti, data la PR rule $\rho : \pi_h|\beta \rightarrow \pi_b \in \Delta$ la sua applicazione è così formalizzata:

$$\frac{\langle \sigma, \gamma, \pi_h \rangle \models \beta}{\langle \sigma, \gamma, \pi_h \rangle \rightarrow_{\texttt{applyRule}(\rho)} \langle \sigma, \gamma, \pi_b \rangle}$$

L'*esecuzione di azioni base* di un agente aggiornano la belief base. Questi aggiornamenti di belief sono rappresentati con la funzione parziale $\mathcal{T}$, per cui $\mathcal{T}(a, \sigma)$ restituisce il risultato dell'aggiornamento della belief base $\sigma$ mediante il compimento dell'azione $a$. Il fatto che $\mathcal{T}$ sia una funzione parziale rappresenta la situazione in cui un'azione può non essere eseguibile in alcuni stati dei belief. I goal raggiunti mediante l'esecuzione di un'azione sono rimossi dalla goal base. Dopo l'esecuzione un'azione è tolta dal piano. L'esecuzione di un'azione base, dato $\gamma' = \gamma \backslash \{\psi \in \gamma | \sigma' \models \psi\}$, è formalizzata come segue

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle \sigma, \gamma, a \rangle \rightarrow_{\texttt{execute}(a)} \langle \sigma', \gamma', E \rangle}$$

Un piano può essere formato da un elemento base o da una sequenza di elementi base e le regole viste sopra descrivono come procedere nell'esecuzione di un unico elemento (un'azione base) o di una sequenza fissata di elementi (la testa di una PR rule). Se si deve eseguire una sequenza arbitraria di elementi base (un piano) si deve usare una regola per l'*esecuzione di composizioni sequenziale*. La sua formalizzazione è la seguente, per cui dato $x \in \{\texttt{applyRule}(\rho), \texttt{execute}(a), \texttt{execute}(\texttt{if}\beta\texttt{then}\pi_1\texttt{else}\pi_2\texttt{fi})\}$, si ha

$$\frac{\langle \sigma, \gamma, \pi_1 \rangle \rightarrow_x \langle \sigma', \gamma', \pi_1' \rangle}{\langle \sigma, \gamma, \pi_1 \circ \pi_2 \rangle \rightarrow_x \langle \sigma', \gamma', \pi_1' \circ \pi_2 \rangle}$$

Si vede che la prima parte del piano è eseguita per prima, ogni cambiamento alla belief base viene registrato e l'agente continua ad eseguire il resto del piano. L'operatore $\circ$ usato, denota che un certo piano è una sequenza di elementi base di cui $\pi_1$ (o $\pi_1'$) è la prima parte, cioè il prefisso di tale piano e, come tale, deve essere eseguito prima, e $\pi_2$ è la parte che resta da eseguire. Questa regola può essere usata solo se la transizione che compare nella premessa non è derivata con la regola di transizione per l'applicazione di una goal rule, cioè se $x \neq \texttt{applyRule}(g)$, infatti una goal rule può essere applicata solo se il piano dell'agente è vuoto. Questa restrizione è necessaria per prevenire la derivazione di transizioni del tipo

$$\langle \sigma, \gamma, E \circ \pi' \rangle \rightarrow_{\texttt{applyRule}(g)} \langle \sigma, \gamma, \pi \circ \pi' \rangle$$

dalla transizione

$$\langle \sigma, \gamma, E \rangle \rightarrow_{\texttt{applyRule}(g)} \langle \sigma, \gamma, \pi \rangle.$$

La semantica di un agente Dribble è derivata direttamente dalla relazione di transizione $\rightarrow$. L'intenzione di un agente Dribble è data da un insieme di cosiddetti *computation run*.
Un computation run $\texttt{CR}(s_0)$ per un agente Dribble con goal rules $\Gamma$ e PR rules $\Delta$ è una sequenza finita o infinita $s_0, ..., s_n$ o $s_0, ...$ dove gli $s_i \in \Sigma$ sono stati mentali, e $\forall_{i>0} : s_{i-1} \rightarrow_x s_i$ è una transizione nel sistema di transizione per l'agente Dribble. Il significato di un agente $\langle \sigma_0, \gamma_0, \Gamma, \Delta \rangle$ è l'insieme dei computation run $\texttt{CR}(\langle \sigma_0, \gamma_0, E \rangle)$, in cui il primo stato dei computation run è lo stato mentale iniziale dell'agente.

Nel caso in cui più di una regola sia applicabile in un certo stato mentale, le regole di transizione per l'applicazione di goal o PR rules non dicono quale regola applicare, né dicono se applicare una regola o eseguire un'azione se entrambe le cose sono possibili in uno stato. Ogni implementazione del linguaggio Dribble deve occuparsi di come ridurre questo non-determinismo nella semantica del linguaggio. Per fare questo servono delle strutture di controllo.

Le proprietà degli agenti Dribble possono essere provate costruendo una logica dinamica ($\mathcal{L}_D$) sul linguaggio Dribble e sulla sua semantica. Si usano forme dello stato mentale in $\mathcal{L}_M$ per descrivere le proprietà dello stato mentale di un agente Dribble. Queste formule sono un'estensione di quelle per belief e goal, viste in predecenza, estese con una nuova clausola che indica quando un piano è una componente di uno stato mentale.

Il passaggio da uno stato mentale all'altro dipende dall'applicazione di una regola o dall'esecuzione di un'azione. La logica é usata per ragionare su questi passaggi, ovvero transizioni, tra gli stati mentali che determinano l'esecuzione dell'agente. Le transizioni sono riferite usando *meta-azioni*. Nella logica dinamica per Dribble una sequenza di meta-azioni è usata come il programma $p$ nella formula $\langle p \rangle \phi$.

### 1.6.4 Implementazioni

In base alle ricerche effettuate attualmente non sembrano esserci implementazioni di Dribble disponibili.

### 1.6.5 Estensioni

Un'estensione oggetto di ricerca futura è volta a superare la limitazione imposta dal fatto che Dribble è un linguaggio proposizionale e prevede di introdurre nel linguaggio delle caratteristiche tipiche dei linguaggi del prim'ordine.

Un'alta estensione riguarda l'aggiunta di regole per il ragionamento sui goal poiché quelle presenti usano i goal solo per selezionare i piani. Una terza

possibilità riguarda la strategia di commitment. Nella versione corrente l'agente elimina un goal solo se crede che sia stato raggiunto.

Una variante interessante potrebbe invece vedere se è possibile che l'agente elimini il suo goal anche se crede che esso sia irraggiungibile.

Infine, si potrebbe memorizzare il goal per cui è stato selezionato un piano e l'agente potrebbe essere programmato per cancellare questo piano nel momento in cui si soddisfa il goal per cui era stato selezionato.

## 1.7 ConGolog

ConGolog è un linguaggio di programmazione concorrente basato sul *situation calculus* [49] che comprende funzionalità che consentono di assegnare delle priorità nelle esecuzioni concorrenti, di interrompere l'esecuzione quando certe condizioni diventano vere, e di trattare con "exogenous actions", cioè con azioni che nascono all'esterno. ConGolog è stato sviluppato da G. DeGiacomo *et al.* [18] e secondo loro rappresenta un'alternativa promettente alla tradizionale pianificazione, poiché permette l'esecuzione ad alto livello del programma. ConGolog è un'estenzione del linguaggio di programmazione Golog [47]. Qui di seguito vengono presentati il situation calculus e Golog dalla cui estensione nasce ConGolog.

### 1.7.1 Situation Calcolus

Il SITUATION CALCULUS (J. McCarthy, 1963 [52]) è ben conosciuto nella ricerca in intelligenza artificiale. La descrizione seguente si basa su uno studio di F. Pirri e R. Reiter [59]. $\mathcal{L}_{sit-calc}$ è un linguaggio del secondo ordine con uguaglianza. Ha tre tipi disgiunti: *action* per le azioni, *situation* per le situazioni e un tipo *object* per ogni altra cosa a seconda del dominio dell'applicazione. Oltre all'alfabeto standard dei simboli logici ($\wedge$, $\neg$ e $\exists$, usati con il loro solito significato), $\mathcal{L}_{sit-calc}$ ha l'alfabeto seguente:

- Simboli di variabile individuale infiniti e numerabili per ogni tipo e variabili di predicati di tutte le arietà infinite e numerabili.
- Due simboli di funzione per il tipo *situation*:
  1. Un simbolo di costante $So$, che denota la situazione iniziale.
  2. Un simbolo di funzione binario $do : action \times situation \rightarrow situation$. $do(a, s)$ denota la situazione successiva risultante dal portare a termine l'azione $a$ nella situazione $s$.
- Un simbolo di predicato binario $\sqsubset$: $situation \times situation$, che definisce una relazione d'ordine sulle situazioni. L'interpretazione data per le situazioni è come la storia delle azioni in cui $s \sqsubset s'$ significa che $s'$ può essere raggiunto da $s$ mediante un'applicazione finita di azioni.

- Un simbolo di predicato binario $Poss : action \times situation$. L'interpretazione data di $Poss(a, s)$ è che è possibile realizzare l'azione $a$ nella situazione $s$.
- Simboli di predicato infiniti e numerabili usati per denotare relazioni che non dipendono dalla situazione e simboli di funzione infiniti e numerabili usati per denotare funzioni che non dipendono dalla situazione.
- Un numero finito o infinito e numerabile di simboli di funzione detti *action functions* e usati per denotare azioni.
- Un numero finito o infinito e numerabile di *fluenti relazionali (relational fluents)*, ovvero simboli di predicati usati per denotare relazioni che dipendono dalla situazione.
- Un numero finito o infinito e numerabile di simboli di funzione detti *fluenti funzionali (functional fluents)* e usati per denotare funzioni che dipendono dalla situazione.

Nell'assiomatizzazione proposta da Levesque gli assiomi sono divisi in assiomi dipendenti dal dominio e assiomi fondamentali indipendenti dal dominio per le situazioni. Oltre agli assiomi si introducono anche le teorie di base delle azioni e una metateoria per il situation calculus che permette di determinare quando una teoria di base delle azioni è soddisfacibile e quando essa implica un particolare tipo di sentenza, detta *regressable sentence*. Qui vengono trattati solo gli assiomi fondamentali indipendenti dal dominio per le situazioni, tralasciando altre cose quali gli assiomi dipendenti dal dominio e la metateoria per il situation calculus, poiché entrambi richiedono una forte preparazione tecnica e non sono indispensabili per la comprensione di quanto segue.

### Foundational axioms for situations

Ci sono quattro assiomi fondamentali per il situation calculus, basato su F. Pirri e R. Reiter [59], qui presentati più semplicemente. Essi catturano l'intuizione che le situazioni sono sequenze finite di azioni in cui vale il principio d'induzione del second'ordine, e che c'è una relazione di "sottosequenza" tra loro. Negli assiomi seguenti, $P$ è un simbolo di predicato.

$$do(a_1, s_1) = do(a_2, s_2) \Rightarrow a_1 = a_2 \wedge s_1 = s_2 \tag{1.1}$$

$$\forall P.P(S_0) \wedge \forall a, s.[P(s) \Rightarrow P(do(a, s))] \Rightarrow \forall s.P(s) \tag{1.2}$$

L'assioma 1.1 è un assioma che designa unicità per le situazioni: due situazioni sono la stessa se e solo se esse sono la stessa sequenza di azioni. L'assioma 1.2 descrive l'induzione del second'ordine sulle situazioni. Gli assiomi

terzo e quarto sono:

$$\neg\ (s \sqsubset S_0) \tag{1.3}$$

$$(s \sqsubset do(a, s')) \equiv (s \sqsubseteq s') \tag{1.4}$$

Qui $s \sqsubseteq s'$ è un'abbreviazione per $(s \sqsubset s') \vee (s = s')$. La relazione $\sqsubset$ fornisce una relazione d'ordinamento sulle situazioni. Intuitivamente, $s \sqsubset s'$ significa che la situazione $s'$ può essere ottenuta dalla situazione $s$ aggiungendo una o più azioni alla fine di $s$. I quattro assiomi sopra sono *indipendenti dal dominio*. Essi forniscono le proprietà di base delle situazioni in ogni assiomatizzazione di dominio specifico per fluenti particolari e azioni.

### Golog

Golog è un linguaggio di programmazione logica le cui azioni primitive sono tratte dal dominio della teoria sottostante. I programmi Golog sono definiti induttivamente da:

- Data un'azione $a$ del situation calculus con tutti gli argomenti della situazione nei suoi parametri rimpiazzati dalla costante speciale *now*, $a$ è un programma Golog (azione primitiva).
- Data una formula $\varphi$ del situation calculus con tutti gli argomenti della situazione nei suoi parametri rimpiazzati dalla costante speciale *now*, $\varphi?$ è un programma Golog (in attesa su una condizione).
- Dati $\delta$, $\delta_1$, $\delta_2$, $\delta_n$ programmi Golog,
  - $(\delta 1; \delta 2)$ è un programma Golog (sequenza);
  - $(\delta 1 | \delta 2)$ è un programma Golog (scelta non-deterministica tra azioni);
  - $\pi v \cdot \delta$ è un programma Golog (scelta non-deterministica di argomenti);
  - $\delta^*$ è un programma Golog (iterazione non-deterministica);
  - $\{$**proc** $P_1(\overrightarrow{v_1})\delta_1$ **end**; $\ldots$ **proc** $P_n(\overrightarrow{v_n})\delta_n$ **end**; $\delta$ $\}$ è un programma Golog (procedure: $P_i$ sono nomi di procedure e $v_i$ sono i parametri).

*Esecuzione del programma.* Dato il dominio di una teoria $\mathcal{D}$ e un programma $\delta$ il compito dell'esecuzione è trovare una sequenza $\overrightarrow{a}$ di azioni tale che:

$$\mathcal{D} \models Do(\delta, S_0, do(\overrightarrow{a}, S_0))$$

dove

$$Do(\delta, s, s')$$

significa che il programma $\delta$ quando eseguito a partire dalla situazione $s$ ha

$s'$ come situazione terminale legale, e

$$do(\vec{a}, s) = do([a_1, \ldots, a_n], s)$$

è un'abbreviazione per

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

Poiché i programmi Golog possono essere non-deterministici, ci possono essere molteplici situazioni terminali per gli stessi programma e situazione di partenza. $Do(\delta, s, s')$ è formalmente definito per mezzo della seguente definizione induttiva, dove $s$ è una variabile di tipo *situation*:

1. Azioni primitive ($a[s]$ denota l'azione ottenuta sostituendo la variabile $s$ a tutte le occorrenze di *now* nei fluenti funzionali che appaiono in $a$):

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

2. Azioni wait/test ($\phi[s]$ denota la formula ottenuta sostituendo la variabile per $s$ per tutte le occorrenze di *now* nei fluenti funzionali e di predicato che appaiono in $\phi$):

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

3. Sequenze:

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

4. Scelta non-deterministica:

$$Do(\delta_1 \mid \delta_2, s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$$

5. Scelta di argomento non-deterministica ($\pi x \cdot \delta(x)$ è eseguito non-deterministicamente scegliendo una singola $x$, e per tale $x$, portando a termine il programma $\delta(x)$):

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

6. Iterazione non-deterministica:

$$Do(\delta^*, s, s') \stackrel{def}{=} \forall P.\{\forall s_1.P(s_1, s_1) \wedge \forall s_1, s_2, s_3.[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \Rightarrow P(s_1, s_3)]\} \Rightarrow P(s, s')$$

$P$ è un simbolo di predicato binario. Dire "$(x, x')$ è nell'insieme (definito da $P$)" è equivalente a dire "$P(x, x')$ è vero". Fare l'azione $\delta$ zero o più volte porta dalla situazione $s$ alla situazione $s'$ se e solo se $(s, s')$ è in ogni insieme (e quindi, il più piccolo insieme) tale che:

(a) $(s_1, s_1)$ è nell'insieme di tutte le situazioni $s_1$.

(b) Ogni qualvolta $(s_1, s_2)$ è nell'insieme, e facendo $\delta$ nella situazione $s_2$ che porta alla situazione $s_3$, allora $(s_1, s_3)$ è nell'insieme.

Quanto sopra è la definizione standard del second'ordine dell'insieme ottenuto mediante iterazione non-deterministica.

L'espansione delle procedure non è trattata ma si può vedere il lavoro di H. J. Levesque *et al.* [47] per i dettagli.

### 1.7.2 I programmi ConGolog

ConGolog è una versione estesa di Golog che incorpora la concorrenza, gestendo

- processi concorrenti possibilmente con differenti priorità;
- interruzioni ad alto livello e
- exogenous actions, azioni che hanno cioè origine esterna, arbitrarie.

I programmi ConGolog sono definiti dalle seguenti regole induttive:

- Tutti i programmi Golog sono programmi ConGolog.
- Dati una formula $\phi$ del situation calculus con tutti gli argomenti della situazione rimpiazzati nei parametri dalla costante speciale *now*, e i programmi ConGolog $\delta$, $\delta_1$, $\delta_2$,
  - **if** $\phi$ **then** $\delta_1$ **else** $\delta_2$ è un programma ConGolog (scelta condizionale sincronizzata);
  - **while** $\phi$? **do** $\delta$ è un programma ConGolog (ciclo sincronizzato);
  - $(\delta_1 \parallel \delta_2)$ è un programma ConGolog (esecuzione concorrente);
  - $(\delta_1 \rangle\rangle \delta_2)$ è un programma ConGolog (concorrenza con diversa priorità);
  - $\delta^{\parallel}$ è un programma ConGolog (iterazione concorrente);
  - $\langle \phi \rightarrow \delta \rangle$ è un programma ConGolog (interruzione).

I costrutti **if** $\phi$ **then** $\delta_1$ **else** $\delta_2$ e **while** $\phi$? **do** $\delta$ sono le versioni sincronizzate degli usuali cicli if-then-else e while-do. Essi sono sincronizzati nel senso che il test della condizione $\phi$ non coinvolge una transizione "per sé" ma la valutazione della condizione e la prima azione del ramo scelto saranno eseguite come un'azione atomica. Il costrutto $(\delta_1 \parallel \delta_2)$ denota l'esecuzione concorrente delle azioni $\delta_1$ e $\delta_2$. $(\delta_1 \rangle\rangle \delta_2)$ denota l'esecuzione concorrente delle azioni $\delta_1$ e $\delta_2$ in cui $\delta_1$ ha priorità più alta di $\delta_2$, restringendo i modi possibili in cui i due processi si alternano: $\delta_2$ è eseguito solo quando $\delta_1$ è completato o bloccato. Il costrutto $\delta^{\parallel}$ è come l'iterazione non-deterministica, ma le istanze di $\delta$ sono eseguite concorrentemente invece che in sequenza.

Infine, $\langle \phi \rightarrow \delta \rangle$ è un'interruzione. Ha due parti: una trigger condition $\phi$ e un corpo $\delta$. L'idea è che il corpo $\delta$ sarà eseguito un certo numero di volte. Se $\phi$ non diventa mai vera, $\delta$ non sarà mai eseguito. Se l'interruzione prende il controllo da processi a priorità più alta quando $\phi$ è vera, allora $\delta$ sarà eseguito. Una volta che la sua esecuzione è completata, l'interruzione è pronta per essere "attivata" di nuovo. Questo significa che un'interruzione ad alta priorità può prendere il controllo completo dell'esecuzione.

### 1.7.3 Semantica

La semantica di Golog e ConGolog è nello stile delle semantiche di transizione. Sono definiti due predicati che dicono quando un programma $\delta$ può terminare legalmente in una certa situazione $s$ ($Final(\delta, s)$) e quando un programma $\delta$ nella situazione $s$ può legalmente eseguire un passo, finendo in una situazione $s'$ con il rimanente programma $\delta'$ ($Trans(\delta, s, \delta', s')$). $Final$ e $Trans$ sono caratterizzati da un insieme di assiomi di equivalenza, ognuno dipendente dalla struttura del primo argomento. Per dare l'idea di come appaiono questi assiomi, vengono mostrati quelli per il programma vuoto $nil$, l'azione atomica $a$, il testing $\phi?$, il branch non-deterministico $(\delta_1|\delta_2)$ e l'esecuzione concorrente $(\delta_1 \parallel \delta_2)$. L'insieme completo degli assiomi per $Final$ e $Trans$ si può trovare nel lavoro di G. DeGiacomo *et al.* [18].

$$
\begin{array}{lcl}
Trans(nil, s, \delta', s') & \equiv & \text{false} \\
Trans(a, s, \delta', s') & \equiv & Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s) \\
Trans(\phi?, s, \delta', s') & \equiv & \phi[s] \wedge \delta' = nil \wedge s' = s \\
Trans(\delta_1 \mid \delta_2, s, \delta', s') & \equiv & Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\
Trans(\delta_1 \parallel \delta_2, s, \delta', s') & \equiv & \\
\multicolumn{3}{l}{\quad \exists\gamma.\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee} \\
\multicolumn{3}{l}{\quad \exists\gamma.\delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s')}
\end{array}
$$

Il significato di questi assiomi è che: $(nil, s)$ non evolve in alcuna configurazione; $(a, s)$ evolve in $(nil, do(a[s], s))$ prevedendo che $a[s]$ è possibile in $s$; $(\phi?, s)$ evolve in $(nil, s)$ prevedendo che $\phi[s]$ sia valido; $(\delta_1|\delta_2, s)$ può evolvere in $(\delta', s')$ prevedendo che sia $(\delta_1, s)$ sia $(\delta_2, s)$ possano evolvere così; e infine, $(\delta_1 \parallel \delta_2, s)$ può evolvere se $(\delta_1, s)$ può evolvere e $\delta_2$ rimane invariato o $(\delta_2, s)$ può evolvere e $\delta_1$ resta invariato.

$$
\begin{array}{lcl}
Final(nil, s) & \equiv & \text{true} \\
Final(a, s) & \equiv & \text{false} \\
Final(\phi?, s) & \equiv & \text{false} \\
Final(\delta_1 \mid \delta_2, s) & \equiv & Final(\delta_1, s) \vee Final(\delta_2, s) \\
Final(\delta_1 \parallel \delta_2, s) & \equiv & Final(\delta_1, s) \wedge Final(\delta_2, s)
\end{array}
$$

Questi assiomi dicono che $(nil, s)$ è una configurazione finale mentre né $(a, s)$ né $(\phi?, s)$ lo sono. $(\delta_1|\delta_2, s)$ è finale se lo è $(\delta_1, s)$ o lo è $(\delta_2, s)$, mentre $(\delta_1 \parallel \delta_2, s)$ è finale se entrambe $(\delta_1, s)$ e $(\delta_2, s)$ lo sono. Le possibili configurazioni che possono essere raggiunte da un programma $\delta$ nella situazione $s$ sono quelle ottenute seguendo ripetutamente la relazione di transizione denotata da $Trans$ che parte da $(\delta, s)$. La chisura riflessiva e transitiva di $Trans$ è denotata da $Trans^*$. Per mezzo di $Final$ e $Trans^*$ è possibile dare una nuova definizione di $Do$ come

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

### 1.7.4 Implementazioni

Una semplice implementazione di ConGolog è stata sviluppata in Prolog. La definizione dell'interprete segue direttamente dalle definizioni di $Final$, $Trans$ e $Do$ date sopra. L'interprete richiede che gli assiomi precondizione del programma, gli assiomi per lo stato successivo e gli assiomi relativi alla situazione iniziale siano esprimibili come clausole Prolog. In particolare, la solita *"closed world assumption"* è fatta nella situazione iniziale.

La Sezione 8 del lavoro di G. DeGiacomo *et al.* [18] descrive l'interprete ConGolog in dettaglio e prova la sua correttezza sotto le debite assunzioni. L'interprete è incluso in un toolkit più sofisticato che fornisce funzionalità per il debugging di programmi ConGolog e applicazioni per la modellazione dei processi per mezzo di un'interfaccia grafica. Per scaricare l'interprete di ConGolog e le sue estensioni si può visitare la Cognitive Robotics Group Home Page [13].

### 1.7.5 Estensioni

Negli ultimi anni sono state sviluppate diverse varianti di ConGolog:

- **Legolog** (*LEGO MINDSTORM in (Con)Golog* [46]) usa un controllore della famiglia Golog dei pianificatori per controllare un robot MINDSTORM. Legolog è capace di trattare con azioni primitive, azioni nate all'esterno e sensing; il controllore Golog è rimpiazzato con un pianificatore alternato. Per dettagli visitare la Legolog Home Page [45].

- **IndiGolog** (*Incremental Deterministic (Con)Golog* [19]) è un linguaggio di programmazione ad alto livello dove i programmi sono eseguiti incrementalmente per permettere azioni in alternanza, pianificazione, percezione, e eventi esterni. IndiGolog fornisce un pratico ambiente per robot reali che possono reagire all'ambiente e dedurre continuamente nuove informazioni da esso. Per influire sulla pianificazione, IndiGolog fornisce un meccanismo locale "lookahead" con un

nuovo costrutto del linguaggio detto *the search operator*.

- **CASL** (*Cognitive Agent Specificaton Language* [67]) è un ambiente per specificare MASs complessi che fornisce un ambiente di verifica basato sul sistema di verifica PVS [58].

- Una classe di programmi Golog basato sulla conoscenza è esteso con azioni di percezione in un lavoro di R. Reiter [64].

Molte delle pubblicazioni su Golog, ConGolog e loro estensioni possono essere trovate nella Cognitive Robotics Group Home Page [13].

## 1.8 AGENT-0

Il linguaggio AGENT-0 è stato sviluppato da Y. Shoham nel 1993 [49]. Y. Shoham ha anche proposto il paradigma di programmazione *Agent-Oriented Programming* (AOP) [68]. AGENT-0 è spesso citato come il primo linguaggio di programmazione per agenti fondato su AOP ed è basato sulle modalità mentali, che trovano una formalizzazione rigorosa nella *logica modale*. Di seguito vengono introdotti la modal logic e i concetti base dell'Agent-Oriented Programming.

### 1.8.1 Modal logic

Questa introduzione è basata su un lavoro di M. Fisher e R. Owens [25]. La logica modale è un'estensione della logica classica con (generalmente) un nuovo connettivo $\Box$ e la sua derivabile controparte $\Diamond$, conosciuti come *necessità* e *possibilità*, rispettivamente. Se una formula $\Box p$ è vera, ciò significa che $p$ è necessariamente vera, cioé vera in ogni possibile scenario, e $\Diamond p$ significa che $p$ è possibilmente vera, cioé in almeno un possibile scenario. È possibile definire $\Diamond$ in termini di $\Box$:

$$\Diamond p \Leftrightarrow \neg\Box\neg p$$

così che $p$ è possibile esattamente quando la sua negazione non è necessariamente vera. Per dare significato a $\Box$ e $\Diamond$, i modelli per la logica modale sono solitamente basati sui *mondi possibili*, che sono essenzialmente una collezione di modelli per la logica classica collegati. I mondi possibili sono collegati da una relazione che determina quali mondi sono accessibili da ciascun mondo dato. È questa relazione di accessibilità che determina la natura della logica modale. A ogni mondo è data un'unica etichetta, presa da un insieme $S$, che è solitamente infinito e numerabile. La relazione di accessibilità $R$ è una relazione binaria su $S$. La coppia con $S$ e $R$ definisce un "frame" o struttura

che sostiene il modello della logica modale. Per completare il modello noi aggiungiamo un interpretazione

$$h : S \times PROP \rightarrow \{\text{true}, \text{false}\}$$

della formula proposizionale $\in PROP$ in ogni stato.

Dati $s \in S$ e $a \in PROP$,

$$\langle S, R, h\rangle \models_s a \qquad \text{sse} \qquad h(s, a) = true$$

Questo si legge come: $a$ è vero nel mondo $s$ nel modello $\langle S, R, h\rangle$ se e solo se $h$ associa ad $a$ "true" nel mondo $s$. In generale quando una formula $\varphi$ è vera in un mondo $s$ in un modello $\mathcal{M}$, ciò si denota con

$$\mathcal{M} \models_s \varphi$$

e se è vero in ogni mondo nell'insieme $S$, ciò si dice che essere vero nel modello, e si denota con

$$\mathcal{M} \models \varphi$$

I connettivi booleani sono dati con il solito significato:

$$\begin{array}{lll} \langle S, R, h\rangle \models_s \varphi \vee \psi & \text{sse} & \langle S, R, h\rangle \models_s \varphi \ \text{ o } \ \langle S, R, h\rangle \models_s \psi \\ \langle S, R, h\rangle \models_s \varphi \Rightarrow \psi & \text{sse} & \langle S, R, h\rangle \models_s \varphi \ \text{ implica } \ \langle S, R, h\rangle \models_s \psi \end{array}$$

Il frame entra a far parte della definizione semantica solo quando è usata la modalità $\Box$, così la formula $\Box\varphi$ è vera in un mondo $s$ esattamente quando ogni mondo $t$ in $S$ che è accessibile da $s$ (ad esempio si supponga che $sRt$) ha $\varphi$ vero. Più formalmente,

$$\langle S, R, h\rangle \models_s \Box\varphi \qquad \text{sse} \quad \text{per ogni} \quad t \in S, \quad s \ R \ t \ \text{ implica } \ \langle S, R, h\rangle \models_t \varphi$$

I modelli $\langle S, R, h\rangle$ e le semantiche che noi introduciamo per i connettivi sono già conosciuti come modelli di Kripke (o structures) e semantiche di Kripke [40, 41, 42], dal nome dell'autore che principalmente contibuì a sviluppare una teoria semantica soddisfacente della logica modale.

### Agent-Oriented Programming (AOP)

Per Shoham, un sistema AOP completo includerà tre componenti primarie:

1. Un linguaggio formale ristretto con sintassi e semantica chiare per descrivere stati mentali; lo stato mentale sarà definito unicamente da molteplici modalità, come credenze (*belief*) e obbligazioni (*commitment*).

2. Un linguaggio di programmazione implementato in cui definire e programmare agenti con comandi primitivi come *REQUEST* e *INFORM*.

3. Un processo di "agentificazione" per trattare i dispositivi hardware e le applicazioni software esistenti come agenti.

L'attenzione del lavoro di Shoham si focalizza sulla seconda componente.

Le categorie mentali su cui AOP è basato sono *belief* e *obligation* (o *commitment*). Si ha inoltre una terza categoria, la *capability*, che non è un costrutto mentale. *Decision* (o *choice*) è trattata come un obligazione verso se stesso. Poiché il tempo (*time*) è basilare nelle categorie mentali, è necessario specificarlo e lo si fa mediante un semplice linguaggio di logica temporale discreta (point-based). Una tipica espressione in questo linguaggio sarà

$$holding(robot, cup)^t$$

che significa che il robot sta tenendo la tazza all'istante di tempo $t$.

Le azioni non sono distinte dai fatti: la rappresentazione dell'occorenza di un'azione coincide con il fatto corrispondente ad essa avente valore vero.

I belief sono rappresentati per mezzo dell'operatore modale $B$. La forma generale di un belief è

$$B_a^t \phi$$

che significa che l'agente $a$ crede $\varphi$ all'istante $t$. $\varphi$ può essere una frase come $holding(robot, cup)^t$ o un belief: belief annidati come $B_a^3 B_b^{10} like(a, b)^7$, che significa che all'istante 3 l'agente $a$ crede che all'istante 10 l'agente $b$ creda che all'istante 7 ad $a$ piaccia $b$, sono legali nel linguaggio AOP.

Il fatto che al tempo $t$ un agente $a$ obblighi se stesso nei confronti dell'agente $b$ circa $\varphi$ è rappresentato dalla frase

$$OBL_{a,b}^t \varphi$$

Una *decision* è un'obbligazione verso se stessi, quale

$$DEC_a^t f \stackrel{def}{=} OBL_{a,a}^t f$$

Il fatto che al tempo $t$ l'agente $a$ sia capace di $\varphi$ è rappresentato da

$$CAN_a^t \varphi$$

Infine, c'è un'"immediata" versione di CAN:

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

in cui $time(B_a^t \psi) = t$ and $time(pred(arg_1, ..., arg_n)^t) = t$).

Per poter recuperare il senso comune della controparte delle modalità introdotte, vanno prima fatte varie assunzioni:

- *Consistenza interna*: si assume che sia belief sia obligation siano internamente consistenti.

- *Buona fede*: gli agenti si impegnano solo per ciò per cui si credono capaci, e solo se ne hanno realmente l'intenzione.
- *Introspezione*: gli agenti sono consapevoli delle loro obbligazioni.
- *Persistenza dello stato mentale*: gli agenti hanno memoria perfetta dei loro belief, e fiducia in essi, e abbandonano un belief solo se essi imparano un fatto per esso contradditorio. Anche le obbligazioni dovrebbero persistere, e le capacità dovrebbero tendere a non fluttuare senza controllo.

### 1.8.2 Il sistema AGENT-0

AGENT-0 è un semplice linguaggio di programmazione che implementa alcuni dei concetti AOP descritti sopra. Poiché AGENT-0 consente di definire un programma per ogni agente compreso nel sistema, non è più necessario dire esplicitamente quale agente sta portando a termine una certa azione; ad esempio, il comando $B_a^t\phi$ diventa $(B(t(\phi)))$ nel corpo del codice associato all'agente $a$. In AGENT-0 il programmatore specifica solo condizioni per fare commitment; i commitment sono costruiti subito e eseguiti in seguito, automaticamente nell'istante appropriato. I commitment sono solo verso azioni primitive, quelle che l'agente può eseguire direttamente. Prima di definire la sintassi dei commitment, sono necessarie altre definizioni di base.
**Fatti.** Le espressioni per i fatti costituiscono un piccolo frammento del linguaggio temporale descritto nel paragrafo precedente: essi sono essenzialmente le frasi atomiche della forma

$$(t\ atom) \text{ e } (NOT(t\ atom))$$

Per esempio, (0 (stored orange 1000)) è un fatto AGENT-0 che afferma che all'istante 0 ci sono 1000 arance in magazzino.
**Azioni private.** La sintassi per le azioni private è

*(DO t p-action)*

dove $t$ è un punto nel tempo e *p-action* è il nome di un'azione privata. Gli effetti delle azioni private possono essere visibili agli altri agenti o meno.
**Azioni comunicative.** Ci sono tre tipi di azioni comunicative:

*(INFORM t a fact)*

in cui $t$ è l'istante nel tempo in cui si verifica l'atto di informare, $a$ è il nome del ricevente e $fact$ è un comando per i fatti.

*(REQUEST t a action)*

dove $t$ è un istante nel tempo, $a$ è il nome del ricevente e *action* è un comando per le azioni.

*(UNREQUEST t a action)*

dove $t$ è un istante nel tempo, $a$ è il nome del ricevente e *action* è un comando per le azioni.
**Non azioni.** Una "nonaction" distoglie un agente dal portare a termine una particolare azione.

*(REFRAIN action)*

**Condizioni mentali.** Una condizione mentale è una combinazione logica di *mental patterns* che possono assumere due forme

*(B fact)*

che significa che l'agente crede $B$ o

*((CMT a) action)*

dove $CMT$ corrisponde a un impegno. L'informazione sul tempo è compresa nei fatti e nelle azioni; un esempio di mental pattern è (B(3(stored orange 950))) che significa che l'agente crede che all'istante 3 si siano lasciate 950 arance in magazzino.

**Capacità.** La sintassi di una capacità è

*(action mentalcondition)*

che significa che l'agente può portare a termine *action* a patto che *mentalcondition* valga.

**Azioni condizionali.** La sintassi di un'azione condizionale è

*(IF mentalcondition action)*

che significa che action può essere portata a termine solo se *mentalcondition* è soddisfatta.

**Messaggi condizionali.** Una message condition è una combinazione logica di *message patterns*, che sono triple

*(From Type Content)*

in cui *From* è il nome del mittente, *Type* è *INFORM*, *REQUEST* o *UNREQUEST* e *Content* è un fatto o un'azione.

**Regola per commitment.** Una regola per il commitment (o commitment rule) ha la forma:

*(COMMIT messagecondition mentalcondition (agent action)*)*

dove *messagecondition* e *mentalcondition* sono rispettivamente message e mental condition, *agent* è il nome di un agente, *action* è un comando action e * denota la ripetizione di zero o più volte. L'intuizione dietro la commitment rule *(COMMIT msgcond mntcond* ($ag_1$ $act_1$) ... ($ag_n$ $act_n$)) nel programma che definisce il comportamento dell'agente $ag$ è che se $ag$ riceve un messaggio che soddisfa *msgcond* e il suo stato mentale verifica la condizione *mntcond*, essa fa si che l'agente $ag$ si impegni a compiere $act_1$ in favore di $ag_1$, ..., e $act_n$ in favore di $ag_n$. Si noti che le commitment rule sono identificate dalla keyword $COMMIT$ e i commitment mental pattern (vedere la definizione sopra) sono identificati dalla keyword $CMT$.

**Program.** Un programma è definito dall'unità di tempo, detta "timegrain", seguita dalle capacità, dalle credenze iniziali e dalle commitment rule di un agente. La granularità del tempo spazia tra $m$ (minuti), $h$ (ore), $d$ (giorni) e $y$(anni).

### 1.8.3 Semantica

Per il linguaggio non è data alcuna semantica formale.

### 1.8.4 Implementazioni

Un prototipo di interprete per AGENT-0 è stato implementato in Common Lisp ed è stato installato su computer Sun/Unix, DecStation/Unix e Macintosh. L'interprete e il manuale per la programmazione sono a disposizione della comunità scientifica. È stata sviluppata un'implementazione separata da Hewlett Packard come parte di un progetto aggiunto volto a incorporare AOP nella architettura New Wave$^{TM}$.

L'interprete di AGENT-0 è caratterizzato dal seguente ciclo composto da due passi:

1. Leggere i messaggi correnti e aggiornare belief e commitment.
2. Eseguire i commitment per il momento corrente, generando possibilmente un ulteriore cambio di belief.

Le azioni a cui gli agenti possono essere indotti comprendono quelle comunicative come informare e richiedere, così come arbitrarie azioni private.

### 1.8.5 Estensioni

Sono state proposte due estensioni di AGENT-0:

- **PLACA** (S. R. Thomas, 1995 [72]) arricchisce AGENT-0 con un meccanismo per la gestione flessibile dei piani. Aggiunge due strutture dati allo stato dell'agente: una lista di intenzioni e una lista di piani. Le intenzioni sono adottate in modo simile a come lo sono i commitment; il comando PLACA ($ADOPT(INTENDx)$) significa che l'agente aggiungerà l'intenzione di fare $x$ alla sua lista di intenzioni. I piani sono creati da un generatore di piani esterno per andare incontro alle intenzioni. Questo approccio dà al sistema l'abilità di alterare dinamicamente i piani che non hanno successo.
- **Agent-K** (W. H. Davies e P. Edwards, 1994 [17]) è un tentativo di standardizzare la funzionalità di passaggio dei messaggi in AGENT-0. Combina la sintassi di AGENT-0 (senza supportare i meccanismi per la pianificazione di PLACA) con il formato di KQML *(Knowledge Query and Manipulation Language* [51]) per assicurare che i messaggi scritti in linguaggi diversi da AGENT-0 siano gestiti. Agent-K introduce due maggiori cambiamenti alla struttura di AGENT-0: primo, rimpiazza le azioni comunicative *INFORM*, *REQUEST*, *UNREQUEST* con un comando *KQML*, che prende come suoi parametri il messaggio, il tempo e il tipo *KQML*; secondo, consente la corrispondenza di più commitment a un singolo messaggio. In AGENT-0 il meccanismo di commitment multiplo non è definito e l'interprete semplicemente seleziona la prima regola che corrisponde a un messaggio.

# Capitolo 2

# AgentSpeak(L)

Il linguaggio AgentSpeak(L), sviluppato da A. S. Rao nel 1996 [61], può essere visto come un'astrazione dei sistemi implementati PRS e dMARS e consente agli agenti di essere definiti e interpretati.

Le operazioni di base degli agenti AgentSpeak(L) si basano su belief, desideri e intenzioni degli agenti. Gli agenti hanno una base di dati per la memorizzazione dei piani disponibili, conosciuta come *libreria dei piani (plan library)*. Gli agenti rispondono a cambiamenti nei propri goal e belief, che risultano da percezioni, e che sono "confezionati" in strutture dati dette *eventi (events)*. Essi rispondono a tali cambiamenti selezionando piani dalla libreria per ogni cambiamento e istanziando uno di tali piani come un'intenzione. Queste intenzioni comprendono azioni e goal o piani da soddisfare, con la possibilità di aggiungere successivamente nuovi piani a tale intenzione. Le operazioni essenziali di un agente AgentSpeak(L) possono essere riassunte come segue.

- Se ci sono più eventi da processare, selezionarne uno.
- Recuperare dalla libreria dei piani tutti i piani *attivabili* da tale evento che possono essere eseguiti nelle circostanze correnti.
- Selezionare uno dei piani così generati per l'esecuzione e generare un'*istanza* di tale piano.
- Aggiungere l'istanza di piano all'intenzione appropriata.
- Selezionare un'intenzione e considerare il prossimo passo del piano al suo top per l'esecuzione.
- Se il piano al top è completo, si considera il piano successivo e se l'intenzione è vuota, l'intenzione ha successo e può essere rimossa dall'insieme delle intenzioni.

Segue una specifica dettagliata, in linguaggio Z [70], dei tipi e delle primitive usati nel sistema, degli agenti e del loro stato, delle operazioni e del ciclo di ragionamento. La notazione Z viene introdotta in Appendice A.

## 2.1 Il sistema AgentSpeak(L)

In questa parte si definiscono i tipi base richiesti per costruire il modello formale.

Tutte le costanti e le variabili sono date mediante gli insiemi denotati rispettivamente come $[Const]$ e $[Var]$. I simboli di funzione sono nell'insieme $[FSym]$. Un *termine* può essere una costante, una variabile o un simbolo di funzione con una sequenza non vuota di termini come parametro.

$$\begin{aligned} Term ::= & \; const\langle\langle Const\rangle\rangle \\ | & \; var\langle\langle Var\rangle\rangle \\ | & \; functor\langle\langle FSym \times \mathrm{seq}_1\ Term\rangle\rangle \end{aligned}$$

I *belief* sono costruiti dalle primitive di cui sopra, usando inoltre l'insieme dei simboli di predicato $[PredSym]$. Un belief *atom* è un simbolo di predicato avente come argomento una sequenza di termini. Un belief *literal* è un atomo o un atomo negato. I belief sono *belief literal* o *congiunzioni* di due belief.

$$\begin{array}{|l} \hline Atom \\ head : PredSym \\ terms : \mathrm{seq}\ Term \\ \hline \end{array}$$

$$Literal ::= pos\langle\langle Atom\rangle\rangle \mid not\langle\langle Atom\rangle\rangle$$

$$Belief ::= literal\langle\langle Literal\rangle\rangle \mid and\langle\langle Belief \times Belief\rangle\rangle$$

La *base beliefs* è un insieme di belief atom privi di variabili. Si definiscono delle funzioni ausiliarie che servono per definire la base beliefs: *beliefvars* che rende l'insieme delle variabili in un belief, *atomvars* che rende l'insieme delle variabili in un atomo e *termvars* che rende l'insieme delle variabili in un termine.

$$BaseBelief == \{b : Belief \mid beliefvars\ b = \varnothing\}$$

Un *goal* è uno stato del sistema che un'agente vuole raggiungere. Ci sono due tipi di goal rappresentati entrambi come un simbolo di predicato e una sequenza di termini con un identificatore appropriato come prefisso. Un *achievement* goal è uno stato del sistema che l'agente vuole raggiungere in cui l'atomo è un belief vero ed è indicato dal prefisso *achieve*. Un *test* goal

si ha quando un agente vuole verificare se l'atomo è un belief vero o meno ed è identificato dal prefisso *query*.

$$Goal ::= achieve\langle\langle Atom\rangle\rangle \mid query\langle\langle Atom\rangle\rangle$$

Per poter soddisfare goal o completare compiti, un agente deve portare a termine delle azioni. L'insieme dei simboli d'azione è $[ActionSym]$ e l'insieme delle azioni è specificato analogamente a quello degli atomi.

*Action*

$name : ActionSym$
$terms : \text{seq}\ Term$

Le precondizioni per l'applicabilità dei piani sono costituite da *triggering event*. I triggering event si hanno quando si percepisce un cambiamento nell'ambiente, o quando si acquisisce un nuovo goal. Si definisce un *trigger* come un belief o un goal, e un triggering event come l'aggiunta o la cancellazione di un trigger.

$$Trigger ::= belief\langle\langle Belief\rangle\rangle \mid goal\langle\langle Goal\rangle\rangle$$
$$TriggerSymbol ::= + \mid -$$
$$TriggerEvent == TriggerSymbol \times Trigger$$

Un *piano* AgentSpeak(L) comprende tre componenti.

- Una *condizione di invocazione* che dettagli le circostanze, in termini di belief o goal, che causano l'individuazione del piano, è specificata mediante un *triggering event*.

- Un *contesto* specifica i belief dell'agente che dovranno essere soddisfatti affinché il piano venga scelto per l'esecuzione.

- Il *corpo* è la parte del piano che specifica la sequenza di azioni e goal che l'agente deve compiere. Entro il corpo si determina se eseguire un'azione, soddisfare un achieve goal o rispondere a un test goal.

$$Invocation == TriggerEvent$$
$$Context == \mathbb{P}\, Belief$$
$$Formula ::= actionformula\langle\langle Action\rangle\rangle \mid goalformula\langle\langle Goal\rangle\rangle$$
$$Body == \mathrm{seq}\, Formula$$

*Plan*
$inv : Invocation$
$context : Context$
$body : Body$

Un'*istanza di piano* è un piano istanziato parzialmente ed è identificata come l'*intended means*. L'istanza di piano rappresenta una copia del piano originale che serve come un'*attitudine mentale* che determina il comportamento. In AgentSpeak(L), al contrario di quanto avviene per altri sistemi BDI, un'istanza di piano ha lo stesso tipo di un piano.

$$PlanInstance == Plan$$

Le *intenzioni* sono i piani che si vogliono eseguire. Un'intenzione è uno stack di istanze di piani, ognuna dei quali porta un contributo al completamento delle istanze di piano ai livelli più bassi, che culmina nel completamento dell'istanza di piano originale in fondo allo stack. Le istanze di piano sullo stack sono inserite secondo il criterio bottom-up.

L'istanza di piano originale in fondo allo stack genera un subgoal da soddisfare e porta all'aggiunta di un ulteriore istanza di piano per eseguire i compiti richiesti per soddisfare tale subgoal. L'istanza di piano al top dello stack è quella correntemente in esecuzione e una volta che è stata eseguita, e che i suoi subtask sono stati completati, viene rimossa. Poi, la formula successiva nell'istanza di piano che segue può essere provata e il piano continua la sua esecuzione, richiedendo possibilmente altre istanze di piano nuove da aggiungere allo stack. Formalmente, un'intenzione è rappresentata come una sequenza non vuota di istanze di piano con il primo elemento della sequenza rappresentante il top dello stack.

$$Intention == \mathrm{seq}_1\, PlanInstance$$

Un *evento* in AgentSpeak(L) può arrivare dall'esterno, nel qual caso è un triggering event esterno non collegato a un'intenzione, o dall'esecuzione di un'intenzione corrente, caso in cui è un triggering event (subgoal) con un collegamento esplicito a un'intenzione. Il tipo degli eventi è definito come segue

$$
\begin{array}{|l}
\textit{Event} \\
\hline
trig : TriggerEvent \\
int : optional[Intention] \\
\hline
\end{array}
$$

$$ExternalEvent == [Event \mid undefined\ int]$$
$$InternalEvent == [Event \mid defined\ int]$$

$$
\begin{array}{|l}
MakeEvent : (TriggerEvent \times optional[Intention]) \rightarrow Event \\
\hline
\forall\, t : TriggerEvent;\ i : optional[Intention] \bullet \\
\quad MakeEvent\ (t, i) = (\mu\, e : Event \mid e.trig = t \wedge e.int = i)
\end{array}
$$

Come visto sopra, in un evento esterno, l'intenzione non è definita mentre in un evento interno lo è. Per costruire un evento si specifica la funzione ausiliaria, *MakeEvent*, che crea un'istanza di tipo *Event* dalle sue parti costituenti.

Un *agente* AgentSpeak(L) è formato da quattro componenti distinte.

- La *libreria dei piani (plan library)* che è una collezione contenente i piani che l'agente può utilizzare. Questi piani sono pre-compilati e non richiedono alcuna pianificazione reale da parte dell'agente. Nello schema che si vedrà in seguito compare una variabile *capabilities* che rappresenta un insieme di azioni che è costruito prendendo ogni azione dal corpo di ogni piano nella libreria.

- La *funzione per la selezione degli eventi (event-selection function)* seleziona da un insieme di eventi un evento da processare. Gli eventi che sono percepiti dall'agente sono accumultati in tale insieme.

- La *funzione per la selezione dei piani applicabili (applicable-plan-selection function)* è responsabile della selezione di un piano tra quelli nella libreria dei piani che sono applicabili nelle circostanze correnti.

- La *funzione per la selezione delle intenzioni (intention-selection function)* seleziona dall'insieme delle intenzioni corrente un'intenzione da eseguire.

La definizione di un agente è schematizzata come segue

$$
\begin{array}{|l}
\textit{AgentSpeakAgent} \\
\hline
planlibrary : \mathbb{P}_1\ Plan \\
intselect : \mathbb{P}_1\ Intention \rightarrow Intention \\
planselect : \mathbb{P}_1\ Plan \rightarrow Plan \\
eventselect : \mathbb{P}_1\ Event \rightarrow Event \\
capabilities : \mathbb{P}_1\ Action \\
\hline
capabilities = \{p : planlibrary;\ a : Action \mid \\
\qquad a \in mapset\ actionformula^{\sim}\ (\operatorname{ran} p.body) \bullet a\} \\
\hline
\end{array}
$$

Un agente non può contenere insiemi di piani vuoti e le sue funzioni di selezione sono definite solo per insiemi non vuoti.

Lo *stato di un agente* durante l'esecuzione è definito da

- le componenti che sono previste da un agente AgentSpeak(L);
- l'insieme dei belief dell'agente;
- l'insieme delle intenzioni;
- l'insieme degli eventi ancora da processare;
- l'insieme delle azioni che l'agente può portare a compimento.

Per chiarire quali sono i goal di un agente in un dato momento si considera anche una variabile *goal* che rappresenta i goal tratti dal corpo di ogni istanza di piano nell'insieme delle intenzioni correnti. Inoltre si ha una variabile *status* con la quale si può tenere traccia di quali delle intenzioni correnti sono attive e di quali sono invece sospese.

$$Status ::= active \mid suspended$$

$$
\begin{array}{|l}
\textit{AgentSpeakAgentState} \\
\hline
AgentSpeakAgent \\
beliefs : \mathbb{P}_1\ Belief \\
intentions : \mathbb{P}\ Intention \\
events : \mathbb{P}\ Event \\
actions : \mathbb{P}\ Action \\
status : Intention \nrightarrow Status \\
goals : \mathbb{P}\ Goal \\
\hline
\operatorname{dom} status = intentions \\
goals = \{i : intentions;\ p : PlanInstance;\ g : Goal \mid \\
\qquad p \in (\operatorname{ran} i) \land g \in (mapset(goalformula^{\sim})(\operatorname{ran} p.body)) \bullet g\} \\
\hline
\end{array}
$$

Prima di cominciare la propria attività un agente ha un insieme iniziale di belief, o base di conoscenza, ma non ha eventi da processare, intenzioni e azioni da portare a compimento.

## 2.2 Ciclo di esecuzione di AgentSpeak(L)

Lo schema di interpretazione di un agente AgentSpeak(L) è evidenziato in Figura 2.1. Un agente può operare in due modi che comprendono il rispondere ad un evento o l'eseguire un'intenzione.

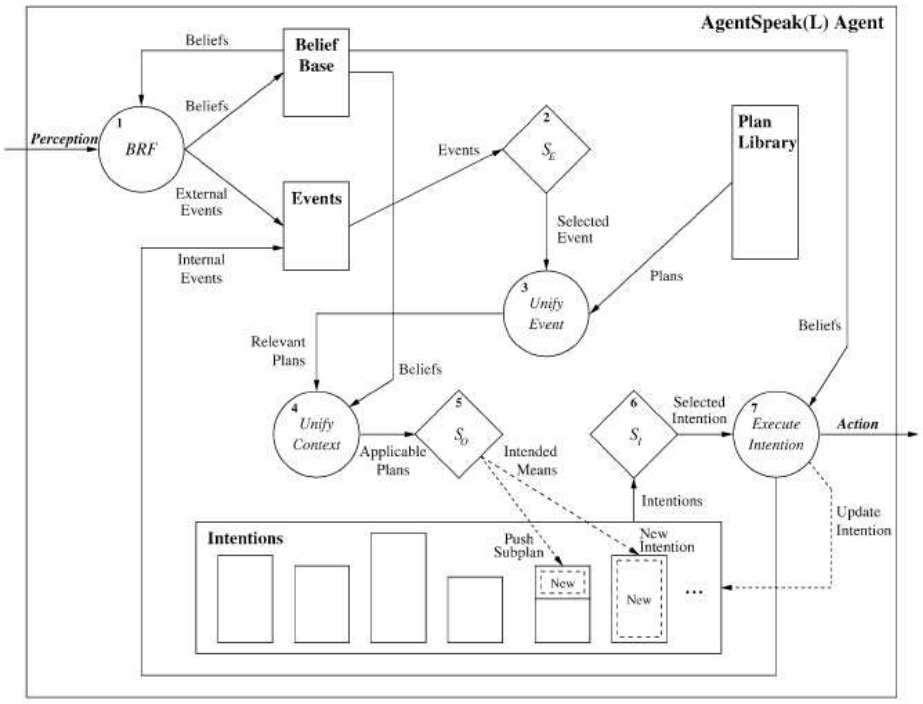



Figura 2.1: Interprete di AgentSpeak(L)

La reazione all'evento si processa come segue.

- L'agente sceglie un evento dall'insieme degli eventi da processare.
- L'agente genera l'insieme dei *piani rilevanti* per l'evento scelto. Un piano è *rilevante* rispetto a un evento se esiste un *mgu* che unifichi l'evento e il piano da invocare. Questo si fa mediante la funzione *genrelplans* che prende un evento e un insieme di piani e genera da questo un insieme di piani *rilevanti* memorizzando inoltre l'unificatore che li rende rilevanti.

- Dall'insieme dei piani rilevanti l'agente genera l'insieme dei *piani applicabili*. Un piano rilevante è *applicabile* se il suo *contesto* è conseguenza logica dei belief dell'agente. Questa conseguenza logica si determina mediante un predicato *LogicalConsequence*.

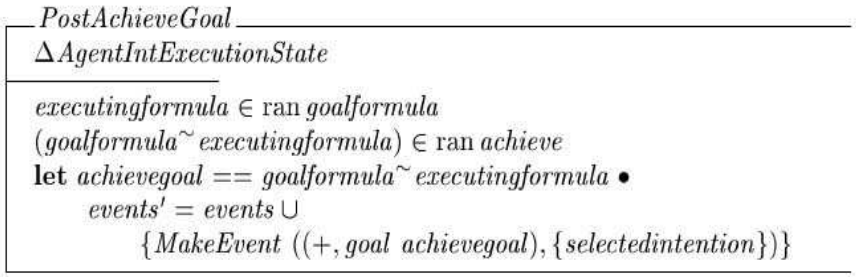
  I piani applicabili sono definiti dalla funzione *genapplplans* che prende un insieme di coppie di piani rilevanti, le sostituzioni ad essi associate e un insieme di belief e restituisce quei piani che sono applicabili insieme a un ulteriore sostituzione che, applicata al contesto del piano, lo rende conseguenza logica dei belief dell'agente. Tale sostituzione viene poi composta con la sostituzione associata al piano generata in precedenza.

- Mediante una funzione apposita si seleziona dall'insieme dei piani applicabili un piano che viene istanziato applicandogli la composizione di sostituzioni ottenuta in precedenza.

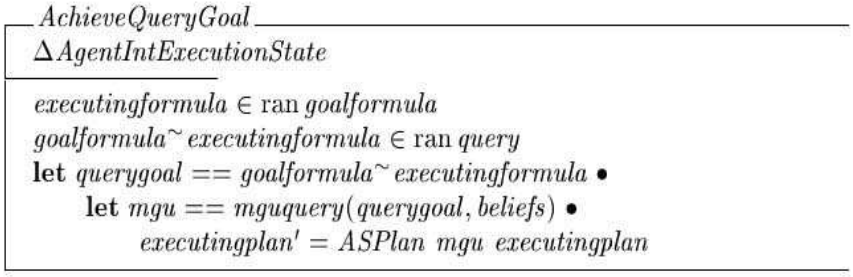
  Questo piano istanziato (intended means) viene usato per aggiornare le intenzioni dell'agente come segue: se l'evento processato è esterno, si crea una nuova intenzione contenente solo l'istanza di piano. Se l'evento è interno, l'istanza di piano è messa sull'intenzione che ha generato tale evento interno. In entrambi i casi lo stato dell'intenzione diviene *active*.

L'esecuzione delle intenzioni avviene dopo che queste sono state aggiornate. Viene selezionata un'intenzione tra quelle aventi stato *active* per l'esecuzione che viene riferita come *intenzione selezionata (selected intention)*. Il piano istanziato in testa a tale stack d'intenzioni è definito come *piano in esecuzione (execution plan)* e la prossima formula in tale piano è definita come *formula in esecuzione*. A seconda dell'intenzione scelta e della formula eseguita del piano in esecuzione, si hanno tre possibili linee di condotta:

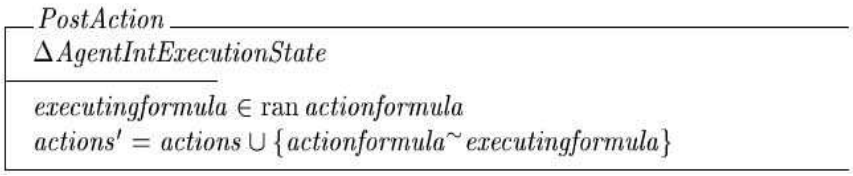
- Se la formula è un *achieve goal*, non si ci impegna subito a soddisfarlo e si crea un evento per il goal. Questo evento è aggiunto all'insieme degli eventi correnti che l'agente deve processare in modo che si possa cercare un piano per soddisfare il goal e far continuare così l'esecuzione dell'intention stack corrente. In questo caso lo stato dell'intenzione diviene *suspended*. Questa parte nella specifica originale data da Rao [61] non è descritta esplicitamente.



- Se la formula è un *query goal* che va unificato con l'insieme dei belief correnti, allora l'mgu risultante viene applicato al resto del piano in esecuzione.




- Se la formula è un'*azione* la si aggiunge all'insieme delle azioni da portare a compimento.




La descrizione originale di AgentSpeak(L) non indica espressamente cosa succede quando un'azione è compiuta, con successo o con fallimento, o quando un query goal non può essere unificato con i belief. Per il sistema quest'omissione è rilevante, tuttavia, sia in caso di soddisfacimento di un query goal, sia in caso di completamento di un'azione, si procede rimuovendo la formula in esecuzione dal corpo del piano eseguito.

Se, dopo la rimozione della formula, non ci sono più formule nel piano corrente ma ci sono ancora piani nello stack dell'intenzione, allora l'esecuzione del piano corrente è completata e il piano successivo è pronto per essere eseguito. Tutti i legami che sono stati introdotti con le sostituzioni applicate nel corso dell'esecuzione del piano corrente non vanno dimenticate e quindi si procede applicando l'mgu del goal che è servito per scegliere il piano appena eseguito, al piano che lo segue sullo stack. Il goal che ha generato il piano la cui esecuzione è terminata viene rimosso dall'insieme degli eventi (questa necessità operazionale non è considerata nella descrizione originale di AgentSpeak(L) [61]).

Se, dopo il soddisfacimento della formula, il piano è stato completato e non ci sono altri piani nell'intenzione in esecuzione, allora l'intenzione è soddisfatta e può essere rimossa dall'insieme delle intenzioni dell'agente. Analogamente, l'evento esterno che ha generato tale intenzione viene rimosso. Nella formalizzazione di questo passo si è evidenziato un problema. Un evento esterno non può essere identificato dall'intenzione completata

perché la componente intenzione di tale evento è *indefinita*. Può essere identificato se c'è solo un evento esterno il cui trigger può essere unificato con la condizione di invocazione del piano completato. Comunque in questa specifica si assume che una soluzione generale potrebbe essere quella di associare a ogni evento un identificatore unico che venga memorizzato come parte dell'istanza di piano da esso generata.

Inoltre si descrive un agente che non ha completato un piano dopo la rimozione della formula.

*NotAchievePlan*

$$\Xi AgentIntExecutionState$$

$$executingplan.body \neq \langle\rangle$$

Infine, si deve specificare cosa succede quando un'azione viene aggiunta all'insieme delle azioni da completare o quando un query goal è soddisfatto. Ci sono tre possibilità: la prima prevede che solo il piano è stato soddisfatto, nel qual caso la prima formula del piano successivo deve essere rimossa (in quanto aveva generato il piano completato); la seconda prevede che il piano e l'intenzione corrispondente siano entrambi soddisfatti; la terza prevede che il piano non venga soddisfatto in quanto formato da più formule. L'operazione totale, CHECKPLAN, con precondizione *true*, è definita come disgiunzione di questi schemi. Si usa la composizione di operazioni denotata da questo schema:

$$CHECKPLAN == (AchievePlanOnly \fatsemi RemoveFormula) \lor AchievePlanAndIntention \lor NotAchievePlan$$

Con questo, si può anche specificare l'operazione di un agente che esegue una formula. Nell'esecuzione di una formula, i possibili risultati sono l'aggiunta di un achieve goal all'insieme degli eventi e la sospensione di un'intenzione; l'aggiunta di un'azione a quelle da completare con la rimozione della formula dell'azione; la soddisfazione di un query goal con la rimozione della formula per il goal. Negli ultimi due casi il piano deve essere controllato come descritto in precedenza dopo la rimozione della formula.

## 2.3 Semantica operazionale di AgentSpeak(L)

Di seguito sono fornite una breve descrizione sintattica di AgentSpeak(L) e la descrizione di una sua semantica operazionale [53].

### Sintassi

Segue una breve descrizione della sintassi di AgentSpeak(L) tramite la grammatica:

$$
\begin{array}{lcll}
ag & ::= & bs \quad ps & \\
bs & ::= & b_1 \ldots b_n & (n \geq 0) \\
at & ::= & \mathtt{P}(t_1, \ldots, t_n) & (n \geq 0) \\
ps & ::= & p_1 \ldots p_n & (n \geq 1) \\
p & ::= & te : ct \leftarrow h & \\
te & ::= & +at \mid -at \mid +g \mid -g & \\
ct & ::= & at \mid \neg at \mid ct \wedge ct \mid \mathsf{T} & \\
h & ::= & a \mid g \mid u \mid h; h & \\
a & ::= & \mathtt{A}(t_1, \ldots, t_n) & (n \geq 0) \\
g & ::= & !at \mid ?at & \\
u & ::= & +at \mid -at &
\end{array}
$$

Figura 2.2: Sintassi astratta di AgentSpeak(L)

Un agente AgentSpeak(L) è specificato da un insieme *bs* di belief (la base beliefs iniziale dell'agente) e da un insieme *ps* di piani (la libreria dei piani dell'agente). La formula atomica *at* del linguaggio è un predicato dove `P` è un simbolo di predicato e $t_1, ..., t_n$ sono termini standard della logica del prim'ordine. Un *belief* è una formula atomica *at* senza variabili ed è indicato con la metavariabile *b*.

Un piano in AgentSpeak(L) è rappresentato dall'espressione $te : ct \leftarrow h$, dove *te* è il *triggering event*, *ct* è il contesto che specifica la condizione che determina la messa in esecuzione del piano, *h* è una sequenza di azioni, goal o aggiornamenti di belief; $te : ct$ è la *testa* del piano e *h* è il *corpo*. La libreria dei piani *ps* è una lista di piani. La formula *ct* deve essere una conseguenza logica della base beliefs se si vuole che il piano sia considerato applicabile.

Un triggering event *te* può essere l'aggiunta o la cancellazione di un belief dalla base beliefs di un agente ($+at$ e $-at$ rispettivamente), o l'aggiunta o la cancellazione di un goal ($+g$ e $-g$ rispettivamente).

L'agente ha a sua disposizione un insieme di *azioni* e gli elementi di tale insieme sono indicati con la metavariabile *a*. Un'azione è un predicato con un simbolo di azione `A`. Un goal *g* può essere un *achievement goal* ($!at$) o un *test goal* ($?at$). Infine, $+at$ e $-at$ (nel corpo del piano) rappresentano le operazioni *u* di aggiornamento della base beliefs che, rispettivamente aggiungono e rimuovono *at*.

### Semantica

Un agente e la sua circostanza formano una configurazione del sistema di transizione che fornisce la semantica operazionale di AgentSpeak(L). La

relazione di transizione

$$\langle ag, C\rangle \rightarrow \langle ag', C'\rangle$$

è definita mediante regole semantiche.

La circostanza dell'agente $C$ è una tupla $\langle I, E, A, R, Ap, \iota, \rho, \epsilon\rangle$ dove:

- $I$ è un insieme di *intenzioni* $\{i, i', ...\}$. Ogni intenzione $i$ è uno stack di piani parzialmente istanziati.

- $E$ è un insieme di *eventi* $\{(te, i), (te', i'), ...\}$. Ogni evento è una coppia $(te, i)$ in cui $te$ è un triggering event e il piano al top dell'intenzione $i$ è quello che ha generato $te$.
  Quando la funzione di revisione del belief aggiorna la base beliefs, l'evento associato viene incluso in questo insieme.

- $A$ è un insieme di *azioni* da compiere nell'ambiente. Un'azione inclusa in tale insieme dice alle altre componenti dell'architettura di portare a termine le rispettive operazioni di modifica sull'ambiente.

- $R$ è un insieme di *piani rilevanti*. Questo insieme è ottenuto come descritto dalla Definizione 1 che compare nel seguito.

- $Ap$ è un insieme di *piani applicabili*. L'insieme si ottiene come specificato dalla Definizione 2 che compare nel seguito.

- Ogni circostanza $C$ ha anche tre componenti dette $\iota$, $\epsilon$ e $\rho$. Queste memorizzano, rispettivamente, una particolare intenzione, un particolare evento e un particolare piano applicabile che saranno considerati durante l'esecuzione dell'agente.

Si specificano delle funzioni ausiliarie che servono per le regole semantiche. Se $p$ è un piano della forma $te : ct \leftarrow h$, si definiscono $\mathsf{TrEv}(p) = te$ e $\mathsf{Ctxt}(p) = ct$.

Un piano è considerato rilevante in relazione a un triggering event se è stato progettato per soddisfare tale evento. La rilevanza si verifica tentando di unificare il triggering event del piano con il triggering event che è stato scelto da $E$ per essere gestito. Nelle definizioni che seguono si determina l'*mgu* che calcola la sostituzione più generale che unifica due triggering events.

**Definizione 1.** *Dati i piani ps di un agente e un triggering event te, l'insieme* $\mathsf{RelPlans}(ps, te)$ *dei piani rilevanti è ottenuto come segue:*

$$\mathsf{RelPlans}(ps, te) = \{p\vartheta \mid p \in ps \wedge \vartheta = \mathsf{mgu}(te, \mathsf{TrEv}(p))\}$$

Un piano è applicabile se è rilevante e se il suo contesto è conseguenza logica dei belief dell'agente.

**Definizione 2.** *Dati un insieme di piani rilevanti $R$ e i belief $bs$ di un agente, l'insieme dei piani applicabili* $\mathsf{AppPlans}(bs, R)$ *è definito come segue:*

$$\mathsf{AppPlans}(bs, R) = \{p\vartheta \mid p \in R \wedge \vartheta \textit{ t.c. } bs \models \mathsf{Ctxt}(p)\vartheta\}$$

Un agente può anche portare a compimento un test goal. La valutazione di un test goal $?at$ consiste nel verificare se la formula $at$ è conseguenza logica dei belief dell'agente. Uno degli effetti di questo controllo è la produzione di un insieme di sostituzioni.

**Definizione 3.** *Dati i belief $bs$ di un agente e una formula $at$, l'insieme delle sostituzioni* $\mathsf{Test}(bs, at)$ *prodotto dalla verifica di $at$ rispetto a $bs$ è definito come segue:*

$$\{\mathsf{Test}(bs, at) = \vartheta \mid bs \models at\vartheta\}$$

Per avere regole semantiche chiare si adottano le seguenti notazioni:

- Se $C$ è una circostanza, si scrive $C_E$ per indicare la componente $E$ di $C$. Per le altre componenti di $C$ ci si comporta analogamente.
- Si scrive $C_\iota = \_$ per indicare che non ci sono intenzioni considerate nell'esecuzione dell'agente. Analogo discorso per $C_\rho$ e $C_\epsilon$.
- Si usano $\iota, \iota', ...$ per denotare intenzioni e $i[p]$ per indicare un'intenzione che ha il piano $p$ al top con $i$ come indicatore dei piani rimanenti in tale intenzione.

Le funzioni di selezione sono indicate con questa notazione: $S_E$ come funzione di selezione per gli eventi, $S_{Ap}$ come funzione di selezione per i piani applicabili e $S_I$ come funzione di selezione per le intenzioni.

Di seguito vengono riassunte le principali regole semantiche [53]:

**Selezione di eventi:** si usa la funzione $S_E$ per selezionare un evento dall'insieme $E$. L'evento scelto viene rimosso da $E$ e assegnato alla componente $\epsilon$ della circostanza.

**Piani rilevanti:** si istanzia la componente $R$ con l'insieme dei piani rilevanti. Se non ci sono piani rilevanti l'evento viene scartato da $\epsilon$.

**Piani applicabili:** si istanzia la componente $Ap$ con l'insieme dei piani applicabili. Se non ci sono piani applicabili l'evento viene scartato da $\epsilon$. In ogni caso anche i piani rilevanti vengono scartati.

**Selezione del piano applicabile:** si assume l'esistenza di una funzione di

selezione $S_{Ap}$ che sceglie un piano dall'insieme dei piani applicabili $Ap$. Il piano scelto è assegnato alla componente $\rho$ della circostanza e l'insieme dei piani applicabili è scartato.

**Preparare l'insieme delle intenzioni:** gli eventi possono essere classificati in esterni o interni. Se l'evento $\epsilon$ è esterno si crea una nuova intenzione e il suo unico piano è il piano $p$ assegnato alla componente $\rho$. Se l'evento è interno il piano in $\rho$ dovrebbe essere messo in cima all'intenzione associata all'evento. In entrambi i casi l'evento e il piano possono essere rimossi dalle rispettive componenti $\epsilon$ e $\iota$. L'intera intenzione $i$ che ha generato l'evento interno viene reinserita in $C_I$ con $p$ al suo top. Questo è collegato all'intenzione sospesa nel corso del soddisfacimento di un achievement goal.

**Selezione dell'intenzione:** con la funzione $S_I$ che seleziona un'intenzione per l'esecuzione.

**Esecuzione del corpo dei piani:** questo gruppo di regole esprime gli effetti dell'esecuzione del corpo dei piani. Il piano da eseguire è sempre quello al top dell'intenzione che è stata precedentemente selezionata. Tutte le regole di questo gruppo rimuovono l'intenzione $\iota$. Dopo questo, un'altra intenzione può essere eventualmente selezionata.

- *Azione di base:* l'azione $a$ nel corpo del piano viene aggiunta all'insieme delle azioni $A$ e viene rimossa dal corpo del piano. L'intenzione viene aggiornata in modo da riflettere questa rimozione.

- *Achievement goal:* si registra nell'insieme $E$ degli eventi un nuovo evento interno che potrà essere eventualmente selezionato. L'intenzione che ha generato l'evento interno viene rimossa dall'insieme delle intenzioni $C_I$. In pratica l'*intenzione è sospesa*.

- *Test goal:* quando viene eseguito un test goal $?at$ si hanno a disposizione due regole. Entrambe le regole tentano di produrre un insieme di sostituzioni che rendano $at$ conseguenza logica dei belief dell'agente. La prima dice che se non si trovano sostituzioni non si fa nulla mentre la seconda dice che una delle sostituzioni è applicata al piano.

- *Aggiornamento dei belief:* l'aggiornamento è realizzato mediante due regole. Una delle due aggiunge un nuovo evento all'insieme $E$ degli eventi. La formula $+b$ è rimossa dal corpo del piano e l'insieme delle intenzioni è aggiornato in modo appropriato. L'altra regola lavora in modo analogo. In entrambe le regole, l'insieme dei belief dell'agente dovrebbe essere modificato opportunamente aggiungendo o togliendo il predicato $b$.

**Rimozione delle intenzioni:** si effettua mediante due regole. La prima rimuove un'intenzione dall'insieme delle intenzioni di un agente quando non rimane nulla in quell'intenzione. La seconda regola rimuove dall'intenzione lasciata il piano che è stato messo al top dell'intenzione stessa per conto dell'achievement goal $!at$, anch'esso rimosso in quanto completato.

## 2.4 Estensioni

Sono state proposte diverse estensioni per il linguaggio AgentSpeak(L). Di seguito sono descritte le due estensioni che saranno alla base di un'implementazione di AgentSpeak(L) esteso per il supporto della cooperatività.

### 2.4.1 AgentSpeak(XL)

AgentSpeak(XL) è stato sviluppato da Bordini *et al.* [6]. Lo scopo di questo linguaggio è migliorare AgentSpeak(L) in vari aspetti mediante l'uso del framework TÆMS [20] e dello scheduler DTC [74] come linguaggi di rappresentazione. Si tenta di generale una funzione di selezione delle intenzioni efficiente sfruttando l'uso di DTC on-the-fly.

TÆMS (Task Analysis, Environment Modeling, and Simulation) è un framework indipendente dal dominio per la rappresentazione formale di aspetti di coordinazione di problemi entro sistemi multi-agente. È usato per trattare ambienti in cui un goal ha un grado di soddisfacimento associato e una scadenza. Ci sono delle strutture per i compiti (o task) che l'agente vede in modo diverso nel tempo a causa di incertezza o di cambiamenti dinamici nell'ambiente. Il concetto centrale di TÆMS è la *task structure (o struttura dei compiti)* in cui sono memorizzate varie informazioni relative a goal, obiettivi, compiti che si vogliono soddisfare, metodi per soddisfarli, grado di soddisfacimento (in termini di caratteristiche misurabili quali qualità, costo e durata) e relazioni tra compiti. I compiti sono considerati in gruppo e la loro qualità dipende da cosa (sottocompiti o metodi) viene eseguito e quando.

DTC (Design-To-Criteria Scheduler) usa approcci di indipendenti dal dominio, realtime e computazione flessibile per compiere la schedulazione di compiti. DTC ragiona efficientemente su qualità, costi e durata dei metodi collegati e costruisce un insieme di schedule soddisfacenti per goal di alto livello. Nell'estensione si usano solo i moduli DTC.

#### Estensione del linguaggio

Si sono inserite varie estensioni allo scopo di ovviare ad alcune carenze di AgentSpeak(L). Si è definito un preciso meccanismo per gestire gli eventi in caso di assenza di piani applicabili; si è introdotta la comunicazione tra agenti secondo lo stile di KQML [51], linguaggio di comunicazione per agenti

che si vedrà in seguito, modificando anche l'interprete affinché rifletta gli effetti di questa comunicazione. L'algoritmo di unificazione è stato modificato per poter usare variabili non istanziate negli atomi negati dei belief entro i contesti dei piani. Infine, sono state progettate delle modifiche alla gestione degli eventi. Per molte di queste estensioni non è ancora stata data una semantica formale.

L'integrazione con DTC avviene specificando delle etichette che consentano di identificare univocamente un piano entro la libreria dei piani. Un'etichetta $l$ è separata dal resto del piano da un simbolo "–>" per cui la definizione di piano ora appare così: "$l$ –> $e : b_1$& ...& $b_m$ <– $h_1; ...; h_n$.".

Per permettere l'uso di costrutti base di ogni linguaggio di programmazione il linguaggio è stato esteso perché supporti l'esecuzione di *azioni interne*. Queste azioni sono tali da non toccare l'ambiente condiviso dagli agenti in una società e proprio per questa caratteristica possono essere eseguite istantaneamente, senza dover attendere un nuovo ciclo di esecuzione dell'interprete. Per queste loro caratteristiche le azioni interne possono essere usate sia nel contesto sia nel corpo dei piani. La possibilità di usare le azioni interne entro il contesto dei piani è piuttosto importante perché spesso si ha bisogno di accedere alle procedure nella libreria per decidere se un piano è applicabile o meno senza correre il rischio di alterare l'ambiente. Sintatticamente le azioni interne hanno un '.' nel nome che è usato per separare il nome della libreria dal nome dell'azione. Questa sintassi ha due vantaggi: il primo è quello di consentire all'interprete la differenziazione tra azioni interne e formule per predicati da controllare nel contesto di un piano, o per differenziare azioni interne e di base entro il corpo del piano; il secondo è quello di consentire al programmatore di organizzare le nuove azioni definite in varie librerie.

AgentSpeak(XL) fornisce una *libreria standard* che definisce alcuni operatori utili. Questa libreria è priva di nome.

La definizione delle azioni interne è stata utile per l'integrazione con DTC. Oltre alla libreria standard si ha la libreria delle task structure che consente l'uso di DTC per la selezione delle intenzioni.

### Estensione dell'interprete

Per ottenere una funzione di selezione delle intenzioni efficiente si procede secondo l'idea che segue. Lo scheduler DTC produce, per una data stack structure di TÆMS, sequenze alternative in cui un agente dovrebbe eseguire i metodi in tale task structure in modo da soddisfare i criteri e le scadenze specificate per essi nella task structure dei metodi collegati. Entro una stack structure TÆMS le etichette del metodo sono le etichette che di fatto identificano univocamente le istanze di piano che sono correntemente nell'insieme delle intenzioni. Il programmatore, inoltre, può fornire un valore specifico per i criteri di schedulazione di ogni piano e definire le relazioni tra

un piano e ogni altro. Da queste premesse segue che applicando DTC alla stack structure si genera l'ordine migliore secondo cui i piani candidati per la selezione delle intenzioni possono essere scelti per l'esecuzione.

Per consentire l'integrazione con DTC è necessario memorizzare insieme i criteri di schedulazione correnti e le relazioni per i piani parzialmente istanziati attualmente all'interno dell'insieme delle intenzioni in modo da poter generare una task structure di TÆMS rappresentante lo stato particolare dell'insieme delle intenzioni. Il motivo per cui si adotta questa soluzione è che i criteri e le relazioni possono cambiare nel corso dell'esecuzione di un piano.

C'è un algoritmo che consente di tradurre l'insieme delle intenzioni in una task structure di TÆMS [20] su cui poi far girare DTC ottenendo una funzione di selezione efficiente. La funzione di selezione delle intenzioni si ottiene semplicemente leggendo gli schedule di etichette di piani restituiti da DTC. Tutto quello che la funzione di selezione deve fare è prendere la prima formula nel corpo dell'istanza di piano la cui etichetta è all'inizio dello schedule fornito da DTC. Quando la formula eseguita segna la fine del piano l'etichetta del piano viene rimossa dallo schedule.

### 2.4.2 AgentSpeak(L) esteso per introdurre la comunicazione basata su Speech-Act

Questo lavoro realizzato da Bordini *et al.* [54] propone un'estensione della semantica operazionale di AgentSpeak(L) con lo scopo di far considerare agli agenti AgentSpeak(L) le principali forze illocutorie relative alla comunicazione. La semantica dice esattamente come implementare il trattamento dei messaggi ricevuti da un agente.

La prima cosa che si fa per consentire agli agenti il trattamento dei messaggi è il cambiamento della sintassi in modo da poter associare a ogni belief la sua sorgente. Questa modifica aumenta il potere espressivo dell'agente e consente l'uso di informazioni aggiuntive esplicite per il ragionamento dell'agente.

Si introduce nella grammatica una nuova regola che permette di annotare ogni proposizione atomica con l'informazione relativa alla sua sorgente: un termine *id* che identifica qual è l'agente nella società che precedentemente ha mandato la proposizione in un messaggio; `self` che denota un belief interno o `percept` che indica che il belief è stato acquisito come percezione dall'ambiente.

$$at ::= \mathtt{P}(t_1, ..., t_n)[an_1, ..., an_m]$$

dove $n \geq 0$, $m > 0$ e $an_i \in \{\mathtt{percept}, \mathtt{self}, id\}$ con $0 < i \leq m$. Con questo nuovo costrutto è possibile conoscere con certezza qual è la sorgente di un belief entro un contesto di piano prima di usare tale piano come istanza di piano.

Un'altra modifica viene introdotta nella definizione di circostanza in quanto serve un insieme $M$ per rappresentare la *mail box* dell'agente. Assumendo che l'interprete di AgentSpeak(L) implementato fornisca un meccanismo per inviare e ricevere messaggi in modo asincrono, è necessaria una mail box in cui inserire tali messaggi. All'inizio del ciclo di ragionamento da questa mail box si estrae un messaggio come punto di partenza dell'esecuzione.

Il formato di messaggi è $\langle Ilf, id, content \rangle$ dove

- $Ilf \in \{Tell, Untell, Achieve, Unachieve, TellHow, UntellHow\}$ è la forza illocutoria associata al messaggio;
- *id* identifica l'agente che ha mandato il messaggio;
- *content* è il contenuto del messaggio che può essere una proposizione atomica ($at$) o un piano ($p$).

Delle forze illocutorie indicate solo $Tell$, $Untell$, $Achieve$ e $Unachieve$ corrispondono a performative previste da KQML [24].

Una circostanza $C$ per un agente è definita dalla tupla

$$\langle I, E, M, A, R, Ap, \iota, \rho, \epsilon \rangle$$

dove $M$ è l'insieme dei messaggi che l'agente ha ricevuto e non ancora processato, cioè la mail box.

Per processare i messaggi serve una nuova funzione di selezione $S_M$ che seleziona da $M$ un particolare messaggio e funziona analogamente alle altre funzioni di selezione di AgentSpeak(L).

Inoltre servono due nuove funzioni che fanno parte della specifica di un agente [7]:

- *Trust(id)* restituisce vero se *id* identifica una sorgente di informazione fidata. È usata per decidere se i messaggi $Tell$ saranno processati in quanto, in base all'informazione sulla sorgente di un belief acquisito mediante comunicazione, si può decidere di ignorare i messaggi provenienti da agenti non fidati.
- *Power(id)* risulta vero se l'agente ha una relazione di subordinazione verso *id*. In tal caso, messaggi del tipo *Achieve* dovrebbero essere processati.

Una volta specificati questi nuovi concetti si procede con la specifica delle regole semantiche per il trattamento dei messaggi speech-act based ricevuti da un agente AgentSpeak(L).

**Ricezione di un messaggio Tell:** il contenuto del messaggio è aggiunto alla base beliefs, se non c'era già, e il mittente del messaggio è aggiunto

all'insieme degli agenti che hanno dato credito a tale belief.

**Ricezione di un messaggio Untell:** il mittente del messaggio è tolto dall'insieme degli agenti che danno credito al belief e se il mittente è l'unica sorgente per il belief questo viene rimosso dalla base beliefs.

**Ricezione di un messaggio Achieve:** se il mittente ha potere sull'agente che ha ricevuto il messaggio, allora questo agente tenterà di eseguire un piano il cui triggering event sia del tipo +!*contenuto del messaggio* e per ottenere questo mette nell'insieme degli eventi un evento esterno con la relativa intenzione.

**Ricezione di un messaggio Unachieve:** si ci comporta come nel caso precedente tranne per il fatto che questa regola rimuove un achievement goal dall'insieme degli eventi. Se l'agente ha un piano con tale triggering event, esso dovrebbe gestire tutti gli aspetti dell'abbandono dell'intenzione. Tuttavia, fare questo in pratica richiede l'alterazione dell'insieme delle intenzioni e quindi un meccanismo speciale che in AgentSpeak(L) non era ancora disponibile.

**Ricezione di un messaggio Tell-How:** la regola per questo caso è applicata quando si usa la forza illocutoria *TellHow*. Questo tipo di messaggio si usa quando una sorgente esterna vuole informare un agente AgentSpeak(L) di un piano che essa usa per gestire certi tipi di evento. Se la sorgente esterna è fidata, il piano (che è il contenuto del messaggio) è aggiunto alla libreria dei piani dell'agente.

**Ricezione di un messaggio Untell-How:** la regola usata è simile alla precedente. Una sorgente esterna può ritenere che un piano non sia più valido, o efficiente, per gestire degli eventi per cui era assunto come gestore. Quindi, quando riceve un messaggio con forza illocutoria *UntellHow*, l'agente elimina il piano (che è il contenuto del messaggio) dalla libreria dei piani.

## 2.5 Implementazioni

Ci sono varie implementazioni per il linguaggio AgentSpeak(L) e sue varianti o estensioni. Quelle citate di seguito sono solo alcune e sono state scelte perché sono progetti che implementano alcuni concetti introdotti con le estensioni viste in precedenza o che sono alla base di queste.

- **SIM_Speak** uno dei primi interpreti per AgentSpeak(L) funzionanti [48]. Gira sul toolkit SIM_AGENT di Sloman [69], un "testbed" per

architetture per agenti “cognitivamente ricche”.

- **Jason** [8] è un interprete per una versione di AgentSpeak(L) che considera l’estensione del linguaggio con comunicazione basata su speech-act descritta in precedenza [54] e che consente di avere agenti distribuiti sulla rete usando SACI [36].

AgentSpeak(L) e le sue estensioni sono stati applicati per programmare agenti animati inclusi in ambienti virtuali e per effettuare ricerche sulla realizzazione di sistemi musicali interattivi che usino sistemi multi-agente.

# Capitolo 3

# Linguaggi di comunicazione per agenti

Una società di agenti intelligenti software si colloca in un ambiente computazionale di tipo altamente distribuito, eterogeneo, estremamente dinamico e con un gran numero di nodi autonomi. Questo sistema deve affrontare tre problemi basilari:

- il modello client-server, architettura predominante su Internet, è troppo restrittivo;
- è necessario gestire diverse forme di eterogeneità;
- mancano strumenti e tecniche per la costruzione di client e server intelligenti o di software basato sugli agenti in generale a fronte della maturazione di altre tecnologie software (simulazione di eventi, ragionamento basato sulla conoscenza, recupero di informazioni avanzato, ecc.) pronte per dare il loro contributo all'interno di un ambiente di questo tipo.

Quando si descrivono gli agenti come intelligenti ci si riferisce alle loro abilità: comunicare con altri usando un linguaggio di comunicazione espressivo; lavorare insieme in modo cooperativo per soddisfare goal complessi; agire di propria iniziativa; usare informazioni e conoscenze locali per gestire risorse locali e richieste da agenti alla pari. Per questi motivi i linguaggi che facilitano la comunicazione ad alto livello sono una componente essenziale delle architetture software per agenti intelligenti.

Trattando con sistemi di agenti software del tipo descritto sopra si hanno a disposizione due categorie di tecnologie: linguaggi per agenti e protocolli di coordinazione. Tra i linguaggi per agenti sono compresi, oltre ai linguaggi usati per implementare gli agenti, quelli di comunicazione pensati specificamente per descrivere e facilitare la comunicazione tra due o più agenti. Questi linguaggi, detti *agent communication languages (ACL)*, si occupano

strettamente della comunicazione tra entità computazionali e sono più di un protocollo per lo scambio di dati, in quanto consentono di comunicare anche qual è l'atteggiamento che determina lo scambio tra gli agenti. In definitiva un ACL può essere pensato come un protocollo di comunicazione che supporta diversi tipi di messaggio.

Un linguaggio di comunicazione per agenti per essere identificato come tale deve possedere diverse caratteristiche che possono risultare anche in contrasto tra loro. Questi requisiti sono suddivisibili in sette categorie:

- **forma**: un buon ACL deve essere dichiarativo, sintatticamente semplice e leggibile. Deve essere coinciso, semplice da parserizzare e generare. Il linguaggio deve essere lineare o facilmente traducibile in una sequenza lineare di caratteri o bit, in modo che la trasmissione dei comandi tra gli agenti sia semplice e possa sfruttare il meccanismo di trasporto sottostante. La sintassi deve essere estensibile e dovrebbe essere comprensibile ai più facendo uso di semplice "zucchero sintattico".

- **contenuto**: un ACL deve essere stratificato in modo da adattarsi bene ad altri sistemi. Questa stratificazione dovrebbe distinguere tra il linguaggio di comunicazione, che esprime atti comunicativi, e il linguaggio per il contenuto, che esprime fatti relativi al dominio. Il linguaggio deve prevedere un insieme di atti di comunicazione ben definiti (primitive) che possa però essere esteso. Questo nucleo di primitive deve assicurare l'utilizzo del linguaggio da parte di vari sistemi e catturare parte dell'intuizione relativa a cosa costituisce un atto di comunicazione rispetto ad un'applicazione. Queste primitive sono scelte in modo che il linguaggio per il contenuto consenta solo un insieme ristretto di azioni di comunicazione e questo comporta uno svantaggio dovuto al fatto che tutte le applicazioni, indipendentemente dalla loro natura (basi di dati, basi di conoscenza, sistemi orientati agli oggetti, ecc.), devono usare lo stesso linguaggio per i contenuti.

- **semantica**: è un aspetto spesso trascurato durante la progettazione di un ACL. La descrizione della semantica di un linguaggio di comunicazione e delle sue primitive è spesso limitata alle descrizioni del linguaggio naturale. Le proprietà che deve esibire sono le stesse prospettate per ogni altro linguaggio. La semantica deve essere non ambigua. Considerando che un linguaggio di comunicazione si occupa di interazione tra applicazioni dislocate nel tempo e nello spazio, la semantica si deve occupare di localizzazione e temporizzazione.

- **implementazione**: deve essere efficiente in termini di velocità e utilizzazione dell'ampiezza di banda. Deve adattarsi bene alle tecnologie software esistenti. L'interfaccia deve essere facile da usare e deve

nascondere all'utente i dettagli degli atti di comunicazione che stanno sotto al livello di rete. L'implementazione deve essere facilmente integrabile con le interfacce di applicazioni per una grande varietà di linguaggi, comprendenti linguaggi procedurali (C e Lisp), linguaggi di scripting (Tcl e Perl), linguaggi orientati agli oggetti (Smalltalk e C++), linguaggi di programmazione logica (Prolog). Il linguaggio, inoltre, potrebbe ricondurre a implementazioni parziali in quanto certi agenti potrebbero dover gestire solo un sottoinsieme degli atti comunicativi primitivi.

- **networking**: un ACL deve adattarsi alle moderne tecnologie di rete. Il linguaggio deve supportare tutti i tipi di connessione base (point-to-point, multicast e broadcast). Deve prevedere la connessione sincrona e asincrona. Il linguaggio deve contenere un ricco insieme di primitive da usare come sottostrato su cui costruire i linguaggi ad alto livello e i protocolli. Questi protocolli ad alto livello devono essere indipendenti dal meccanismo di trasporto usato (TCP/IP, email, http, ecc.).

- **ambiente**: l'ambiente in cui gli agenti software lavorano deve essere distribuito, eterogeneo e dinamico. Per fornire un canale di comunicazione con il mondo esterno all'ambiente, il linguaggio di comunicazione deve fornire strumenti per far fronte a dinamismo ed eterogeneità; deve supportare interoperabilità con altri linguaggi e protocolli, e la scoperta di conoscenza entro ampie reti. In fine, deve essere facilmente integrabile con sistemi pre-esistenti.

- **affidabilità**: un ACL deve supportare la comunicazione tra agenti in modo che sia affidabile e sicura. Ci deve essere un modo di garantire l'autenticazione degli agenti. Il linguaggio di comunicazione deve essere robusto rispetto a messaggi inappropriati o malformati e deve supportare meccanismi di ragionamento per identificare e segnalare errori e avvertimenti.

Un concetto base nell'ambito degli ACL è lo *speech-act (atto comunicativo)*, detto anche *performative*. Il termine performative è usato per indicare il tipo primitivo di un messaggio. Nella teoria degli speech-act [5, 66] una performative è un'affermazione che ha credito semplicemente perché l'"emittente" la dice o la sostiene.

Due ACL largamente usati e studiati sono Knowledge Query and Manipulation Language (KQML) e Foundation for Intelligent Physical Agents Agent Communication Language (FIPA ACL). Ognuno di essi offre un insieme minimale di performative per descrivere le azioni dell'agente e permettere agli utenti di estendere tale insieme in modo che le regole sintattiche e semantiche dell'ACL vengano rispettate.

## 3.1 KQML

Un linguaggio di comunicazione per agenti progettato per l'interazione tra agenti intelligenti software è Knowledge Query and Manipulation Language (KQML) [51, 24, 23]. Questo è stato sviluppato dal consorzio Knowledge Sharing Effort sponsorizzato da ARPA (ARPA KSE, o KSE per brevità) e implementato da diversi gruppi di ricerca. È stato usato per implementare con successo vari sistemi informativi basati su diverse architetture software.

### 3.1.1 The Knowledge Sharing Effort

L'ARPA Knowledge Sharing Effort (KSE) è un consorzio per sviluppare convenzioni che facilitino condivisione e riuso delle basi di conoscenza e dei sistemi basati sulla conoscenza. Il suo scopo è definire, sviluppare e controllare infrastrutture e tecnologie di supporto, per consentire ai partecipanti di costruire sistemi più grandi, con maggiori funzionalità di quelle che potrebbero portare a compimento lavorando da soli. Il KSE è organizzato in quattro gruppi di lavoro, ognuno dei quali si occupa di un problema complementare, quali:

- *Interlingua Group*: si occupa dello sviluppo di un linguaggio comune per esprimere il contenuto di una base di conoscenza. Questo gruppo ha pubblicato un documento di specifica che descrive il Knowledge Interchange Formalism (KIF), basato sulla logica del prim'ordine estesa per supportare definizioni e ragionamento non-monotoni. KIF fornisce una specifica per la sintassi e la semantica di un linguaggio. KIF può essere usato per supportare la traduzione da un linguaggio per il contenuto ad un altro, o come un comune linguaggio per il contenuto tra due agenti che usano diversi linguaggi di rappresentazione nativi. Informazioni su KIF e strumenti associati sono disponibili su `http://www.cs.umbc.edu/kse/kif/`.

- *Knowledge Representation System Specification Group (KRSS)*: focalizza il proprio lavoro sulla definizione di costrutti comuni entro famiglie di linguaggi di rappresentazione. Documenti e informazioni relative a questo gruppo si trovano su `http://www.cs.umbc.edu/kse/krss/`.

- *Shared, Reusable Knowledge Bases Group (SRKB)*: la sua attività riguarda il contenuto di basi di conoscenza condivise, si interessa alla conoscenza condivisa per particolari aree tematiche e allo sviluppo di strumenti e metodologie indipendenti dalla tematica. Ha costruito un repository per ontologie e strumenti condivisi che è disponibile su Internet su `http://www.cs.umbc.edu/kse/srkb/`.

- *External Interfaces Group*: ha come scopo l'interazione a run-time tra sistemi basati sulla conoscenza e altri moduli. KQML è uno principali

risultati ottenuti dall'External Interfaces Group del KSE. Informazioni generali sono disponibili su `http://www.cs.umbc.edu/kqml/`.

### 3.1.2 Il linguaggio KQML

KQML è un linguaggio di comunicazione ad alto livello orientato al messaggio e un protocollo per lo scambio di informazioni indipendente dal contenuto sintattico e dall'ontologia applicabile. È un linguaggio indipendente dal meccanismo di trasporto (TCP/IP, SMTP, IIOP, o altro), indipendente dal linguaggio per il contenuto (KIF, SQL, STEP, Prolog, o altro), e indipendente dall'ontologia considerata nel contenuto.

La comunicazione si colloca su tre livelli:

- *livello contenuto*: porta il contenuto reale del messaggio. Il contenuto del messaggio può essere espresso con un qualsiasi linguaggio di rappresentazione e scritto in stringhe ASCII o in notazione binaria. Tutte le implementazioni KQML ignorano il contenuto del messaggio eccetto la porzione relativa all'estensione necessaria per riconoscere inizio e fine dello stesso.

- *livello comunicazione*: codifica nel messaggio un insieme di proprietà che descrivono i parametri della comunicazione al livello più basso, come le identità di mittente e destinatario e un unico identificatore associato alla comunicazione.

- *livello messaggio*: codifica il messaggio che un'applicazione vuole mandare ad altri ed è il cuore di KQML. A questo livello si determina il tipo di interazione che si può avere con un agente che parla KQML. La prima funzione di questo livello è identificare il protocollo di rete con cui consegnare il messaggio e fornire una performative (o speech-act) che il mittente possa allegare al contenuto. Questo speech-act indica se il messaggio è un'asserzione, una query, un comando o un'altra performative presa da un insieme fissato. Inoltre, poiché il contenuto del messaggio non è visibile a KQML, il livello messaggio si occupa di inserire proprietà opzionali che descrivono il linguaggio per il contenuto, l'ontologia assunta, e varie descrizioni più generali come il descrittore di una tematica fissata entro l'ontologia. Queste funzionalità rendono possibile per l'ambiente supportante analizzare e consegnare messaggi basandosi sul loro contenuto anche quando il contenuto stesso non è accessibile.

La sintassi di KQML è basata su una lista di parentesi bilanciate. L'elemento iniziale della lista è la *performative* e gli elementi restanti sono gli argomenti della performative forniti come coppie (parola-chiave, valore). Il linguaggio è piuttosto semplice e per questo la sintassi attuale potrebbe essere modificata in futuro, se necessario.

Ci si aspetta che KQML sia supportato da un sottostrato software che renda possibile la localizzazione di un agente, in ambiente distribuito, da parte di ogni altro agente. Molte delle implementazioni correnti prevedono questi ambienti e sono solitamente basate su programmi aiutanti detti *router* o *facilitator*. Gli ambienti non sono una parte specificata di KQML, non sono standardizzati ma molti degli ambienti KQML correnti sono evoluti in modo da usare alcuni framework quali OMG's Corba o Microsoft's OLE2.

Concettualmente, un messaggio KQML consiste di una performative, degli argomenti ad esso associati, costituenti il vero contenuto del messaggio, e di un insieme di argomenti opzionali *transport*, descriventi il contenuto e probabilmente il mittente e il destinatario.

L'insieme predefinito di performative di KQML non è né minimale né chiuso. Un agente deve scegliere solo alcune performative da gestire, ma questo insieme può essere esteso in modo che una società di agenti possa usare tali performative addizionali per andar incontro all'interpretazione e al protocollo associati con essa. Tuttavia, un'implementazione che scelga di implementare una delle performative riservate dovrà farlo in modo standard.

Le performative riservate sono indicate di seguito:

**Basic query performatives**
*evaluate, ask-if, ask-in, ask-one, ask-all, ...*

**Multi-reponse query performatives**
*stream-in, stream-all, ...*

**Reponse performative**
*reply, sorry, ...*

**Generic informational performatives**
*tell, achieve, cancel, untell, unachieve, ...*

**Generator performatives**
*standby, ready, next, rest, discard, generator, ...*

**Capability-definition performatives**
*advertise, subscribe, monitor, import, export, ...*

**Networking performatives**
*register, unregister, forward, broadcast, route, ...*

Di seguito l'indicazione del significato di alcune performative:

- *ask-one*, con una query entro il contenuto, è una richiesta di risposta singola all'interrogazione espressa nella query, scritta nel linguaggio indicato e interpretata secondo l'ontologia data, diretta al ricevente idententificato per tale messaggio.

- *ask-all* è una richiesta analoga a quella descritta per ask-one ma fatta per ottenere un insieme di risposte.

- *stream-all* è una richiesta analoga a quella descritta per ask-all ma fatta per ottenere come risposta un insieme.

- *standby* è una performative che consente di esercitare il controllo sulle risposte ottenute con un'altra performative indicata come suo contenuto.

Oltre le performative di comunicazione standard come ask, tell, deny, delete, e altre performative orientate al protocollo, KQML contiene performative relative ad aspetti estranei al protocollo, quali *advertise*, che consente ad un agente di annunciare quale tipo di messaggio asincrono è pronto a gestire, e *recruit*, che può essere usata per trovare agenti disponibili per particolari tipi di messaggi.

### Architettura software di KQML

KQML non è stato definito da un singolo gruppo di ricerca ma da una commissione di rappresentanti di diversi progetti, tutti volti alla gestione dell'implementazione di sistemi distribuiti. Questi gruppi avevano in comune l'obiettivo di gestire una collezione di processi che possano cooperare e di semplificare i requisiti di programmazione per implementare un sistema di questo tipo ma non condividevano alcuna architettura di comunicazione comune. Come risultato di tale situazione, KQML non detta un'architettura di sistema particolare.

Ci sono due implementazioni di KQML, scritte una in Common Lisp, l'altra in C. Entrambe sono pienamente interoperabili e sono frequentemente usate insieme. La progettazione di queste implementazioni è motivata dalla necessità di integrare una varietà di sistemi esperti preesistenti in un gruppo di processi collaborativi in quanto molti dei sistemi coinvolti non sono progettati per operare in un ambiente orientato alla comunicazione. Il progetto è costruito intorno a due programmi specializzati, un *router* e un *facilitator*, e a una libreria di interfacce, detta *KRIL*.

### KQML Router

I *router* sono instradatori di messaggi indipendenti dal contenuto. Ogni agente software KQML comunicante è associato ad un proprio processo router separato. Tutti i router sono identici; ognuno è una copia in esecuzione dello stesso programma. Un router gestisce tutti i messaggi KQML che arrivano da e vanno a l'agente ad esso associato. Poiché ogni programma ha associato un router, non è necessario fare cambiamenti per estendere l'organizzazione interna di ogni programma per permettergli di ricevere messaggi da varie sorgenti indipendenti in modo asincrono. Il router fornisce tale servizio per l'agente e fa sì che l'agente abbia un singolo punto di contatto

con la rete. Fornisce entrambe le funzioni client e server per l'applicazione e gestisce connessioni con altri agenti multiple e simultanee.

Il router non guarda mai il campo contenuto del messaggio che manipola ma fa affidamento sulle performative KQML e sugli argomenti di tale messaggio. Se un messaggio KQML uscente specifica un particolare indirizzo Internet, il router dirige il messaggio ad esso; se il messaggio specifica un particolare servizio, il router tenta di trovare un indirizzo Internet per tale servizio e gli consegna il messaggio; se il messaggio fornisce solo una descrizione del contenuto il router tenta di trovare un server che possa andar bene per il messaggio e glielo manda, o può scegliere di mandarlo a un agente di comunicazione più pronto che sia capace di indirizzarlo. I router possono essere implementati in maniera più o meno sofisticata, il che determina il fatto che non possono garantire la consegna di tutti i messaggi.

### KQML Facilitator

Per consegnare messaggi che non sono completamente indirizzati, i router si affidano ai *facilitator*. Un facilitator è un'applicazione di rete che fornisce servizi di rete utili. Esso tiene un registro di nomi dei servizi e manda messaggi ai servizi individuati secondo le richieste. I facilitator sono agenti software reali che hanno il loro proprio router KQML per gestire il traffico e distribuire i messaggi KQML. Tipicamente c'è un facilitator per ogni gruppo locale (sito o progetto) di agenti.

Quando ogni applicazione parte, il suo router la notifica al facilitator locale in modo che questo la registri nel database locale. Quando l'applicazione si arresta, il router manda un altro messaggio KQML al facilitator in modo che l'applicazione venga rimossa dal database del facilitator. In questo modo un'applicazione può trovare ogni altra applicazione senza dover tenere una lista di servizi locali.

### KQML KRIL

Il router è un processo separato dalle applicazioni e per questo è necessario avere un programma interfaccia tra l'applicazione e il router. Questa interfaccia (API) è detta KRIL (KQML Router Interface Library). La KRIL API è incapsulata nell'applicazione e ha accesso agli strumenti dell'applicazione per analizzare il contenuto. Ci sono varie KRIL, una per ogni tipo di applicazione e una per ogni linguaggio di applicazione. L'obiettivo principale della KRIL è rendere l'accesso al router il più semplice possibile per il programmatore. A tal fine, una KRIL può essere incapsulata strettamente nell'applicazione o nel linguaggio dell'applicazione, se si preferisce, ma è necessario incapsulare la KRIL completamente nel linguaggio di programmazione dell'applicazione.

Una semplice KRIL fornisce due funzioni. Per iniziare una transazione

c'è una funzione **send-kqml-message** che accetta il contenuto di un messaggio e tante informazioni circa il messaggio e la sua destinazione quante ne possono essere fornite e restituisce la risposta all'agente remoto o un semplice codice per indicare che il messaggio è stato mandato. Per gestire messaggi asincroni in arrivo, si ha una funzione **declare-message-handler** che permette al programmatore dell'applicazione di dichiarare quali funzioni invocare quando arriva un messaggio. A seconda delle proprietà delle KRIL, il messaggio entrante può essere ordinato in base a performative, o tematica, o altre caratteristiche, e indirizzato verso diverse funzioni di gestione dei messaggi differenti.

Le KRIL accettano diversi tipi di dichiarazioni che consentono loro di registrare le loro applicazioni con facilitator locali e di contattare agenti remoti per avvisarli che sono interessati a ricevere messaggi da loro. Sono state implementate diverse KRIL sperimentali per Common Lisp, C, Prolog, Mosaic, SQL, e altri linguaggi.

### 3.1.3 Semantica di KQML

Esiste solo una descrizione informale e parziale della semantica di KQML [43] anche se si stanno facendo molteplici sforzi per fornire una semantica formale.

Per ogni performative di KQML è data una semantica in termini di *precondizioni*, *postcondizioni*, e *condizioni di completamento*.

Assumendo di avere un mittente A e un ricevente B, le precondizioni indicano lo stato necessario perché un agente mandi una performative, **Pre(A)**, e perché il ricevente l'accetti e la processi con successo, **Pre(B)**. Se la precondizione non è soddisfatta, le risposte più probabili sono *error* e *sorry*.

Le postcondizioni descrivono lo stato del mittente dopo l'asserzione di una performative con successo, e del ricevente dopo aver ricevuto e processato un messaggio ma prima di una contro asserzione. Le postcondizioni **Post(A)** e **Post(B)** sono soddisfatte a meno che un *sorry*, o un *error*, sia mandato come una risposta per riportare l'esecuzione di messaggi con insuccesso.

Una condizione di completamento per la performative, **Completion**, indica lo stato finale.

Stabilire precondizioni per una performative non garantisce una sua esecuzione di successo. Le precondizioni indicano solo cosa può essere assunto come stato degli interlocutori coinvolti nello scambio. Le postcondizioni descrivono gli stati degli interlocutori assumendo che la primitiva di comunicazione sia portata a compimento con successo. Precondizioni, postcondizioni e condizioni di completamento descrivono gli stati degli agenti in un linguaggio per attitudini mentali (credenza, conoscenza, desiderio e intenzioni) e per descrittori di azioni (per mandare e processare un messaggio). Non sono forniti modelli semantici per le attitudini mentali, ma il linguag-

gio usato per descrivere gli stati degli agenti restringe notevolmente il modo in cui le attitudini mentali possono essere combinate per comporre gli stati degli agenti.

Un altro approccio semantico si basa su un lavoro precedente fondato sulla capacità di agire razionalmente propria degli agenti. Il nucleo di tale lavoro considera il termine *performative* inappropriato per descrivere primitive di comunicazione di KQML; l'approccio suggerito vede i tipi dei messaggi riservati del linguaggio come tentativi di comunicazione. Questi tentativi coinvolgono due o più agenti razionali che temporaneamente formano gruppi da impegnare nella comunicazione. Questo approccio lega la semantica ACL alla teoria per agenti su cui si assume che gli agenti coinvolti nello scambio con ACL siano basati.

### 3.1.4 Riferimenti

Informazioni su KQML, incluse pubblicazioni, specifiche di linguaggi, accesso alle API, liste di discussione, e altro si possono trovare alla pagina `http://www.cs.umbc.edu/kqml/`.

## 3.2 FIPA ACL

Il FIPA ACL è basato, come KQML, sulla teoria degli speech-act: i messaggi sono azioni o atti comunicativi [43].

### 3.2.1 FIPA

La Foundation for Intelligent Physical Agents [26] è un'associazione nonprofit il cui scopo è promuovere il successo di applicazioni, servizi, e attrezzature emergenti basate sugli agenti. L'obiettivo di FIPA è rendere disponibili specifiche che massimizzino l'interoperabilità attraverso i sistemi basati sugli agenti. FIPA è un'organizzazione per standard nell'area degli agenti software. L'organizzazione inizialmente prevedeva nel suo nome la parola *physical* per indicare gli agenti della specie robotica. Nel tempo la presenza di questo aggettivo è servita per ricordare che gli agenti fisici, cioè umani, e l'interazione con essi sono parte dello scopo dell'associazione.

FIPA opera mediante la collaborazione internazionale e aperta di organizzazioni membro, che sono compagnie e università attive nel campo. Compagnie che si occupano di tecnologia in Europa ed Estremo-Oriente sono state tra le più attive e sollecite partecipanti, tra cui Alcatel, British Telecom, France Telecom, Deutsche Telecom, Hitatchi, NEC, NHK, NTT, Nortel, Siemens e Telia.

Le opearazioni di FIPA sono incentrate su insiemi di specifiche rilasciate annualmente. La specifica corrente è *FIPA 2000*, disponibile alla home page di FIPA, `www.fipa.org`. FIPA assegna compiti a commissioni tecniche,

ognuna delle quali ha come principale responabilità la produzione, il mantenimento e l'aggiornamento delle specifiche applicabili ai propri compiti.

La commissione tecnica più importante è quella incaricata di produrre la specifica per un ACL. Inoltre, la commissione per la gestione degli agenti copre servizi per agenti quali facilitazioni, registrazione e piattaforme per agenti; la commissione per l'interazione tra agenti e software si occupa di integrazione di agenti con applicazioni software pre-esistenti. Insieme queste tre commissioni creano la spina dorsale delle specifiche FIPA.

### 3.2.2 Concetti di FIPA ACL

La specifica di FIPA ACL consiste di un insieme di tipi di messaggi e della descrizione della loro pragmatica, cioè gli effetti sulle attitudini mentali degli agenti mittente e destinatario. La specifica descrive ogni atto comunicativo con una forma narrativa e una semantica formale basata sulla logica modale. Fornisce anche la descrizione normativa di un insieme di protocolli di interazione ad alto livello, includendo il richiedere un'azione, il contratto con la rete e svariati tipi di aste.

FIPA ACL è superficialmente simile a KQML. La sua sintassi è identica a quella di KQML eccetto per i nomi diversi delle primitive riservate. Esso mantiene l'approccio di KQML secondo cui si separa il linguaggio "esterno" da quello del contenuto. Il linguaggio esterno definisce il significato desiderato del messaggio; il linguaggio per il contenuto denota l'espressione a cui si riferiscono belief, desire e intention degli interlocutori, secondo quanto descritto dal significato della primitiva di comunicazione.

KQML è stato criticato per l'uso del termine *performative* in riferimento alle primitive di comunicazione. In FIPA ACL, le primitive di comunicazione sono dette *atti di comunicazione (communication acts)*, o CAs per brevità. Nonostante la differenza nel modo di chiamarle, le performative KQML e i communicative acts FIPA ACL sono lo stesso tipo di entità. Per evitare confusione, si usano i termini performative, primitiva (di comunicazione) e atto di comunicazione in modo intercambiabile.

Il documento di specifica di FIPA ACL impone che non ci sia nessun vincolo sull'uso di un particolare linguaggio per il contenuto per FIPA ACL (come per KQML). Comunque, per comprendere e processare alcune primitive FIPA ACL, gli agenti riceventi devono avere conoscenza del Semantic Language (SL).

### 3.2.3 Semantica di FIPA ACL

SL è il linguaggio formale usato per definire la semantica di FIPA ACL. SL è una logica multimodale, quantificata con operatori modali per i belief ($B$), i desideri ($D$), gli uncertain belief ($U$), e le intenzioni (persistent goal, $PS$). SL può rappresentare proposizioni, oggetti e azioni. SL trae origine da un

lavoro di P. R. Cohen e H. J. Levesque, ma la sua forma corrente è basata principalmente sul lavoro di M. D. Sadek. Una specifica dettagliata di SL, inclusa la sua semantica, si può trovare all'interno della specifica di FIPA ACL.

In FIPA ACL, la semantica di ogni atto di comunicazione è specificata mediante un insieme di formule SL che descrivono la *precondizione di fattibilità (feasibility precondition)* dell'atto e il suo *effetto razionale (rational effect)*, RE. Per un CA $a$ dato, la precondizione di fattibilità FP($a$) descrive la condizione necessaria per il mittente del CA. Quindi, perché un agente possa portare a compimento l'atto comunicativo $a$ mandando un messaggio particolare, la precondizione di fattibilità deve essere soddisfatta per il mittente. L'agente non è obbligato a compiere $a$ se FP($a$) è soddisfatta, ma lo può fare se vuole. Un effetto razionale per un atto comunicativo rappresenta l'effetto che un agente si può aspettare come risultato del compimento dell'azione; esso tipicamente specifica anche condizioni che dovrebbero essere vere per il ricevente. L'agente ricevente non è tenuto ad assicurare che si abbia l'effetto atteso e infatti potrebbe trovare che questo sia impossibile. In questo modo, un agente può usare la sua conoscenza degli effetti razionali per pianificare quale CA completare, ma non può assumere con certezza gli effetti razionali che ne seguiranno.

Essere conforme al FIPA ACL significa che quando un agente A manda un CA $x$, la FP($x$) deve essere soddisfatta. Il fatto che RE($x$) non sia garantito non è rilevante per la questione della conformità.

### 3.2.4 Riferimenti

Le specifiche messe a disposizione da FIPA sono disponibili nel FIPA Specification Repository, `http://www.fipa.org/repository/index.html`, e classificate in diversi insiemi.

Possono essere visionate secondo il loro stato nel ciclo di vita, il soggetto dell'area in cui ricadono, il loro tipo o l'anno in cui sono state messe a disposizione. Il repository può anche essere visto nella sua interezza.

Le specifiche per ACL si trovano nell'area *Agent Communication*, `http://www.fipa.org/repository/aclspecs.html`, ed hanno come argomento:

- i messaggi ACL. In dettaglio si considerano:
  - FIPA ACL Message Structure Specification alla pagina `http://www.fipa.org/specs/fipa00061/`;
  - FIPA Ontology Service Specification alla pagina `http://www.fipa.org/specs/fipa00086/`.
- i protocolli per lo scambio di messaggi ACL con le FIPA Interaction Protocols (IPs) Specification.

- le diverse espressioni per i messaggi ACL con la FIPA Communicative Act Library Specification alla pagina `http://www.fipa.org/specs/fipa00037/`.
- le differenti rappresentazioni di un messaggio ACL con:
  - FIPA SL Content Language Specification in `http://www.fipa.org/specs/fipa00008/`;
  - FIPA CCL Content Language Specification in `http://www.fipa.org/specs/fipa00009/`;
  - FIPA KIF Content Language Specification in `http://www.fipa.org/specs/fipa00010/`;
  - FIPA RDF Content Language Specification in `http://www.fipa.org/specs/fipa00011/`;

# Capitolo 4

# Coo-BDI: estensione del modello BDI con cooperatività

Nel primo capitolo si è descritto brevemente il modello BDI. La descrizione non è dettagliata e non affronta la questione relativa a cosa deve fare un agente se l'insieme dei piani applicabili è vuoto. Questa questione solitamente non è considerata nelle specifiche ad alto livello del modello BDI.
A. S. Rao e M. P. Georgeff nella loro specifica [63] scrivono che il generatore di istanze di piano, dopo aver letto la event queue e restituito una lista di istanze applicabili, sceglie da tale lista un insieme di istanze che vanno aggiunte alla struttura per le intenzioni ma non stabiliscono esplicitamene cosa accade se nessuna istanza può essere adottata. Quello che si intuisce per questo caso è che l'interprete salta direttamente al passo successivo.
M. d'Inverno, D. Kinny, M. Luck e M. Wooldrige forniscono una specifica più dettagliata [22] in cui, tuttavia, il caso "nessun piano applicabile trovato" non è considerato.

Quando si passa dalla specifica all'implementazione, questo problema non può più essere ignorato, ma i diversi sistemi lo affrontano in modo differente.

### I linguaggi di specifica e il problema

Segue una breve trattazione di come i linguaggi e i sistemi di specifica per agenti visti nel Capitolo 1 si comportano in relazione al problema"nessun piano applicabile trovato".

**dMars** nel caso in cui non si trovino piani applicabili per un dato evento ignora tale evento e lo rimuove dalla coda degli eventi [22].

**3APL** prevede delle regole, le *failure-rules*, che consentono all'agente, nel caso in cui un goal non sia soddisfatto in corrispondenza di una certa

situazione, di procedere in modo opportuno per trattare il fallimento [21].

**JACK** tratta due tipologie di eventi. Eventi comuni che corrispondono a informazioni transienti cui l'agente deve reagire, analogamente a quanto avviene nei programmi di tipo event-driven, ed eventi BDI che consentono all'agente di perseguire obiettivi a lungo termine e che ne influenzano il comportamento. Gli eventi comuni sono gestiti in modo tale che se non ci sono piani rilevanti per un evento, il task per esso fallisce e il sistema torna ad uno stato da cui può processare un nuovo evento. Per ogni piano rilevante trovato si controlla che sia applicabile nella circostanza corrente eseguendo un metodo `context()`. Se il metodo fallisce l'agente esamina il piano rilevante successivo. La specifica non è chiara su cosa succede se non ci sono piani applicabili. Si presume che si comporti come nel caso in cui non ci sono piani rilevanti. Gli eventi BDI sono tali che i piani per essi sono scelti seguendo delle euristiche e in caso di fallimento si agisce intelligentemente scegliendo piani alternativi o ricalcolando l'insieme dei piani tra cui scegliere escludendo quelli falliti [2].

**JAM** genera delle Applicable Plan Library (APL) entro cui scegliere il piano da eseguire. La scelta di questo piano è fatta con un meccanismo di ragionamento al metalivello. Se la APL è vuota si esamina quella del livello precedente. Se questa non è vuota si sceglie un piano da essa, lo si mette nella struttura delle intenzioni e si esgue un elemento in esso, altrimenti se è vuota si esegue un altro elemento nella struttura delle intenzioni. In pratica quando non è piu possibile ricondursi ad un livello precedente si passa ad eseguire un'altra intenzione [34].

**AgentTalk** è tale che se non si trova un piano per un evento, l'evento viene scartato [76].

**Dribble** procede in modo tale per cui se la goal rule non determina alcun piano non si segue nessuna operazione particolare e il goal non viene eliminato finché non è soddisfatto [73].

**ConGolog** per quanto si intuisce dalla documentazione studiata [49], non adotta nessuna strategia particolare nel caso in cui non si riesca a procedere con le transizioni a disposizione partendo da una data situazione.

**AGENT-0** secondo quanto suggerito dalla documentazione esaminata [49], si comporta in modo che una commitment rule è attivata dalla ricezione di un messaggio. Se per un messaggio in arrivo non è prevista alcuna commitment rule non si generano nuovi commitment.

**AgentSpeak(L)** nella sua specifica originale non descrive in modo chiaro come si comporta l'interprete se l'insieme delle istanze di piano applicabile è vuoto. Secondo una specifica di AgentSpeak(L) che segue un orientamento operazionale, descritta brevemente in [54], se non ci sono istanze di piani rilevanti l'evento va scartato. Diverse implementazioni di AgentSpeak(L), proposte da Bordini *et al.* [6, 8] prevedono che l'utente abbia a disposizione due opzioni possibili nel caso in cui non ci siano istanze di piani applicabili per un evento: l'utente può chiedere all'interprete di scartare l'evento o di tenerlo all'interno dell'insieme degli eventi in modo da poterlo eventualmente gestire se in un istante successivo si avesse un'istanza di piano applicabile a disposizione. Nell'implementazione descritta in [8], si prevede un meccanismo per gestire un certo tipo di evento per cui non ci sono piani rilevanti o applicabili. A fronte del fallimento si genera un nuovo evento corrispondente gestito mediante un piano definito appositamente entro l'agente.

Per i linguaggi esplicitamente basati sulle logiche BDI (quali dMARS, 3APL, JACK, JAM, AgentTalk e AgentSpeak(L)) è evidente che ci si trova di fronte a due comportamenti: o si scarta l'evento privo di piani applicabili corrispondenti o si ricorre a un piano di default o a un comportamento alternativo che rimandi nel tempo la gestione dell'evento; per gli altri pare che l'approccio sia quello di fermarsi e proseguire con altre esecuzioni, "rassegnandosi" di fronte al fallimento.

Questo evidenzia come il problema "nessun piano applicabile" sia risolto solitamente ad hoc. Un'approccio al problema che tenta di risolvere parzialmente la questione in un modo più generale e che si fonda sulla capacità di cooperare degli agenti è quello che introduce il modello Cooperative BDI.

## 4.1 Panoramica su Coo-BDI

L'*estensione Cooperative BDI (Coo-BDI) per il modello BDI*, proposta da D. Ancona e V. Mascardi [3], promuove la *cooperazione*, cioé la capacità di un agente di aiutare altri agenti a soddisfare i loro desideri. In Coo-BDI la cooperazione consente agli agenti di scambiare i piani risolvendo parzialmente il problema "nessuna istanza di piano applicabile".

Coo-BDI si basa sulla specifica di dMARS [22]. Come prima estensione Coo-BDI prevede che eventi esterni e desideri principali siano tenuti separati. A tal fine ci sono due strutture: la "event queue" che contiene solo gli eventi esterni e il "desire set" che contiene solo desideri di tipo *achieve* generati dagli eventi. Si distingue tra *desideri principali (main desires)* che sono tenuti nel desire set, e *desideri subalterni (subaltern desires)* che sono generati mentre si tenta di soddisfare un desiderio principale e restano impliciti nella struttura stack per le intenzioni. Quando un desiderio principale

non viene soddisfatto si torna in dietro (si esegue il backtracking) e si sceglie per esso un'istanza di piano non ancora provata. Se non ci sono piani non ancora provati per esso, il desiderio principale è rimosso dall'insieme. Il backtracking per i desideri sulbarterni non viene effettuato, sia per mantenere le strutture dati e l'interprete Coo-BDI semplici, sia per seguire la stessa strategia adottata da dMARS.

La principale estensione di Coo-BDI, tuttavia, prevede

- l'introduzione della *cooperazione* tra agenti al fine di recuperare piani esterni per soddisfare desideri (principali e subalterni);
- l'estensione di piani con *specificatori d'accesso*;
- l'introduzione dei piani di *default*;
- l'estensione delle *intenzioni* per tener conto del meccanismo di recupero delle istanze di piano esterne; e
- la modifica del Coo-BDI *engine* (interprete) per far fronte a tutti questi problemi.

*Strategia di cooperazione.* La strategia di cooperazione, o più semplicemente cooperazione, di un agente $A$ comprende l'insieme degli agenti con cui si aspetta di cooperare, la politica di recupero dei piani e la politica di acquisizione dei piani. La stategia di cooperazione può evolvere durante il tempo garantendo massima flessibilità e autonomia agli agenti.

*Piani.* I piani Coo-BDI sono classificati in *specifici* e *di default*. Come i piani BDI, entrambi i tipi di piano sono formati da un trigger, una precondizione, un corpo, un'invariante e due insiemi di azioni di successo e di fallimento. Oltre a queste componenti, tali piani hanno anche uno *specificatore d'accesso (access specifier)* che determina l'insieme degli agenti con cui il piano può essere condiviso. Esso può assumere due valori: *private*, il piano non può essere condiviso, *public*, il piano può essere condiviso con qualunque agente e *only(TrustedAgents)*, il piano può essere condiviso solo con gli agenti contenuti nell'insieme *TrustedAgents*. Un *piano di default* è un piano in cui lo specificatore d'accesso è *private*, il trigger è una variabile e la precondizione è la costante *true*; inoltre, per definizione, un piano di default può essere sempre applicabile e non può mai essere scambiato. Ogni agente deve sempre fornire almeno un piano di default in modo che ogni desiderio possa essere gestito. Un *piano specifico* è un piano non di default.

*Intenzioni.* Le intenzioni Coo-BDI sono in relazione uno-a-uno con i desideri principali: ogni intenzione è creata quando un nuovo desiderio principale entra nel desire set, ed è cancellata quando il desiderio principale fallisce o è soddisfatto. Le intenzioni sono caratterizzate dalle componenti

"standard" più le componenti introdotte per gestire il meccanismo di recupero dei piani esterni. I piani esterni sono recuperati, secondo la politica di recupero sia per i desideri principali sia per quelli subalterni.

*Interprete Coo-BDI.* L'interprete Coo-BDI differisce da quello classico per tenere in considerazione sia la generazione di desideri sia la cooperazione. È caratterizzato da tre passi:

1. processare la event queue;
2. processare le intenzioni sospese;
3. processare le intenzioni attive.

Prima di descrivere questi passi viene spiegato il meccanismo per il recupero dei piani rilevanti che è indispensabile per capire i passi 1 e 3. Tale meccanismo consiste di quattro passi sequenziali. a) L'intenzione viene sospesa. b) Si generano le istanze locali di piani rilevanti per il desiderio e le si associano all'intenzione. c) Secondo la cooperazione, si definisce l'insieme $S$ degli agenti con cui ci si aspetta di cooperare. d) Si crea una richiesta di piano per il desiderio e la si manda a tutti gli agenti in $S$.

Gli eventi nella coda possono essere eventi sia di *cooperazione*, sia *ordinari*. Gli eventi di cooperazione includono richieste di istanze di piani rilevanti per un desiderio e risposte a richieste di piano. Gli eventi ordinari comprendono, almeno, ricezione di messaggi e notifica di aggiornamenti fatti all'insieme dei belief dell'agente.
Quando un agente riceve una richiesta da un altro agente $A$, esso manda ad $A$ l'insieme, anche vuoto, di tutte le sue istanze di piani locali che sono rilevanti per il desiderio e visibili ad $A$ (i piani di default non sono visibili). Quando un agente riceve una risposta ad una richiesta di piano per un desiderio, controlla se la risposta è ancora valida e in caso affermativo aggiorna l'intenzione associata al desiderio in modo che includa le istanze di piano appena ottenute e memorizzi la risposta. Infine, se l'evento è ordinario, l'insieme dei desideri corrispondenti è generato e aggiunto al desire set. Per ogni nuovo desiderio si crea un'intenzione vuota e il meccanismo per il recupero dei piani rilevanti è avviato.

La gestione delle intenzioni sospese consiste nel guardare se ci sono intenzioni sospese che possono essere ripristinate. Quando un'intenzione è ripristinata si genera l'insieme delle istanze di piano applicabili dall'insieme delle istanze di piano rilevanti escluse le istanze di piano già fallite, si sceglie un'istanza di piano applicabile e si crea l'istanza di piano corrispondente da mandare in esecuzione. Se l'insieme delle istanze di piano applicabili è vuoto, il desiderio fallisce, è cancellato dall'insieme dei desideri e l'intenzione è distrutta. Altrimenti, l'istanza di piano applicabile scelta è messa sullo stack dell'intenzione. Se il piano scelto è uno di quelli recuperati all'esterno, può essere usato e scartato, aggiunto alla libreria dei piani o usato

per rimpiazzare i piani con un trigger unificante, a seconda della politica di acquisizione.

Infine, le intenzioni attive sono gestite come nell'architettura BDI eccetto per il meccanismo di recupero dei piani rilevanti triggerati per il soddisfacimento del desiderio.

## 4.2 Specifica strutturale di Coo-BDI

La descizione delle strutture presenti in Coo-BDI è data mediante una BNF all'interno della quale si usano i non-terminali definiti informalmente come segue:

- `Variable`: un simbolo di variabile
- `Pred:` un simbolo di predicato
- `Term:` un termine
- `GroundTerm:` un termine privo di variabili (ground)
- `AgentId:` una stringa che rappresenta l'identità dell'agente
- `RequestId:` una stringa che identifica una richiesta
- `IntentionId:` una stringa che identifica un'intenzione
- `Message:` ogni tipo di termine eventualmente contenente variabili libere
- `Substitution:` una corrispondenza tra variabili e termini
- Ogni non-terminale NT seguito dalla parola Set rappresenta un insieme di elementi in NT
- Ogni non-terminale NT seguito dalla parola Sequence rappresenta una sequenza di elementi in NT
- Ogni non-terminale NT seguito dalla parola Queue rappresenta una coda di elementi in NT
- Ogni non-terminale NT seguito dalla parola Stack rappresenta uno stack di elementi in NT.

Le sole differenze per gli ultimi quattro elementi consistono nelle primitive che saranno usate.

*Belief, desire, query e action.* Una belief formula è un atomo o un

atomo negato. I belief sono belief formula ground. I desire sono della forma `achieve(BeliefFormula)`. Le query sono denotate da `query(SituationFormula)` dove una situation formula è una belief formula, o una costante `true` o `false`, o una congiunzione o una disgiunzione di situation formula.

Le azioni possono essere interne o esterne. Le azioni interne sono aggiornamenti dei belief dell'agente e possono essere eseguite solo se il loro argomento è ground. Le azioni esterne includono almeno la capacità di mandare messaggi agli agenti.

```
BeliefFormula ::= Pred(Term, ...., Term) | not(Pred(Term, ...., Term))
Belief ::= Pred(GroundTerm, ...., GroundTerm) |
  not(Pred(GroundTerm, ...., GroundTerm))
SituationFormula ::= true | false | BeliefFormula |
  SituationFormula and SituationFormula |
  SituationFormula or SituationFormula
Desire ::= achieve(BeliefFormula)
Query  ::= query(SituationFormula)
InternalAction ::= add(BeliefFormula) | remove(BeliefFormula)
ExternalAction ::= send(AgentId, Message)
Action ::= InternalAction | ExternalAction
```

*Plan.* Sono definiti da uno specificatore d'accesso, un trigger, una precondizione, un corpo, un'invariante e due insiemi di azioni interne usati quando il piano fallisce o ha successo. Sintatticamente non ci sono differenze tra piani specifici e piani di default. Gli specificatori d'accesso possono assumere i valori seguenti:
– `private` quando il piano non può essere fornito;
– `public` quando il piano può essere fornito ad un agente qualsiasi;
– `only(TrustedAgents)` dove `TrustedAgents` è l'insieme degli identificatori degli agenti che indicano i soli agenti fidati che possono ricevere istanze del piano.

Il trigger di un piano specifico è il desiderio che il piano deve soddisfare. Precondizioni e invarianti sono situation formula. I body dei piani sono alberi non vuoti i cui nodi sono stati d'esecuzione e i cui archi sono etichettati da desire, o query, o action interne o esterne. Le azioni di successo e fallimento sono sequenze di azioni interne.

```
AccessSpecifier ::= private | public | only(AgentIdSet)
EdgeLabel ::= Desire | Query | Action
Body ::= state(EdgeBodySequence)
EdgeBody ::= (EdgeLabel, Body)
Plan ::= plan(AccessSpecifier, Desire, SituationFormula, Body,
SituationFormula, InternalActionSequence, InternalActionSequence)
```

*Plan instance.* Un'istanza di piano è una coppia `(Plan, Substitution)` formata da un piano e una sostituzione. Un'istanza di piano è detta *rilevante*

con riferimento a un desire se `Substitution` è l'unificatore più generale per tale desire e il trigger di `Plan`. Un'istanza di piano è detta *applicabile* se la formula ottenuta applicando `Substitution` alla precondizione di `Plan` è una conseguenza logica dei belief dell'agente. Si noti che le istanze di piano formate mediante un piano di default sono sempre rilevanti e applicabili.

L'esecuzione di un'istanza di piano è definita dall'istanza di piano con la sostituzione calcolata, con lo stato corrente del corpo del piano e l'insieme dei restanti fratelli dello stato corrente che non sono ancora stati eseguiti.

*Intention e request.* Un'intenzione è composta da un identificatore unico, uno stack per l'esecuzione delle istanze di piano, uno stato, un insieme di istanze di piano rilevanti, un insieme di identificatori di agente e un insieme di istanze di piano fallite. L'identificatore dell'intenzione è usato per modellare in modo conveniente le due relazioni `DesireIntention` e `IntentionRequest` che associano rispettivamente ogni desiderio principale con esattamente un'intenzione corrispondente e ogni intenzione sospesa con la corrispondente richiesta di istanze di piano rilevanti.

Lo stack di esecuzione delle istanze di piano è simile allo stack di esecuzione dei programmi logici. Lo stato può essere `suspended` o `active`. Un'intenzione è sospesa se l'esecuzione dell'istanza di piano al top dello stack necessita della soddisfazione di un desiderio per cui non sono ancora state scelte istanze di piano; in questo caso l'insieme delle istanze rilevanti contiene le istanze di piano rilevanti che sono già state collezionate per il desiderio e l'insieme degli identificatori d'agente contiene tutti gli identificatori di quegli agenti che ci si aspetta che cooperino per soddisfare il desiderio.

Una richiesta di collaborazione è specificata da un identificatore unico, dall'identificatore dell'agente richiedente e dal desiderio da soddisfare.

Una relazione `relationIntentionRequest` associa qualsiasi identificatore di intenzione sospesa con la sua richiesta di cooperazione corrente (se c'è) e qualsiasi richiesta con l'intenzione sospesa che l'ha originata, se la richiesta è ancora valida.

*Event.* Ci sono due tipi di eventi: eventi di cooperazione e eventi ordinari. Un evento di cooperazione è del tipo `requested(request(RequestedId, ReqAgentId, Desire))` secondo cui l'agente identificato da `ReqAgentId` sta richiedendo un'istanza di piano rilevante per `Desire`, o `provide(AgentId, request(RequestedId, ReqAgentId, Desire), Instance)` per cui l'agente identificato da `AgentId` ha cooperato rispondendo alla richiesta `request( RequestedId, ReqAgentId, Desire)` fornendo un insieme `Instances` di istanze di piano rilevanti per `Desire`. Gli eventi ordinari includono almeno quelli del tipo:
– `received(AgentId, Message)` per cui `Message` è il messaggio ricevuto da parte dell'agente identificato da `AgentId`;
– `added(Belief)` secondo cui il nuovo belief `Belief` è stato aggiunto alla

base di conoscenza dell'agente;
– `removed(Belief)` secondo cui il belief `Belief` è stato rimosso dalla base di conoscenza dell'agente.

Gli eventi percepiti dall'agente sono messi in una coda di priorità indicata come `eventQueue(EventQueue)`.

*Agent Definition.* Un agente è definito mediante predicati asseriti nella sua stessa base di conoscenza:

- `agentId(AgentId)` che specifica l'identificatore unico dell'agente;
- `eventQueue(EventQueue)` che indica la event queue corrente dell'agente;
- `isDesire(Desire)` che specifica l'insieme corrente dei desideri principali dell'agente;
- `isPlan(Plan)` che specifica l'insieme corrente dei piani (di default e specifici) dell'agente;
- `isIntention(Intention)` specifica l'insieme corrente delle intenzioni dell'agente;
- i belief correnti dell'agente;
- tre predicati che specificano la *cooperazione* corrente per l'agente:
  - `trustedAgents(TrustedAgents)` che indica l'insieme degli identificatori di agenti attualmente fidati per l'agente;
  - `retrievalPolicy(Retrieval)` che specifica la politica di recupero attuale, dove `Retrieval ::= always | noLocal`;
  - `acquisitionPolicy(Acquisition)` che specifica la politica corrente di acquisizione dei piani, dove `Acquisition ::= discard | add | replace`.
- `relationDesireIntention(Desire, IntentionId)` che specifica una relazione uno-a-uno tra il desiderio principale corrente per l'agente e gli identificatori di tutte le sue attuali intenzioni;
- `relationIntentionRequest(IntentionId, Request)` che specifica una relazione uno-a-uno tra gli identificatori di alcune delle intenzioni dell'agente attualmente sospese e alcune richieste emesse dall'agente;
- `canResume(Request, AgentIdentifiers, PlanInstances)` che indica quando una certa intenzione sospesa, in attesa sulla richiesta di

cooperazione `Request`, può essere ripristinata, fornendo l'insieme degli agenti che devono ancora rispondere alla richiesta in `AgentIdentifiers`, e l'inisieme delle istanze di piano rilevanti raccolte finora in `PlanInstances`;

- `getDesires(OrdinaryEvent, Desires)` che specifica una funzione totale che associa ogni evento ordinario all'insieme (anche vuoto) di desideri principali che deve essere generato a partire da tale evento;

- `selectInstance(PlanInstances, SelectedPlanInstance)` che specifica una funzione totale che restituisce un elemento specifico `SelectedPlanInstance` di ogni insieme non vuoto di istanze di piano `PlanInstances`, tale che se `SelectedPlanInstance` è l'istanza di un piano di default, allora `PlanInstances` non contiene istanze di piani specifici;

- `selectIntention(Intentions, SelectedIntention)` che specifica una funzione totale che restituisce un elemento specifico `SelectedIntention` di ogni insieme non vuoto di intenzioni attive;

- `selectState(States, SelectedState)` che specifica una funzione totale che restituisce un elemento `SelectedState` di ogni insieme non vuoto di stati `States`.

Tra i predicati elencati sopra `agentId`, `canResume`, `getDesires`, `selectInstance`, `selectIntention` e `selectState` sono *statici* in quanto non possono essere modificati nell'arco della vita dell'agente, mentre tutti gli altri possono essere modificati dinamicamente.

## 4.3 Specifica comportamentale di Coo-BDI

La descrizione del modo di procedere di Coo-BDI è data usando Prolog che è un linguaggio di programmazione logica [71]. Prolog è stato scelto perché offre alcuni vantaggi nel dettagliare il funzionamento di un sistema multi-agente:

- *esecuzione del MAS:* l'evoluzione di un MAS è costituita da una successione di eventi; da un punto di vista astratto un linguaggio di programmazione logica è un linguaggio non-deterministico in cui la computazione avviene attraverso un processo di ricerca.

- *capacità di Meta-ragionamento:* gli agenti necessitano di modificare dinamicamente il proprio comportamento per adattarlo ai cambiamenti dell'ambiente. La possibilità di vedere programmi come dati fornita dalla programmazione logica è utile in tal senso.

- *razionalità e reattività degli agenti:* sono collegate strettamente all'interpretazione *dichiarativa* e *operazionale* dei programmi logici in quanto un programma logico puro si può vedere come specifica della componente razionale di un agente e la visione operazionale dei programmi logici può essere usata per modellare il comportamento reattivo di un agente.

*Notazioni e assunzioni.* Le variabili iniziano con la lettera maiuscola. Si introducono:

- *predicati meta-logici* per testare l'identità di variabili e l'unificabilità di termini, per chiamare goal e per denotare congiunzioni di goal;

- *predicati extra-logici* per aggiornare lo stato dell'agente;

- *predicati del second'ordine* per recuperare un insieme di elementi che soddisfano una data condizione;

- *predicato di negazione* che è implementato usando `cut` e `fail`.

Si definisce un insieme di primitive standard per le strutture dati usate comunemente:

- *primitive per code*;

- *primitive per insiemi e relazioni*;

- *primitive per stacks*;

- *primitive per alberi*;

- *primitive per sequenze.*

Si forniscono inoltre delle primitive specifiche di Coo-BDI per lo scambio di piani e la socialità:

- `multicastRequestOp(AgentIdSet, Request)`: prende un insieme di identificatori d'agente `AgentIdSet` e una richiesta `Request` e mette `requested(Request)` nella event queue di ogni agente nell'insieme `AgentIdSet`;

- `provideOp(ProviderId, request(RequestId, AgentId, Desire), Instances)`: prende un identificatore d'agente `ProviderId`, una richiesta `request(RequestId, AgentId, Desire)` e un insieme di istanze `Instances` e mette `provided(ProviderId, request(RequestId, AgentId, Desire), Instances)` nella event queue di `AgentId`;

- `sendOp(SenderId, ReceiverId, Message)`: prende due identificatori d'agente, `SenderId` e `ReceiverId`, e un messaggio `Message` e mette `received(SenderId, Message)` nella event queue di `ReceiverId`. `Message` può essere un qualsiasi tipo di termine contenente eventualmente variabili libere che implementano le informazioni da passare in quanto per il momento non ci si conforma ad alcun linguaggio di comunicazione per agenti specifico.

Infine, sono disponibili le primitive:

- `newId(Identity)` per generare una nuova identità;
- `ground(Literal)`: che ha valore vero se `Literal` è ground;
- `apply(Theta, Term, TermTheta)`: prende una sostituzione `Theta` e un termine `Term` e restituisce il termine ottenuto da `Term` rimpiazzando ogni variabile `X` con `Theta(X)`;
- `compose(Sigma, Theta, SigmaTheta)`: prende due sostituzioni `Sigma` e `Theta` e restituisce la loro composizione;
- `mgu(Expr1, Expr2, Mgu)`: prende due espressioni e restituisce il loro most general unifier.

*Coo-BDI engine.* Come detto in precedenza, differisce dall'interprete classico del modello BDI in quanto tiene conto anche della generazione dei desideri e della cooperazione e si occupa di processare la event queue, le intenzioni sospese e quelle attive.

```
cooBDIengine :- processEventQueue, processSuspendedIntentions,
  processActiveIntentions.
```

*Processing Events.* Se la coda degli eventi è vuota non si fa nulla, altrimenti si prende dalla coda un evento `Event`, lo si gestisce e si aggiorna la coda degli eventi dell'agente.

```
processEventQueue :- eventQueue(EventQueue), empty(EventQueue).

processEventQueue :- eventQueue(EventQueue), not(empty(EventQueue)),
  get(Event, EventQueue, RemainingEventQueue), manageEvent(Event),
  retract(eventQueue(EventQueue)),
  assert(eventQueue(RemainingEventQueue)).
```

La gestione degli eventi può condurre a tre situazioni:

1) Se `Event` è di tipo `requested(request(RequestId, RequestingAgentId, Desire))`, allora il predicato `getRelInstance(Instances, request(RequestId, RequestingAgentId, Desire))` recupera tutte le istanze di piani specifici rilevanti per `Desire` il cui specificatore d'accesso è `public` o

`only(TrustedAgents)`, e per le quali `TrustedAgents` include `RequestingAgentId`.

L'insieme `Instances` recuperato è messo nella coda degli eventi di `RequestingAgentId` chiamando `provideOp(AgentId, request(RequestId, RequestingAgentId, Desire), Instances)`, in cui `AgentId` è l'identificatore dell'agente che porta a termine l'azione `provideOp`.

```
manageEvent(requested(request(ReqId, ReqAgentId, Desire))) :-
  agentId(AgentId),
  getRelInstances(Instances, request(ReqId, ReqAgentId, Desire)),
  provideOp(AgentId, request(ReqId, ReqAgentId, Desire), Instances).
```

2) L'`Event` è di tipo `providedOp(ProvidingAgentId, request(ReqId, AgentId, Desire), RetrievedInstances)`. Se esiste un'intenzione `Intention` tale che l'identificatore dell'intenzione e la richiesta `request(ReqId, AgentId, Desire)` appartengono alla relazione `relationIntentionRequest` si avvia il recupero delle istanze di piano rilevanti per `Desire`.
`Intention` è aggiornata aggingendo le `Instances` recuperate alle istanze rilevanti e rimuovendo l'identificatore dell'agente che ha risposto `(ProvidingAgentId)` dall'insieme `WaitingOnAgents`. L'intenzione aggiornata rimpiazza `Intention`. Altrimenti, se non c'è `Intention` tale che l'identificatore dell'intenzione e la richiesta appartengono a `relationIntentionRequest`, l'evento è ignorato.

```
manageEvent(provided(ProvidingAgentId, request(ReqId, AgentId,
        Desire), Instances)) :-
 relationIntentionRequest(IntentionId, request(ReqId, AgentId, Desire),
 isIntention(intention(IntentionId, Stack, Status, RelevantInstances,
   WaitingOnAgents, FailedInstances)),
 singleton(ProvidingAgentId, SingletonProvidingAgentId),
 setDifference(WaitingOnAgents, SingletonProvidingAgentId,
   UpdatedWaitingOnAgents),
 setUnion(Instances, RelevantInstances, UpdatedRelevantInstances),
 retract(isIntention(intention(IntentionId, Stack, Status,
   RelevantInstances, WaitingOnAgents, FailedInstances))),
 assert(isIntention(intention(IntentionId, Stack, Status,
   UpdatedRelevantInstances, UpdatedWaitingOnAgents,
 FailedInstances))).
```

3) Se `Event` è un evento ordinario, si recupera l'insieme dei desideri corrispondenti mediante `getDesires(OrdinaryEvent, DesireSet)` e l'insieme dei desideri principali viene aggiornato in modo da contenerli.

Per ogni desiderio generato che non è ancora nell'insieme dei desideri principali si crea e si inizializza un nuovo stack delle intenzioni `createIntention`, l'insieme delle intenzioni e la relazione `DesireIntention` sono aggiornate per tener conto delle nuove intenzioni create e il recupero di istanze di piano rilevanti per l'intenzione e il desiderio viene avviata (mediante la chiamata a `retrieveRelevantInstance(Intention, Desire)` all'interno di `createOneIntentionForOneDesire(Desire)` richiamata a sua volta da `createIntentionsForDesires(DesireSet)`).

```
manageEvent(Event) :- Event \= provided(_,_,_), Event \= requested(_),
  getDesires(Event, DesireSet), createIntentionsForDesires(DesireSet).
```

Quando si crea una nuova intenzione le sue componenti `RelevantInstances`, `WaitingOnAgents` e `FailedInstances` assumono come valore l'insieme vuoto e il suo stato ha valore `suspended`.

```
createIntention(IntentionId) :-
  assert(isIntention(intention(IntentionId, emptyStack, suspended,
  emptySet, emptySet, emptySet))).
```

Aggiornare un'intenzione recuperando le istanze di piano rilevanti per il desiderio che l'intenzione sta attualmente tentando di soddisfare (sia esso principale o subalterno) significa assegnare all'intenzione il valore `suspended`, prendere le istanze di piano locali rilevanti per il desiderio (`getLocalRelevantInstances(Desire, LocalRelevantInstances)`), assegnare l'insieme restituito alla componente `RelevantInstances` e aggiornare la componente `WaitingOnAgents` secondo la politica di recupero dei piani. Se la politica di recupero è `always` o `noLocal` si cercano le istanze locali di piani specifici, la componente `WaitingOnAgents` è aggiornata con il valore dell'insieme degli agenti fidati, si emette una richiesta di piani per il desiderio per ogni agente fidato (`multicastRequestOp(TrustedAgents, Request)`) e si aggiorna la relazione `relationIntentionRequest`. Altrimenti, l'insieme `WaitingOnAgents` è inizializzato al valore vuoto e non si mandano richieste di piano.

```
retrieveRelevantInstances(IntentionId, Desire) :- agentId(AgentId),
  getLocalRelevantInstances(Desire, LocalRelInst),
  (retrievalPolicy(always); specificPlans(LocalRelInst, emptySet)),
  trustedAgents(TrustedAgents), newId(RequestId),
  multicastRequestOp(AgentId, TrustedAgents,
    request(RequestId, AgentId, Desire)),
  retract(isIntention(intention(IntentionId, Stack, _Status,
    _RelevantInstances, _WaitingOnAgents, FailedInstances))),
  assert(isIntention(intention(IntentionId, Stack, suspended,
    LocalRelInst, TrustedAgents, FailedInstances))),
  assert(relationIntentionRequest(IntentionId,
    request(RequestId, AgentId, Desire))).

retrieveRelevantInstances(IntentionId, Desire) :-
  getLocalRelevantInstances(Desire, LocalRelevantInstances),
  retrievalPolicy(noLocal),
  not(specificPlans(LocalRelevantInstances, emptySet)),
  retract(isIntention(intention(IntentionId, Stack, _Status,
    _RelevantInstances, _WaitingOnAgents, FailedInstances))),
  assert(isIntention(intention(IntentionId, Stack, suspended,
     LocalRelevantInstances, emptySet, FailedInstances))).
```

*Processing Suspended Intentions.* Durante questo passo l'interprete controlla se ci sono intenzioni sospese che possono essere ripristinate in quanto la richiesta `Request` associata, la componente `WaitingOnAgents` e la componente `RelevantInstances` soddisfano la condizione `canResume`. Se non

si trova un'intenzione ripristinabile, `processSuspendedIntentions` procede senza fare nulla.

```
processSuspendedIntentions :-
  isIntention(intention(IntentionId, _Stack, _Status,
  RelevantInstances, WaitingOnAgents, _FailedInstances)),
  relationIntentionRequest(IntentionId, Request),
  canResume(Request, WaitingOnAgents, RelevantInstances),
  resume(IntentionId, Request).
```

Se si trova un'intenzione ripristinabile, si possono verificare due casi:

1) Lo stack dell'intenzione è vuoto: il desiderio per cui si sono raccolte le istanze di piano rilevanti è un desiderio principale. Per eseguire il meccanismo di backtracking sulle istanze di piano che possono essere usate per soddisfare il desiderio principale, non si riprovano le istanze di piano per le quali si è già verificato un fallimento. Le istanze di piano applicabili sono quindi determinate a partire dalla collezione di istanze rilevanti `RelevantInst` meno le istanze fallite.

```
resume(IntentionId, Request) :-
  isIntention(intention(IntentionId, emptyStack, _Status,
  RelevantInst, _WaitingOnAgents, FailedInst)),
  setDifference(RelevantInst, FailedInst, NotAttemptedInst),
  getApplInstances(ApplicableInstSet, NotAttemptedInst),
  manageApplicableInstances(ApplicableInstSet, IntentionId, Request).
```

2) Lo stack dell'intenzione è non vuoto: il desiderio per cui si sono raccolte le istanze di piano rilevanti è un desiderio subalterno e per esso non viene eseguito alcun meccanismo di backtracking. Le istanze di piano applicabili sono determinate a partire da `RelevantInst`.

```
resume(IntentionId, Request) :-
  isIntention(intention(IntentionId, Stack, _Status, RelevantInst,
  _WaitingOnAgents, _FailedInst)), not(empty(Stack)),
  getApplInstances(ApplicableInstSet, RelevantInst),
  manageApplicableInstances(ApplicableInstSet, IntentionId, Request).
```

L'atomo `getApplInstances(ApplicableInstSet, RelevantInst)` unifica `ApplicableInstancesSet` con le coppie `(Plan, Substitution)` ottenute da `RelevantInst` in modo che la precondizione del piano istanziata con `Substitution` sia una conseguenza logica ground dei belief dell'agente.

Per definizione, per ogni desiderio c'è almeno un'istanza di piano rilevante (quella originata da un piano di default) e questa istanza è anche applicabile (la precondizione è sempre vera): quando `getApplInstances` è applicata a un insieme non vuoto di istanze di piano rilevanti restituisce sempre almeno un'istanza di piano; restituisce l'insieme vuoto solo se il suo argomento è l'insieme vuoto. `ApplicableInstanceSet` può essere vuoto solo se tutte le istanze rilevanti per il desiderio principale sono fallite.

Se l'insieme delle istanze applicabili è vuoto il desiderio fallisce: sia il desiderio, sia l'intenzione vengono rimossi dai corrispondenti insiemi e le relazioni che li coinvolgono vengono aggiornate.

```
manageApplicableInstances(emptySet, IntId,
request(ReqId, AgId, Des)) :-
  isDesire(Des), isIntention(intention(IntId, _Stack, _Status,
  _RelevantInstances, _WaitingOnAgents, _FailedInstances)),
  relationDesireIntention(Des, IntId),
  relationIntentionRequest(IntId, request(ReqId, AgId, Des)),
  retract(isDesire(Des)),
  retract(isIntention(intention(IntId, _Stack, _Status,
  _RelevantInstances, _WaitingOnAgents, _FailedInstances))),
  retract(relationDesireIntention(Des, IntId)),
  retract(relationIntentionRequest(IntId, request(ReqId, AgId, Des))).
```

Se l'insieme delle istanze applicabili è non vuoto da esso si sceglie un'istanza di piano e si avvia la sua esecuzione assegnando al campo per l'istanza il valore dell'istanza di piano scelta, al campo per la sostituzione il contenuto di `Substitution`, allo stato corrente il valore della radice del corpo del piano e al prossimo stato il figlio dello stato corrente. L'intenzione pronta per l'esecuzione è messa in cima allo stack dell'intenzione e lo stato dell'intenzione assume valore `active`. Se il piano è già presente nella libreria dei piani non si fa nulla. Altrimenti, a seconda della politica di acquisizione dell'agente, il piano scelto è scartato, aggiunto alla libreria dei piani o usato per rimpiazzare tutti i piani esistenti il cui trigger unifica con il trigger del piano a cui è applicata `Substitution`.

```
manageApplicableInstances(ApplicableInstancesSet, IntentionId,
  _Request) :- ApplicableInstancesSet \= emptySet,
  selectInstance(ApplicableInstancesSet, (Plan, Substitution)),
  createExecution((Plan, Substitution), Execution),
  isIntention(intention(IntentionId, Stack, Status, RelevantInstances,
  WaitingOnAgents, FailedInstances)),
  push(Execution, Stack, UpdatedStack), acquirePlan(Plan),
  retract(isIntention(intention(IntentionId, Stack, Status,
  RelevantInstances, WaitingOnAgents, FailedInstances))),
  assert(isIntention(intention(IntentionId, UpdatedStack, active,
  RelevantInstances, WaitingOnAgents, FailedInstances))).
```

*Processing Active Intentions.* Le intenzioni il cui stato è `active` sono processate come in dMARS. Se non ci sono intenzioni attive non si fa nulla, altrimenti se ne sceglie una e si manda in esecuzione il piano al top del suo stack. Si possono verificare tre casi.

1. Lo stato corrente è una foglia: l'istanza di piano è eseguita con successo e viene rimossa dallo stack dell'intenzione. La sostituzione associata all'intenzione per l'esecuzione viene applicata alla sequenza di azioni interne che sarà quindi eseguita. Se non ci sono altri elementi sullo stack, l'intenzione è stata portata a compimento con successo e può

essere rimossa, così come il desiderio principale che l'ha generata (si deve aggiornare la relazione `relationDesireIntention`). Altrimenti, la sostituzione è composta con la sostituzione dell'istanza attualmente in esecuzione al top dello stack.

2. L'istanza in esecuzione al top dello stack non verifica l'invariante o lo stato corrente non è una foglia e non ci sono stati da esso raggiungili: l'istanza di piano fallisce. La sostituzione associata per l'esecuzione viene applicata alle azioni per il fallimento del piano che sono quindi eseguite; infine, si aggiorna l'insieme delle istanze di piano fallite con l'istanza alla base dello stack dell'intenzione e si rimuovono tutti gli elementi dello stack. Si provano tutte le alternative per il soddisfacimento del desiderio principale associato all'intenzione.

3. L'istanza in esecuzione al top dello stack non fallisce né ha successo: si recupera l'azione da compiere che etichetta l'arco tra lo stato corrente e lo stato successivo scelto. L'azione può essere

   (a) `add(BeliefFormula)` o `remove(BeliefFormula)`: `BeliefFormula`, che al momento della chiamata dell'operazione deve essere ground, è aggiunta a (rimossa da) l'insieme dei belief dell'agente. L'agente manda a se stesso un messaggio per notificare che l'insieme dei belief è cambiato.

   (b) `send(ReceiverAgentId, Message)`: `agentId(AgentId)` si usa per trovare l'identità del mittente e si usa `sendOp(ReceiverAgentId, AgentId, Message)` per mandare il messaggio.

   (c) `query(SituationFormula)`: se `SituationFormula` istanziata con la sostituzione per l'esecuzione corrente può essere resa ground, e l'istanza ground è una conseguenza logica dei belief dell'agente, allora la sostituzione per l'esecuzione è aggiornata componendola con la sostituzione di base. Altrimenti, lo stato successivo scelto è rimosso dall'insieme degli stati raggiungibili dallo stato corrente.

   (d) `achieve(Desire)`: se `Desire` instanziato con la sostituzione per l'esecuzione corrente può essere reso ground, e l'istanza ground è una conseguenza logica dei belief dell'agente, allora la sostituzione per l'esecuzione è aggiornata componendola con la sostituzione di base. Altrimenti, il meccanismo di recupero delle istanze rilevanti è avviato con il desiderio istanziato con la sostituzione per l'esecuzione corrente.

Dopo che un'azione è stata eseguita, si aggiorna l'istanza di piano correntemente in esecuzione.

## 4.4 Vantaggi dell'estensione Coo-BDI

Coo-BDI è adatto per modellare i "learning agent" che apprendono dall'interazione con altri agenti. L'estensione Coo-BDI consente di modellare il comportamento umano e questo comporta sicuramente dei vantaggi che possono essere evidenziati con l'analisi dell'esempio seguente.

Si vuole modellare questa situazione:

1. Valentina e Roberto ricevono una e-mail da Maurizio che li invita a raggiungerlo a casa di Chiara per una festa alle 19.

2. Valentina vorrebbe andare alla festa: desidera cancellare tutti gli appuntamenti presi in precedenza per questa sera e raggiungere la casa di Chiara alle 19. Invece, a Roberto non piacciono le feste: desidera telefonare a casa di Chiara per declinare l'invito.

3. Valentina sa come concellare gli appuntamenti precedenti.

4. Valentina non sa come raggiungele la casa di Chiara. Roberto non sa come telefonare a casa di Chiara.

Per ognuna delle affermazioni sopra si deve mostrare come possono essere modellate in BDI e in Coo-BDI per poi discutere il comportamento degli interpreti BDI e Coo-BDI sugli agenti dati.

*Valentina e Roberto ricevono una e-mail da Maurizio.* Sia in BDI, sia in Coo-BDI, la ricezione di una e-mail da Maurizio si modella con l'inserimento di un evento

```
received(maurizio,
   message(by(e-mail), content(join_party, chiara_home, today, 7pm)))
```

entro le event queue di Valentina e di Roberto.

*Valentina desidera cancellare gli appuntamenti presi in precedenza e raggiungere casa di Chiara. Roberto desidera telefonare a Chiara e declinare l'invito.* In BDI non c'è un modo chiaro di modellare una corrispondenza tra eventi esterni e relativi desideri. Questa corrispondenza può essere modellata in modo implicito definendo un piano il cui trigger sia l'evento esterno e il cui corpo contenga il desiderio da esaudire. I piani BDI sono rappresentati da `plan(Trigger, Precondition, Body, Invariant, SuccessActionSequence, FailureActionSequence)`. Per una miglior leggibilità i corpi di piano saranno rappresentati graficamente. Il piano BDI per Valentina potrebbe essere:

```
plan(received(Sender, message(by(CommunicationMeans),
      content(join_party, Where, Day, Hour))),    ⇐ Trigger
   true,                                          ⇐ Precondition
   s0 •
      ↓ achieve(del_appointments(Day, Hour))
```

```
s1 •                                                ⇐ Body
   ↓ achieve(go_to(Where, Day, Hour))
s2 •,
true, emptySeq, emptySeq)                           ⇐ Inv., Succ. and Fail.
```

Un piano simile, con corpo differente, può essere usato per simulare la generazione di desideri per Roberto.

In Coo-BDI la corrispondenza tra eventi esterni e desideri è rappresentata esplicitamente dal predicato `getDesire(OrdinaryEvent, DesireSet)` che ogni agente deve definire. La specifica dell'agente Valentina potrebbe quindi includere:

```
getDesires(received(Sender,message(by(CommunicationMeans),
   content(join_party, Where, Day, Hour))),
   {achieve(del_appointments(Day, Hour),
   achieve(go_to(Where, Day, Hour))})
```

mentre quella dell'agente Roberto potrebbe includere:

```
getDesires(received(Sender,message(by(CommunicationMeans),
   content(join_party, Where, Day, Hour))),
   {achieve(phone(Where, content(decline))})
```

Una volta che si sono trovati i desideri principali legati alla ricezione dell'e-mail da Maurizio per ognuno di essi si crea un'intenzione. L'intenzione per il desiderio `achieve(go_to(chiara_home, today, 7pm))` si identifica con `go_to_chiara_intention`.

*Valentina sa come cancellare gli appuntamenti precedenti.* Sia in BDI, sia in Coo-BDI, il fatto che Valentina sappia come cancellare gli appuntamenti presi in precedenza significa che nella libreria dei piani di Valentina c'è almeno un piano attivato da `achieve(del_appointments(today, 7pm))`. Per esempio il piano mostrato sotto rappresenta il fatto che la cancellazione degli appuntamenti con una lista di persone richiede il recupero della lista di persone e l'invio di un messaggio a ogni persona della lista. Un piano per gestire `achieve(get_appointment_list(Day, Hour, List))` dovrebbe appartenere alla libreria dei piani di Valentina e l'azione esterna `multicast(List, content(delete_appointment))` dovrebbe essere definita in termini di `send`.

Il piano Coo-BDI appare esattamente come quello BDI più gli specificatori d'accesso.

```
plan(achieve(del_appointments(Day, Hour)),   ⇐ Trigger
   true,                                     ⇐ Precondition
   s0 •
      ↓ achieve(get_appointment_list(Day, Hour, List))
   s1 •                                                ⇐ Body
      ↓ multicast(List, content(delete_appointment))
   s2 •,
   true, emptySeq, emptySeq)                 ⇐ Inv., Succ. and Fail.
```

*Valentina non sa come raggiungere casa di Chiara. Roberto non sa come telefonare a casa di Chiara.* Sia in Coo-BDI, sia in BDI questa situazione è modellata dall'assenza, nella libreria dei piani di Valentina, di piani attivati dal desiderio `achieve(go_to(chiara_home, today, 7pm))` e dall'assenza, nella libreria dei piani di Roberto, di piani attivati dal desiderio `achieve(phone(Where, content(decline))`.

La differenza tra Coo-BDI e BDI sta nel modo in cui è gestita la mancanza di piani rilevanti per il soddisfacimento di tali desideri.

Si consideri prima il caso di Valentina. In molti sistemi BDI, Valentina può abbandonare questo desiderio o adottare un piano di default. Rinunciare al desiderio non modella correttamente il comportamento umano consueto: nessuno rinuncerebbe ad andare ad una festa perché non sa come raggiungere il luogo dove si tiene la festa. Usare un piano di default potrebbe essere un'ottima soluzione ma i piani di default dovrebbero essere progettati per essere utilizzabili in tutte le situazioni per cui non sono disponibili piani specifici. Questo richiede molta intuizione per prevedere tutte queste situazioni in anticipo. Per esempio, un piano di default che dice "cerca nelle pagine bianche/gialle" potrebbe essere utile in molte situazioni, compresa questa, ma è completamente inutile per esaudire desideri del tipo "prepara una torta" o "supera un esame".

Estendere il modello BDI con la cooperazione aiuta a superare la mancanza di piani rilevanti in modo simile al comportamento umano. Si supponga che la strategia di cooperazione di Valentina sia caratterizzata da:

```
trustedAgents({maurizio, davide, sonia})
retrievalPolicy(noLocal)
acquisitionPolicy(add)
```

Poiché Valentina non ha piani locali rilevanti per il desiderio `achieve(go_to(chiara_home, today, 7pm))`, il recupero di piani esterni viene avviato:

- si crea una richiesta `request(go_to_chiara_request, valentina, achieve(go_to(Chiara_home, today, 7pm)))` per ottenere piani rilevanti per il desiderio.
- la richiesta è mandata ad ogni agente dell'insieme {`maurizio, davide, sonia`} per mezzo della primitiva `multicastRequestOp`.
- alla componente `WaitingOnAgent` dell'intenzione `go_to_chiara_intention` è assegnato l'insieme {`maurizio, davide, sonia`}.
- l'intenzione `go_to_chiara_intention` viene sospesa.

Il processo cooperativo di recupero di piani dipende dal contenuto delle librerie dei piani di Maurizio, Davide e Sonia. Maurizio ha solo il piano rilevante pubblico mostrato sotto. Quando riceve la richiesta di Valentina egli manda l'istanza di piano corrispondente a Valentina.

```
plan(public,                                        ⇐ Access specifier
   achieve(go_to(chiara_home, Day, Hour)),          ⇐ Trigger
   true,                                            ⇐ Precondition
   s0 •
      ↓ achieve(drive_to(collins_street, Day, (Hour-30min)))
   s1 •                                                          ⇐ Body
      ↓ achieve(park_near(main_church))
   s2 •,
   true, emptySeq, emptySeq)                        ⇐ Inv., Succ. and Fail.
```

Davide ha solo un piano privato per raggiungere casa di Chiara e di conseguenza manda a Valentina un insieme di istanze vuoto. Infine, Sonia ha un unico piano rilevante mostrato sotto. Poiché Valentina è inclusa tra le persone autorizzate a ricevere il piano, Sonia manda l'istanza corrispondente ad esso a Valentina.

```
plan(only({valentina, alberto}),                    ⇐ Access specifier
   achieve(go_to(chiara_home, Day, Hour)),          ⇐ Trigger
   true,                                            ⇐ Precondition
   s0 •
      ↓ achieve(take_a_bus(number(168), Day, (Hour-40min)))
   s1 •                                                          ⇐ Body
      ↓ achieve(get_off(collins))
   s2 •,
   true, emptySeq, emptySeq)                        ⇐ Inv., Succ. and Fail.
```

Quando Valentina riceve tutte e tre le risposte, procede come segue:

- realizza che l'intenzione `go_to_chiara_intention` può essere ripristinata.

- genera tutti i piani applicabili dall'insieme di quelli rilevanti (i due piani ricevuti da Maurizio e Sonia più tutte le istanze di piano di default). In questo semplice esempio, tutti i piani rilevanti sono anche applicabili.

- seleziona uno dei piani specifici applicabili associati all'intenzione.

- se il piano scelto è uno di quelli esterni (i soli piani specifici applicabili sono quelli ricevuti da Maurizio e Sonia) lo memorizza.

Probabilmente, un umano si comporterebbe in modo simile: contatterebbe alcuni amici chiedendo consigli, sceglierebbe le istruzioni più convenienti e memorizzerebbe le istruzioni per riutilizzarle in futuro.

Roberto si comporterà come Valentina per ottenere il numero telefonico di casa di Chiara e declinare l'invito. Se non si separano gli eventi dai desideri, l'attitudine completamente diversa di Valentina e Roberto non può essere catturata facilmente. Sopponendo che la ricezione dell'evento `received(maurizio, message(by(e-mail), content(join_party, chiara_home, today, 7pm)))` attivi un piano come nel caso BDI e che né Valentina né Roberto posseggano un piano rilevante per esso, la strategia

cooperativa implementata in Coo-BDI porterebbe alla richiesta di piani per la gestione di tale evento a tutti gli agenti fidati. Questo però ha poco senso in quanto i piani rilevanti recuperati rappresenterebbero suggerimenti veramente eterogenei quali declinare l'invito, raggiungere la festa, chiedere di posticipare la festa o inventare rapidamente una scusa. Come fa un agente a scegliere il piano corretto tra questi se non conosce quale stato vuole raggiungere? Questo problema si supera associando gli eventi esterni con desideri generati internamente e cooperando per esaudire desideri piuttosto che per gestire eventi esterni.

## 4.5 Coo-AgentSpeak e altre estensioni del modello BDI

Sono state proposte molte altre estensioni al modello BDI [3]. Tra tutte le estensioni proposte il lavoro più vicino alla proposta del Coo-BDI è l'estensione della semantica operazionale di AgentSpeak(L) con l'introduzione di comunicazione basata su speech-act [54]. Così come succede per la proposta Coo-BDI, anche questa estensione di AgentSpeak(L) è sufficientemente dettagliata da poter essere implementata. L'estensione di AgentSpeak(L), inoltre, ha lo stesso scopo di forzare la cooperazione tra agenti di Coo-BDI in quanto consente agli agenti di scambiare informazioni e di delegare alcuni compiti ad altri agenti.

L'aspetto che rende più simili Coo-BDI e AgentSpeak(L) esteso è il fatto che entrambi consentono agli agenti lo scambio di piani. In entrambi i casi si può modellare facilmente un agente BDI di base. Nell'estensione di AgentSpeak(L), quando l'insieme degli agenti fidati è vuoto, l'agente può semplicemente scartare informazioni e richieste. Un agente BDI classico può essere modellato in Coo-BDI assegnando valore *private* allo specificatore d'accesso di tutti i suoi piani e assegnando il valore vuoto all'insieme degli agenti fidati. In questo modo, l'agente non cercherà piani rilevanti esterni e non condividerà i propri piani, esattamente come succede nel modello BDI originale.

Queste due estensioni proposte sono così vicine che è naturale pensare che possano convergere in un unica architettura per agenti BDI altamente cooperativi. I messaggi dovrebbero aderire alla forma proposta per AgentSpeak(L) esteso in modo da avere una semantica chiara. I piani andrebbero estesi con specificatori d'accesso, in modo che gli agenti possano scegliere quando un piano possa essere condiviso con altri agenti mediante un messaggio `tellHow`. L'interprete dovrebbe considerare entrambe le estensioni. Si dovrebbe realizzare l'implementazione tenendo conto della formalizzazione e dell'approccio presentati in entrambi i lavori. Nei capitoli successivi si descriverà una proposta di implementazione per questa architettura Coo-

AgentSpeak che vuole fondere le due estensioni Coo-BDI e “AgentSpeak(L) comunicativo”.

# Capitolo 5

# Coo-AgentSpeak

Coo-AgentSpeak fornisce i mezzi necessari affinché agenti autonomi possano scambiarsi piani in modo che un agente possa accrescere la propria conoscenza procedurale relativa al soddisfacimento di un certo obiettivo. Questa conoscenza aggiuntiva è rappresentata dai piani ottenuti da altri agenti che ne sono in possesso.

Coo-AgentSpeak porta il linguaggio ad un maggior livello di astrazione. Con esso diventa possibile programmare sistemi multi-agente, scritti in AgentSpeak(L), ad un livello più alto. L'utente non deve preoccuparsi di certi dettagli relativi al trasferimento di conoscenza tra gli agenti, già reso possibile grazie alla forza illocutoria *tellHow* introdotta con l'estensione di AgentSpeak(L) basata sugli speech-act, in quanto questo avviene in modo automatico attraverso il meccanismo di richiesta e recupero di piani esterni.

Come anticipato, Coo-AgentSpeak [4] nasce dall'integrazione dell'idea Coo-BDI [3] con Jason [8], un'implementazione per l'estensione di AgentSpeak(L) con comunicazione basata sugli speech-act [54].

Nelle sezioni che seguono si descrivono le variazioni apportate al modello Coo-BDI in vista di questa integrazione e le caratteristiche di Jason.

Nell'ultima sezione viene accennato uno studio di fattibilità relativo alla proposta per la realizzazione di Coo-AgentSpeak formulata in [4]. Vengono descritti i punti salienti della proposta e gli eventuali scostamenti da questa con le relative motivazioni.

## 5.1 Coo-BDI per Coo-AgentSpeak

La versione di Coo-BDI su cui si fonda la realizzazione di Coo-AgentSpeak è diversa da quella originale [3] e qui vengono descritte alcune delle ragioni che ne giustificano le differenze.

**Aumento della flessibilità e del potere espressivo di Coo-BDI.** Si ridefinisce la granularità delle strategie di cooperazione. Diventa possibile applicare diverse strategie a tipi diversi di piani invece di avere

un'unica politica di cooperazione per tutti i piani. I piani sono estesi in modo da includere l'indicazione della sorgente del piano, cioè dell'agente che originariamente "possiede" il piano.

**Conformità tra AgentSpeak e Jason.** Per facilitare l'integrazione tra Coo-BDI e AgentSpeak implementato in Jason [8] si abbandona la distinzione tra "eventi esterni" e "desideri" proposta in [3]: si considerano solo gli "eventi" di AgentSpeak(L) che corrispondono ai "desideri" di Coo-BDI; i piani sono adattati alla sintassi di Jason: costrutti quali invarianti, azioni di successo e fallimento e le possibili alternative entro il body sono eliminati; i messaggi scambiati tra gli agenti sono modificati in modo da essere conformi con la sintassi di Jason.

Le modifiche apportate al modello Coo-BDI vengono analizzate nel seguito sfruttando la suddivisione seguita per la descrizione generale di Coo-BDI nel capitolo precedente in modo da rendere meglio individuabili le differenze da esse introdotte.

*Strategia di cooperazione.* La strategia di cooperazione di un agente `Ag` può evolvere nel tempo ed è modellata da tre predicati:

- `trusted(Te, TrustedAgentSet)` dove `Te` è un triggering event (non necessariamente ground) e `TrustedAgentSet` è un insieme di agenti che `Ag` può contattare per ottenere piani rilevanti per `Te`.
- `retrievalPolicy(Te, Retrieval)` dove `Retrieval` può assumere i valori *always*, se i piani rilevanti per `Te` possono essere recuperati da altri agenti in ogni caso, o *noLocal* se i piani rilevanti per `Te` possono essere recuperati solo quando non sono disponibili piani rilevanti locali.
- `acquisitionPolicy(Te, Acquisition)` dove `Acquisition` può assumere i valori *discard*, *add* e *replace* a seconda che un piano rilevante per `Te` recuperato da un agente fidato debba essere, rispettivamente, scartato dopo il suo utilizzo, aggiunto alla libreria dei piani o usato per aggiornare la libreria dei piani rimpiazzando tutti i piani attivabili da `Te` esistenti.

Sono questi predicati che consentono di associare la strategia di cooperazione al tipo del piano e non più all'intero insieme dei piani dell'agente.

*Piani.* Nell'estensione di Coo-BDI i piani sono, come anticipato, adattati alla sintassi di Jason. Le componenti che costituiscono un piano sono quindi quelle standard, come il triggering event, il contesto, il body (privo di alternative), a cui si aggiungono una *sorgente* e uno *specificatore d'accesso*. La sorgente si riferisce all'agente da cui si ottiene il piano e può assumere valore *self* se è l'agente stesso a possedere il piano, o *Ag* se il piano apparteneva

originariamente all'agente *Ag*. Lo specificatore d'accesso determina l'insieme di agenti con cui un piano può essere condiviso ed è definito in modo analogo a quanto visto nella descrizione originale di Coo-BDI [3] e in quella del capitolo precedente.

*Intenzioni.* Le intenzioni sono state adattate alla sintassi di Jason. L'intenzione è costituita dall'unica componente stack che conterrà le istanze di piano da eseguire. All'intenzione vengono associate alcune informazioni introdotte per gestire il meccanismo di recupero dei piani esterni:

- i piani rilevanti che sono già stati raccolti per la gestione del goal corrente;
- l'insieme di identificatori di quegli agenti con cui si aspetta ancora di collaborare (ricevendo piani da essi) per il soddisfacimento del goal corrente.

Un'intenzione può essere *active* o *suspended*. È suspended quando l'esecuzione del piano in cima al suo stack genera un subgoal e il recupero dei piani rilevanti per tale subgoal non è ancora completato. Lo stato dell'intenzione non è indicato in alcuna componente ma è determinato dall'assegnazione dell'intenzione a determinati insiemi o relazioni.

*Interprete Coo-BDI.* L'interprete per Coo-BDI è diverso dal classico interprete BDI ed è caratterizzato da quattro passi: (1) processare la mailbox; (2) processare la event queue; (3) processare le intenzioni sospese; (4) processare le intenzioni attive. Il completamento di questi passi è strettamente legato al meccanismo di recupero dei piani rilevanti. Tale meccanismo, come accenato nella descrizione di Coo-BDI, è avviato quando un nuovo evento entra nella event queue. La gestione di eventi esterni e interni è resa omogenea, associando un'intenzione a entrambi i tipi di evento che nel caso in cui sia un evento esterno ad entrare nella event coincide con una nuova intenzione vuota.

Il meccanismo di recupero dei piani qui è descritto evidenziando l'associazione tra triggering event `Te` e intenzione ed è costituito da quattro passi sequenziali:

(a) l'intenzione associata a `Te` viene sospesa;

(b) si generano i piani rilevanti locali per `Te` e vengono associati all'intenzione sospesa;

(c) si definisce l'insieme `S` degli agenti con cui ci si aspetta di cooperare per gestire l'evento (a seconda della strategia di cooperazione prevista per gli eventi che corrispondono a `Te`) e lo si associa all'intenzione sospesa;

(d) se `S` $\neq \{\}$, si crea una richiesta di piano per l'evento `Te`, la si manda a tutti gli agenti in `S` e si associa all'intenzione sospesa l'identificatore unico di tale richiesta.

In dettaglio i quattro passi dell'interprete consistono in:

1. *Processare la mailbox:* quando un agente riceve una richiesta di piano da un altro agente Ag, esso manda ad Ag i suoi piani locali che sono rilevanti per tale evento e che possono essere condivisi con Ag. D'altra parte, quando un agente riceve una risposta a una richiesta di piani per la gestione di un certo evento, esso controlla che la risposta sia ancora valida e, in caso affermativo, aggiorna l'intenzione associata all'evento affinché includa il piano appena ottenuto.

2. *Processare la event queue:* dopo che un evento è stato selezionato, viene avviato il meccanismo per il recupero dei piani attivabili da tale evento.

3. *Processare le intenzioni sospese:* la gestione delle intenzioni sospese consiste nel guardare tra tutte le intenzioni sospese quali possono essere ripristinate. Quando un'intenzione è ripristinata, si genera l'insieme delle istanze di piano applicabili a partire dall'insieme di piani rilevanti associato all'intenzione, si seleziona un'istanza di piano applicabile e la si mette in cima allo stack dell'intenzione corrispondente. Se l'insieme dei piani applicabili è vuoto, l'evento per cui si sono raccolti i piani non può essere gestito e si procede scartando tale evento o attivando un meccanismo per la gestione ad hoc del fallimento dell'evento stesso mediante piani appositamente definiti. Questo meccanismo è riconducibile all'introduzione dei piani di default per l'evento prevista da Coo-BDI e verrà descritto meglio nella Sottosezione **Sistema run-time di Jason** 5.2.2.

   I piani recuperati dagli agenti cooperativi possono essere scartati, aggiunti alla libreria dei piani, o usati per rimpiazzare piani locali con un trigger event unificante, a seconda della politica di acquisizione relativa al trigger event. Si noti che la politica di acquisizione descritta per Coo-BDI originale riguarda solo il piano scelto per l'esecuzione dopo il recupero esterno mentre quella per Coo-AgentSpeak riguarda l'intero insieme dei piani rilevanti ottenuti dopo il recupero esterno (che potrebbe contenere anche piani rilevanti locali se la politica di recupero era noLocal). La scelta fatta per Coo-AgentSpeak appare più convincente in quanto, nei casi in cui la politica di acquisizione impone di aggiungere o rimpiazzare i piani, si acquisisce conoscenza procedurale in quantità maggiore, o nel caso peggiore uguale, rispetto a quella che si acquisirebbe scegliendo l'approccio proposto da Coo-BDI originale, traendo il maggior vantaggio possibile dal meccanismo di recupero esterno attivato.

4. *Processare le intenzioni attive:* le intenzioni attive sono processate come nell'architettura BDI originale.

Un'assunzione che viene fatta sia per Coo-BDI, sia per Coo-AgentSpeak è che gli agenti condividano l'ontologia usata per lo scambio di piani. Un'altra assunzione che si fa è che tutti i piani siano scritti seguendo la sintassi di AgentSpeak.

## 5.2 Jason

*Jason* è un interprete progettato e realizzato per una versione di AgentSpeak(L) estesa per supportare la comunicazione inter-agente basata sugli speech-act che consente di avere agenti distribuiti sulla rete attraverso l'uso di SACI [36]. È stato realizzato da R. H. Bordini e J. F. Hübner [8] e la sua specifica è stata formalizzata mediante una semantica operazionale fornita per AgentSpeak(L) e per la sua estensione comunicativa [9, 54]. Jason è implementato in Java, caratteristica che lo rende portabile su diverse piattaforme, ed è disponibile *Open Source* sotto GNU LGPL alla pagina `http://jason.sourceforge.net`.

Oltre a interpretare il linguaggio AgentSpeak(L) originale, Jason offre altre funzionalità, quali:

- negazione forte (strong negation) e disponibilità di entrambe le assunzioni mondi aperti e mondi chiusi (closed-world e open-world);
- gestione del fallimento di piani;
- supporto per lo sviluppo di Environments che vengono programmati in Java anziché in AgentSpeak;
- una libreria di "azioni interne" essenziali;
- comunicazione inter-agente basata sugli speech-act e annotazione dei belief con informazioni sulla loro sorgente;
- annotazione di piani con etichette che possono essere usate per elaborare le funzioni di selezione;
- possibilità di eseguire un sistema multi-agente distribuito su una rete usando SACI;
- funzioni di selezione, funzioni di fiducia e architettura complessiva dell'agente (percezioni, revisione di belief, comunicazione inter-agente e funzionamento) completamente modificabili secondo esigenze specifiche (in Java);
- estensibilità semplice mediante l'uso di azioni interne definite dall'utente che sono programmate in Java.

Di queste caratteristiche le ultime cinque sono quelle rilevanti rispetto alla realizzazione di un interprete per Coo-AgentSpeak.

### Simple Agent Communication Infrastructure (SACI)

La *Simple Agent Communication Infrastructure (SACI)* è stata sviluppata da J. F. Hübner e J. S. Sichman nel 2003. Il suo scopo è fare in modo che agenti distribuiti comunichino in modo semplice. È costituita da una API (Application Program Interface) Java e da un insieme di strumenti che possono essere usati per supportare lo sviluppo di società di agenti distribuiti.

SACI ha le seguenti caratteristiche:

- Gli agenti sono raggruppati in società secondo un modello organizzativo (*Moise+* [38]).
- Gli agenti comunicano tra loro mediante messaggi KQML.
- Gli agenti sono identificati da un nome. Essi possono mandare messaggi ad altri agenti solo usando il nome del destinatario. La localizzazione nella rete è resa trasparente al mittente mediante un *facilitator* (servizio di pagine bianche).
- Gli agenti possono conoscere gli altri mediante un servizio di pagine gialle. Gli agenti possono registrare i propri servizi con il facilitator e interrogare tale facilitator per trovare quali servizi sono offerti e da quali agenti.
- Gli agenti possono essere implementati come applets e possono girare su web browser.

Gli strumenti che fornisce sono:

- Il controllo dell'esecuzione degli agenti distribuiti mediante un'interfaccia che consente all'utente di:
  - far partire nuovi agenti, localmente o in remoto, come un thread o come un processo;
  - arrestare gli agenti;
  - spostare gli agenti.
- Il monitoraggio: sono mostrati tutti gli agenti in esecuzione, la loro posizione e ogni messaggio scambiato tra loro. È un valido strumento per il debug della società.
- La classe Java `MailBox`: è una classe Java che incapsula tutti i metodi necessari per avere un agente in un ambiente SACI. Per programmare un agente SACI si deve istanziare un oggetto di questa classe.

Un agente SACI è programmato seguendo questi passi:

1. **Importare il package per SACI**: all'inizio del file sorgente si inserisce la linea

   ```
   import saci.*;
   ```

2. **Definire la classe**: il modo più semplice per costruire un agente SACI è estendere la classe `saci.tools.Agent`. (Un modo alternativo consiste nell'usare la classe `MBoxSAg` [37]). La classe `Agent` implementa l'interfaccia `Launchable` che consente di lanciare un agente in remoto.

   ```
   public class NomeSACIAg extends Agent
   ```

3. **Definire il codice dell'agente**: si scrive il metodo `run` che incapsula il comportamento dell'agente.

   ```
   public void run()
   ```

   (a) `Invio di un messaggio:` si crea un oggetto `KQML Messagge` e si chiama `MBox` per inviarlo:

   ```
   Message m = new Message(
          "(performative :content \"contenuto del messaggio\""
                + " :receiver nomeAgDestinatario" +
                  " :reply-with nomeRisposta)");
      mbox.sendMsg(m);
   ```

   Il metodo `sendMsg` manda il messaggio all'agente che potrebbe anche essere in esecuzione su un'altra macchina.

   (b) `Ricezione di un messaggio` ci sono diversi metodi Java usati per recuperare messaggi che giungono all'agente (polling, receive, ask, ...). Il codice che segue usa il polling. Il metodo attende finché arriva un messaggio o finché non scade il tempo di attesa massima fissato:

   ```
   boolean ok = false;
       while (! ok) {
           Message r = mbox.polling();

           String irt = (String)r.get("in-reply-to");
           if (irt.equals("nomeRisposta")) {
               String ans = (String) r.get("content");
               showResult("Answer is " + ans);
               ok = true;
           }
       }
   ```

   Il metodo `get` di `Message` si usa per recuperare i campi del messaggio. L'agente considera solo i messaggi ricevuti in risposta al messaggio sopra guardando se il campo ‘‘`in-reply-to`’’ del messaggio ricevuto è uguale al campo ‘‘`reply-with`’’ del messaggio inviato.

4. **Definizione del metodo** `main`: se si vuole eseguire l'agente dal sistema operativo, senza l'agent launcher, si deve definire il metodo `main`:

```
public static void main(String[] args) {
    NomeSACIAg a = new NomeSACIAg();

    //Si eseguono alcune operazioni invocando metodi degli
    //agenti SACI come enterSoc, initAg e leaveSoc
    if (a.enterSoc("NomeSACIAg")) {
        a.initAg(null);
        a.run();
        a.leaveSoc();
        System.exit(0);
    }
}

void showResult(String s) {
    System.out.println(s);
}
```

5. **Cambiamento dell'agente con un applet**: passare dal codice di un agente a un applet è semplice, si deve solo:

   - estendere `AppletAgent` invece di `Agent`;
   - definire un metodo `init` invece di un metodo `main` (che serve per attivare l'agente dal sistema operativo)

     ```
     public void init() {
     if (enterSoc("NomeAppletAgent")) {
         run();
         leaveSoc();
       }
     }
     ```

   - sovrascrivere il metodo `showResult` per rendere visibile il risultato entro l'area dell'applet

     ```
     void showResult(String s) {
         add (new java.awt.Label(s));
     }
     ```

Gli agenti SACI possono essere avviati in modi differenti: attraverso una Java Virtual Machine (JVM), come applicazione Java; entro un web browser, come se fosse un'applet dentro a una pagina web; mediante un tool per l'avvio degli agenti. Questo tool è il Launcher Demon che ha come funzione l'avvio di agenti e società in sistemi distribuiti.

### 5.2.1 Sintassi di Jason

#### Agenti

La sintassi AgentSpeak(L) accettata da Jason per la definizione di un agente è definita dalla grammatica in Figura 5.1 in cui `<ATOM>` è un identificatore che comincia con una lettera minuscola o '.', `<VAR>` è un identificatore che comincia con una lettera maiuscola e indica una variabile, `<NUMBER>` è un numero intero o in virgola mobile e `<STRING>` è una stringa racchiusa tra doppie virgolette come di solito.

Le differenze principali dal linguaggio AgentSpeak(L) originale sono le seguenti. In ogni punto in cui nel linguaggio originale compariva un predicato [1] si usa un letterale (*literal*). Questo può essere un predicato $\texttt{p}(\texttt{t}_1, ..., \texttt{t}_n)$, $n \geq 0$ o $\sim \texttt{p}(\texttt{t}_1, ..., \texttt{t}_n)$, dove '$\sim$' denota la negazione forte (*strong negation*). La negazione di default (*default negation*) è usata nel contesto dei piani ed è indicata facendo precedere un letterale dal '`not`'. Il contesto (*context*) è una congiunzione di letterali di default (*default literal*). I termini possono essere atomi, strutture, variabili e liste in stile Prolog; possono essere numeri interi o in virgola mobile e stringhe racchiuse tra doppie virgolette.

Una delle differenze maggiori è che i predicati, cioè le formule atomiche, ora possono avere "annotazioni" (*annotations*). Un'annotazione è una lista di termini racchiusa tra parentesi quadrate che segue immediatamente il predicato. Entro la base dei belief (*beliefs)* le annotazioni sono usate per registrare informazioni relative alle sorgenti dei belief. Due atomi speciali, `percept` e `self`, sono usati per denotare, rispettivamente, un belief che deriva da una percezione dell'ambiente o un belief che viene aggiunto alla base dei belief dall'agente stesso per effetto dell'esecuzione del corpo di un piano. I belief iniziali che fanno parte del codice sorgente dell'agente AgentSpeak(L) sono assunti come belief interni a meno che non abbiano esplicite annotazioni date dall'utente (questo può essere utile se il programmatore vuole che l'agente abbia un belief iniziale relativo all'ambiente o comunicato da un altro agente).

I piani hanno anche etichette (*labels*). Anche se un'etichetta di piano può essere un predicato qualsiasi con annotazioni gli autori suggeriscono di usare come etichette di piano predicati di arietà zero, con annotazioni se necessarie (ad esempio `aLabel` o `aLabel[p(0.9)]`). Le annotazioni nei predicati usati per etichettare i piani possono essere usate per implementare funzioni di selezione dei piani applicabili sofisticate. Benché la corrente distribuzione di Jason non lo consenta ancora, è semplice per l'utente definire funzioni di selezione che usino qualcosa come le utilità attese annotate nelle etichette dei piani per scegliere tra piani alternativi. L'etichetta è parte di un'istanza

[1] Si ricorda che le azioni sono particolari predicati con un simbolo di azione al posto di un simbolo di predicato. Nel seguito della sezione si fa riferimento ai predicati normali e non alle azioni.

```
agent               → beliefs plans
beliefs             → ( literal ".* )*
                      N.B.: a semantic error is generated if the
                      literal was not ground.
plans               → ( plan )+
plan                → [ "@" atomic_formula ]
                      triggering_event ":" context "<-" body "."
triggering_event    → "+"     literal
                    | "-"     literal
                    | "+" "!" literal
                    | "-" "!" literal
                    | "+" "?" literal
                    | "-" "?" literal
literal             → "~" atomic_formula
                    | "~" "(" atomic_formula ")"
                    | atomic_formula
default_literal     → "not" literal
                    | "not" "(" literal ")"
                    | literal
context             → "true"
                    | default_literal ( "&" default_literal )*
body                → "true"
                    | body_formula ( ";" body_formula )*
body_formula        → literal
                    | "!" literal
                    | "?" literal
                    | "+" literal
                    | "-" literal
atomic_formula      → <ATOM> [ "(" list_of_terms ")" ]
                      [ "[" list_of_annotations "]" ]
structure           → <ATOM> "(" list_of_terms ")"
list_of_terms       → term ( "," term )*
list_of_annotations → as list_of_terms, but generating a
                      semantic error if not ground;
list                → "["
                      [ term ( ( "," term )*
                             | "|" ( list | <VAR> )
                             )
                      ] "]"
term                → structure
                    | list
                    | <ATOM>
                    | <VAR>
                    | <NUMBER>
                    | <STRING>
```

Figura 5.1: Agente Jason

di piano nell'insieme delle intenzioni e poiché le annotazioni possono essere modificate dinamicamente, questa é anche un mezzo per l'implementazione di funzioni di selezione delle intenzioni efficienti.

Si noti che per un agente che assuma di lavorare con la Closed-World Assumption tutto ciò che l'utente deve fare è evitare l'uso di letterali con negazione forte in ogni punto del programma incluse le percezioni negative nell'ambiente.

Infine, ci sono *azioni interne* che possono essere usate sia nel contesto, sia nel corpo degli agenti. Ogni simbolo di azione che comincia con '.' o contiene '.' indica un'azione interna. Queste sono azioni definte dall'utente e l'agente le esegue internamente. Sono dette "interne" per distinguerle da quelle azioni che appaiono nel corpo di un piano e che indicano azioni che l'agente compie per cambiare l'ambiente condiviso. In Jason le azioni interne sono definite dall'utente in Java. Con la distribuzione di Jason è fornita una libreria standard (`stdlib`) di azioni interne che sono usate dagli agenti per inviare messaggi (`.send`, `.broadcast`) e per stamparli (`.print`), per operare su desideri e intenzioni (`.desire`, `.intend`, `.dropDesire`, `.dropIntention`, `.dropAllDesires`, `.dropAllIntentios`) e per eseguire operazioni relazionali e aritmetiche (`.equals`, `.gt`, `.gte`, `.plus`).

### MAS

Un sistema di agenti AgentSpeak(L) multipli è costituito da un insieme di istanze di agenti AgentSpeak(L) e da un ambiente in cui sono situati gli agenti. L'ambiente viene programmato in Java. La configurazione dell'intero sistema multi-agente è data in un file di testo la cui sintassi è descritta dalla grammatica BNF in Figura 5.2. Il file di configurazione ha estensione `.mas2j`. `<NUMBER>` è usato per i numeri interi, `<ASID>` sono gli identificatori degli agenti AgentSpeak(L) che devono avere la prima lettera minuscola, `<ID>` è un identificatore e `<PATH>` è usato per definire i percorsi per i file entro il sistema operativo.

L'identificatore `<ID>` usato dopo la parola chiave `MAS` è il nome della società; questo è usato, tra le altre cose, nel nome degli script che sono generati automaticamente per aiutare l'utente a compilare ed eseguire il sistema. La parola chiave `architecture` è usata per specificare quale delle due architetture messe a disposizione dalla distribuzione di Jason sarà usata. Le opzioni disponibili sono `Centralised` o `Saci`; l'ultima è il default e consente di eseguire gli agenti su macchine diverse in rete. Questa indicazione è necessaria per decidere quali classi usare per definire l'architettura dell'agente, per gestire le percezioni dall'ambiente al momento dell'istanziazione dell'agente e, in seguito, per eseguire gli agenti e gestire la comunicazione tra essi.

La parola chiave `environment` è un riferimento al nome della classe Java che si usa per programmare l'ambiente. Si può specificare il nome di un host

```
mas         → "MAS" <ID> "{"
                [ "architecture" ":" <ID> ]
                environment
                agents
              "}"
environment → "environment" ":" <ID> [ "at" <ID> ]
agents      → "agents" ":" ( agent )+
agent       → <ASID>
                [ filename ]
                [ options ]
                [ "agentArchClass" <ID> ]
                [ "agentClass" <ID> ]
                [ "#" <NUMBER> ]
                [ "at" <ID> ]
                ";"
filename    → [ <PATH> ] <ID>
options     → "[" option ( "," option )* "]"
option      → "events" "=" ( "discard" | "requeue" | "retrieve" )
            | "intBels" "=" ( "sameFocus" | "newFocus" )
            | "verbose" "=" <NUMBER>
```

Figura 5.2: MAS Jason

su cui girerà l'ambiente ma solo se l'architettura sottostante usata è quella basata su SACI.

La parola `agents` si usa per definire l'insieme degli agenti che faranno parte del sistema multi-agente. Un agente è specificato dal suo nome simbolico dato come termine AgentSpeak(L) (di solito un identificatore che comincia con la lettera minuscola); questo è il nome che gli agenti useranno per riferirsi gli uni agli altri nella società (ad esempio per la comunicazione tra agenti). Con un'indicazione opzionale si può fornire un nome di file (comprendente il percorso completo se non è nella stessa directory del file di configurazione) contenente il codice sorgente AgentSpeak(L) per l'agente. Per default Jason assume che il codice sorgente AgentSpeak(L) di un agente sia nel file `<nome>.asl` in cui `<nome>` è il nome simbolico dell'agente. C'è anche una lista di assegnazioni opzionali usate per configurare l'interprete per AgentSpeak(L) disponibile con Jason. Con il simbolo `#` seguito da un numero si indica un numero opzionale di istanze di agenti che usano lo stesso codice sorgente; se questo numero è presente, nel sistema multi-agente sono creati tanti "cloni" quanto è il numero specificato. Nel caso in cui sia richiesta più di un'istanza di un agente, il nome dell'agente clone sarà il nome simbolico concatenato con un indice che indichi il numero dell'istanza (partendo da 1). Analogamente a quanto avviene per l'ambiente, una definizione di agente, preceduta da "`at`" può finire con il nome di un host

su cui l'agente sarà eseguito. Anche in questo caso questa possibilità si ha solo se si sceglie di usare l'architettura basata su SACI.

La lista di assegnazioni citata prima è costituita da campi seguiti da '=' e poi da una parola chiave che indica il valore assunto.

`events:` le opzioni sono `discard` che indica che gli eventi esterni per cui non ci sono piani applicabili sono cancellati (e un avvertimento viene stampato sulla console su cui girano il sistema o SACI), `requeue` è usata quando l'evento può essere reinserito alla fine della lista degli eventi che l'agente deve gestire. Dalla versione 0.4 è disponibile l'opzione `retrieve`; quando si seleziona tale opzione la funzione `selectOption` viene chiamata anche se l'insieme dei piani rilevanti/applicabili è vuoto. Questa sarà usata, tra l'altro, per consentire agli agenti di chiedere piani ad altri agenti che possono avere conoscenze necessarie di cui l'agente al momento non è in possesso. Il default è `discard`.

`intBels:` le opzioni sono `sameFocus` e `newFocus`. Quando i belief interni sono aggiunti o rimossi esplicitamente entro il corpo di un piano, se l'evento associato è un triggering event per un piano, allora l'istanza di piano (*intended means)* risultante dal piano applicabile scelto per tale evento viene messo in cima all'intenzione (fuoco di attenzione) che ha generato l'evento, oppure lo si può trattare come se fosse un evento esterno (come aggiunta o rimozione di belief dalle percezioni dell'ambiente), creando un nuovo fuoco di attenzione. Questa opzione ritenuta utile è lasciata come un'opzione selezionabile dall'utente in quanto non è considerata nella versione originale del linguaggio.
Usando `newFocus`, ad esempio, l'utente può creare diverse intenzioni per un solo evento esterno le quali competeranno per avere l'attenzione dell'agente. Il default è `sameFocus`.

`verbose:` si specifica un numero tra 0 e 6. Più alto è il numero più sono numerose le informazioni circa l'agente (o gli agenti se ci sono più istanze) che vengono stampate sulla console di Jason. Il default è 1 e permette di stampare solo le azioni che l'agente compie e i messaggi che questi si scambiano.

Infine, con le parole chiave `agentArchClass` e `agentClass` l'utente può spercificare l'architettura per l'agente e le funzioni di selezione usate dall'interprete AgentSpeak(L) per ogni agente particolare. Jason è molto flessibile poiché consente agli utenti di ridefinire le funzioni di default usate nell'interprete.

### 5.2.2 Sistema run-time di Jason

L'interprete di Jason è costituito da un motore (engine) che determina le transizioni tra gli stati del sistema in esecuzione. Uno stato del sistema

è costituito da diverse componenti. Le componenti citate sono in parte elementi dell'agente come l'insieme dei belief e quello dei piani, le funzioni di selezione e di fiducia mentre per il resto queste componenti sono date dalle circostanze e dall'ambiente specificato nel file di configurazione del MAS.

Una *circostanza*, formalmente definita come una tupla, non è altro che una classe Java rappresentante un insieme eterogeneo di strutture dati adatte per la memorizzazione di vari elementi quali:

- la coda degli eventi o event queue;
- l'insieme delle intenzioni attive;
- l'insieme dei messaggi ricevuti, o mailbox;
- l'insieme dei piani rilevanti;
- l'insieme dei piani applicabili;
- l'azione correntemente in esecuzione;
- l'evento scelto per essere trattato;
- l'intenzione correntemente in esecuzione;
- l'opzione scelta per l'esecuzione, costituita da una coppia i cui elementi sono un piano e l'unificatore che ne determina l'applicabilità;
- una lista di azioni eseguite con l'indicazione del risultato di tale esecuzione (insieme dei feedback delle azioni);
- l'insieme delle azioni pendenti che sono state mandate in esecuzione e per le quali si attende un feedback (viene utilizzato quando l'architettura usata è quella basata su Saci).

La coda degli eventi e gli insiemi indicati sopra, ad eccezione di quello delle azioni pendenti, sono implementati con delle liste e inserimenti e cancellazioni sono gestite con il criterio FIRST IN-FIRST OUT.

Un *evento* è definito da una coppia triggering event e intenzione associata.

Un *ambiente* è definito mediante una classe Java che estende la classe `Environment`. Il nome di questa classe va indicato nel file di configurazione del MAS come valore per `environment`. La definizione dell'ambiente prevede l'istanziazione delle percezioni iniziali positive dell'agente e, nel caso della *Open-World Assumption*, anche di quelle negative. Il codice per la costruzione dell'ambiente viene definito con il metodo `executeAction`. Questo metodo definisce anche come eseguire le azioni degli agenti facenti parte del MAS.

A proposito di azioni, si evidenzia il fatto che ne esistano di due tipi: *azioni "di base"* [2] (che modificano lo stato dell'ambiente) e *azioni interne*. Quando nel corpo di un agente si incontra un'azione interna, la si manda in esecuzione richiamando il metodo `execute` entro la classe Java che definisce l'azione interna stessa. Quando invece un'agente tenta di eseguire un'azione "di base" tale azione è assegnata all'opportuna struttura della circostanza. L'azione selezionata nella circostanza viene mandata in esecuzione in modo asincro e il risultato di tale esecuzione, una volta pervenuto, viene memorizzato con l'azione nell'insieme dei feedback delle azioni. Si noti che quando si attende il risultato di un'azione l'intenzione relativa viene sospesa, analogamente a quanto avviene per un evento interno. Solo dopo aver ottenuto il feedback per l'azione, l'intenzione torna ad essere attiva, mediante reinserimento nell'insieme delle intenzioni attive, e può essere scelta per l'esecuzione dall'apposita funzione di selezione.

Il metodo `executeAction` ha come parametri il nome dell'agente e il termine che rappresenta l'azione scelta per l'esecuzione. L'esecuzione del metodo inizia con il controllo degli argomenti e prosegue facendo quanto necessario per completare l'azione nel particolare modello d'ambiente correntemente disponibile. `executeAction` restituisce un valore di tipo `boolean` che stabilisce se il tentativo di esecuzione dell'azione nell'ambiente ha avuto successo o meno. Un piano fallisce se una qualsiasi azione di base tentata dall'agente fallisce.

Il *motore* vero e proprio è definitio mediante la classe Java `TransitionSystem`. Questa classe contiene riferimenti all'agente in esecuzione, alla circostanza corrente, alle opzioni (`settings`), all'architettura del MAS e ad un'istanza del motore stesso. Il motore in esecuzione è interamente guidato dal metodo `reasoningCycle` e porta a termine diverse attività.

- Si attende che ci siano elementi quali messaggi, eventi, intenzioni o azioni disponibili per essere processati;
- Si aggiornano le percezioni provenienti dall'ambiente del MAS e, nel caso di architettura basata su SACI, si recuperano i risultati di eventuali azioni pendenti.
- Si istanzia la mailbox entro la circostanza.
- Si aggiorna la base dei belief dell'agente cancellando i belief annotati con `percept` che non sono inclusi nelle percezioni e aggiungendo le percezioni non ancora presenti.
- Si avvia il processo di applicazione delle regole a partire dalla lettura di un messaggio.

[2] Nel seguito le azioni di base saranno indicate semplicemente come azioni, mentre le azioni interne saranno indicate espressamente.

- Si manda in esecuzione l'azione scelta correntemente nella circostanza.

Le operazioni di aggiornamento delle percezioni e della base dei belief, di istanziazione della mailbox e di esecuzione dell'azione corrente sono influenzate dall'architettura su cui si basa il MAS. In particolare, mentre nell'esecuzione dell'azione corrente entro un MAS con architettura centralizzata l'azione completata viene messa direttamente nell'insieme dei feedback, nel caso di architettura basata su SACI l'azione va prima posta in un'apposita struttura per la memorizzazione dell'azione pendente in attesa del risultato.

Il *processo di applicazione delle regole* è articolato in modo tale che si parte processando un messaggio prelevato dalla mailbox. Il tipo del messaggio determina il tipo di attività compiuta su di esso, ma indipendentemente da esso il flusso del processo continua con la selezione di un evento.

Se la coda degli eventi è non vuota si sceglie un evento e si cercano i piani rilevanti rispetto al suo triggering event.
Se la coda degli eventi è vuota si procede processando un'azione. Con "processare un'azione" s'intende selezionare un'azione con il proprio risultato e procedere nella gestione di evento e azione corrente a seconda del valore di quest'ultimo.

Se ci sono piani rilevanti, o l'opzione per la gestione degli eventi del MAS è `retrieve`, si cercano i piani applicabili.
Se non ci sono piani rilevanti e l'opzione è `discard` o `requeue` si procede in modo diverso a seconda del tipo di evento:

- se è un evento avente come triggering event l'aggiunta di un goal si aggiunge alla coda degli eventi un evento avente come triggering event la cancellazione dello stesso goal e si visualizza un avvertimento per l'utente;

- se è un evento interno si seleziona l'intenzione ad esso associata come intenzione corrente e la si aggiorna (rimuovendo l'elemento corrente dal corpo dell'istanza di piano in cima ad essa);

- se è un evento esterno e l'opzione per la gestione degli eventi del MAS è `requeue` lo si reinserisce nella coda degli eventi.

Indipendentemente dal tipo di evento non gestibile si procede processando un'azione.

Se ci sono piani applicabili oppure l'opzione per la gestione degli eventi del MAS è `retrieve` si procede scegliendo tra i piani applicabili un'istanza di piano (`Option`) per l'esecuzione.
Se non ci sono piani applicabili e l'opzione per la gestione degli eventi è

`requeue` o `discard` si richiede la rimozione dell'evento corrente procedendo in modo diverso a seconda del tipo dell'evento:

- se è un evento interno il cui triggering event è l'aggiunta di un goal si aggiunge alla coda degli eventi un evento avente come triggering event la cancellazione dello stesso goal e si visualizza un avvertimento per l'utente;

- se è un evento interno lo si elimina e si visualizza un avvertimento per l'utente;

- se è un evento esterno si considera l'opzione per la gestione degli eventi del MAS e se questa è `requeue` lo si reinserisce nella coda degli eventi, altrimenti lo si elimina e si visualizza un avvertimento per l'utente.

Indipendentemente dal tipo di evento non gestibile si procede processando un'azione.

Se non si è trovata un'istanza di piano applicabile per l'esecuzione si procede visualizzando un avvertimento per l'utente e richiedendo la rimozione dell'evento come nel caso precedente.
Se si è trovata un'istanza di piano applicabile si mette tale istanza in cima allo stack dell'intenzione associata all'evento, la quale è vuota se questo è di tipo esterno. Si procede quindi con la scelta di un'azione da processare.

La `selectAction` seleziona un'azione con il relativo risultato dall'insieme dei feedback delle azioni. Se il risultato dell'azione scelta è negativo si procede aggiornando l'intenzione, altrimenti si procede alla rimozione dell'evento corrente come visto sopra. Indipendentemente dal risultato il processo di applicazione delle regole continua con la rimozione dell'intenzione, se completata. Se non ci sono azioni da processare si procede con la selezione di un'intenzione.

Se non ci sono intenzioni attive tra cui scegliere quella da processare il processo di applicazione delle regole si arresta, altrimenti si sceglie un'intenzione che diviene l'intenzione corrente entro la circostanza e si procede con l'esecuzione.

L'intenzione selezionata viene processata prelevando l'istanza di piano in cima allo stack dell'intenzione. Se il corpo di tale piano è vuoto si aggiorna l'intenzione, altrimenti si prende il primo elemento del corpo del piano e a seconda del suo tipo lo si processa come segue:

- se è un'*azione* si guarda se è interna: in caso affermativo la si esegue (si richiama il metodo `execute` entro la classe che definisce l'azione

interna), poi si aggiorna l'intenzione, altrimenti si istanzia con essa l'azione da eseguire correntemente nella circostanza.

- se è un *achivement goal* si inserisce nella coda degli eventi un evento interno avente come triggering event tale goal istanziato opportunamente e come intenzione quella correntemente selezionata.

- se è un *test goal* si guarda se esiste un unificatore per cui il test risulti vero. Se tale unificatore esiste si sostituisce l'unificatore dell'istanza di piano con quello appena determinato, poi si aggiorna l'intenzione. Se l'unificatore non esiste si visualizza un avvertimento per l'utente e si inserisce nella coda degli eventi un evento interno avente come triggering event il goal opportunamente istanziato e come intenzione quella correntemente selezionata.

- se è un'*aggiunta di belief* si istanzia opportunamente il belief, lo si aggiunge alla base dei belief e poi si procede inserendo nella coda degli eventi un nuovo evento con l'aggiunta di belief come triggering event e con l'intenzione che cambia a seconda che il valore dell'opzione per la gestione dei belief interni sia `sameFocus` o `newFocus`. Nel primo caso l'intenzione scelta è quella correntemente selezionata, nel secondo è una nuova intenzione vuota. In seguito a tale inserimento si procede con l'aggiornamento dell'intenzione corrente.

- se è una *rimozione di belief* si determina l'unificatore che verifica la consistenza del belief e se tale unificatore non esiste si rimuove il trigger event del piano che ha determinato la rimozione. La rimozione dipende dal tipo del trigger event: se è l'aggiunta di un goal si aggiunge alla coda degli eventi un evento avente come triggering event la cancellazione dello stesso goal. Se il triggering event è di altro tipo, a seconda che l'opzione per la gestione degli eventi del MAS sia `requeue`, o meno, lo si reinserisce nella coda degli eventi come evento esterno, o si visualizza un avvertimento per l'utente. Se l'unificatore esiste ci si comporta analogamente al caso di aggiunta del belief visto sopra, con la differenza che il belief viene rimosso dalla base anziché aggiunto.

Una volta che l'elemento è stato processato si procede all'eventuale rimozione dell'intenzione.

La rimozione dell'intenzione è subordinata al suo completamento. Se il corpo dell'istanza di piano in cima all'intenzione è vuoto e l'intenzione è vuota questa viene rimossa dall'insieme delle intenzioni attive e il processo di applicazione delle regole si arresta. Se invece l'intenzione contiene ancora delle istanze di piano, si rimuove quello appena completato, si seleziona il successivo dal cui corpo si preleva un elemento e si aggiorna l'unificatore dell'istanza di piano corrente componendolo con quello dell'istanza di piano

appena rimossa. A questo punto si procede controllando se ora l'intenzione può essere rimossa. Se il corpo dell'istanza di piano non è vuoto il processo di applicazione delle regole si arresta.

Si è notato più volte che se un'azione fallisce o non c'è un piano applicabile per un evento interno del tipo aggiunta di un goal, allora si genera un evento interno per la cancellazione di un goal associato alla stessa intenzione che aveva generato quest'ultimo evento interno. Se il programmatore fornisce un piano che ha un trigger event unificabile con un triggering event per la cancellazione di un goal che è applicabile, questo piano viene messo in cima all'intenzione e il programmatore può specificare nel corpo di tale piano come gestire questo particolare fallimento. Dalla descrizione data nel manule di Jason risulta che se tale piano non è disponibile l'intera intenzione viene rimossa e sulla console viene stampato un avvertimento. Questo meccanismo fa si che in Jason siano disponibili degli eventi, e conseguentemente dei piani, per la gestione del fallimento di un piano anche se non sono ancora formalizzati nella semantica.

### 5.2.3 Agenti, architetture, azioni interne e ambienti definibili dall'utente

Come visto prima, nell'interprete di Jason un agente è costituito da un insieme di belief, da un insieme di piani, da alcune funzioni di selezione e di fiducia e da una circostanza che include eventi e intenzioni pendenti e varie altre stutture che servono durante l'interpretazione di un agente. Molti di questi elementi costitutivi dell'agente possono essere modificati dall'utente per ottenere particolari comportamenti (come descritto nel Capitolo 5 del manuale di Jason [8]).

#### Agenti

Le modifiche all'agente riguardano la ridefinizione delle funzioni di fiducia (`acceptAchieve`, `acceptTell`) e di selezione (`selectEvent`, `selectOption`, `selectIntention`, `selectMessage` e `selectAction`). Ridefinendo queste funzioni l'agente può, ad esempio: dare precedenza a certi tipi di messaggi provenienti da determinati agenti; scegliere una determinata azione dall'insieme delle azioni compiute per cui si è ottenuto un feedback.

Per poter ridefinire il comportamento delle funzioni si definisce una nuova classe che estende la classe `Agent` e nel file di configurazione si indica il nome di tale classe con `agentClass`.

#### Architetture

Un'architettura fornisce alcuni metodi che si occupano di:

- gestire le percezioni dall'ambiente (metodo `perceive`). Il modo di percepire può essere modificato in modo che gli agenti possano percepire cose diverse da uno stesso ambiente o simulare percezioni erronee.

- revisionare i belief (metodo `brf`) generando eventi esterni a seconda delle percezioni correnti, senza però garantire la consistenza di tali belief, cosa che è di competenza dell'utente.

- decidere come vengono ricevuti i messaggi provenienti da altri agenti (metodo `checkMail`).

- decidere come l'agente deve agire entro l'ambiente (metodo `act`).

Per ridefinire tali funzioni si definisce una classe che estenda la classe `CentralisedAgArch`, per le architetture centralizzate, o la classe `SaciAgArch`, per le architetture basate su SACI, e si ridefinisce il comportamento dei metodi in modo opportuno. Nel file di configurazione del MAS si indica il nome di tale classe in corrispondenza di `agentArchClass`.

### Azioni interne

Le azioni interne sono organizzate in librerie. L'azione è riferita mediante il nome della libreria seguito da '.' seguito dal nome dell'azione. Le librerie sono sottodirectory di `\src\jason` (quella standard si trova in `\src\jason\stdlib`) e il loro nome deve iniziare con la lettera minuscola.

Ogni azione è definita in codice Java, in un file con estensione "`.java`" entro la directory della libreria relativa e il nome del file (che è lo stesso della classe `public` in esso contenuta) deve coincidere con quello dell'azione. Entro il file va indicato che l'azione appartiene ad un `package` avente lo stesso nome della libreria.

L'azione interna viene eseguita mediante l'invocazione del metodo `execute` che ne definisce il comportamento. Il terzo argomento di tale metodo è una array di `String` usato per il passaggio dei parametri dell'azione. Gli altri argomenti di `execute` sono uno di tipo `TransitionSystem`, contenente tutte le informazioni sullo stato corrente dell'agente, l'altro di tipo `Unifier`. Questo corrisponde alla funzione di unificazione determinata dall'esecuzione o dal controllo dell'applicabilità di un piano. Tale funzione è importante nei casi in cui i valori delle variabili sono necessari per l'implementazione dell'azione.

### Ambienti

Relativamente all'ambiente, le modifiche effettuabili dall'utente riguardano le percezioni individuali dell'agente entro l'ambiente stesso e sono ottenute

ridefinendo i metodi `getPercepts` e `getNegativePercepts`. Il comportamento di default è quello che rende accessibili tutte le proprietà correnti dell'ambiente in qualità di percezioni a tutti gli agenti. La modifica di queste funzioni dell'ambiente fa in modo che un singolo agente possa percepire solo un sottoinsieme delle proprietà dell'ambiente.

## 5.3 Coo-AgentSpeak: come realizzarlo con Jason

Per realizzare l'interprete per Coo-AgentSpeak si sfruttano alcune caratteristiche di Jason funzionali all'implementazione dello scambio di piani. Il progetto per la realizzazione dell'interprete prevede il completamento di alcuni compiti principali:

1. nella specifica di ogni agente si devono includere tre funzioni per definire la *strategia di cooperazione*, analogamente a quanto avviene per le funzioni `trust` e `power` in [54];

2. si devono includere nei piani gli specificatori d'accesso e le sorgenti;

3. si deve tenere traccia degli eventi associati alle intenzioni che vengono sospese in attesa del completamento di un recupero di piani esterni;

4. si deve modificare l'interprete di Jason per rendere il meccanismo di recupero dei piani esterni trasparente all'utente, come previsto da Coo-BDI.

Questi compiti possono essere portati a termine in diversi modi. Una proposta su come procedere è stata formulata nell'articolo [4]. Questa forniva una serie di indicazioni preliminari da cui ci si è talvolta discostati nel corso della progettazione dell'implementazione vera e propria.

Nel seguito sono indicate le scelte fatte. Quando la proposta data in [4] e l'implementazione effettiva non coincidono, viene sottolineata questa discrepanza e ne viene fornita una motivazione. Si evidenziano le parti relative alle diverse proposte, iniziale e realizzata, ponendo all'inizio del relativo paragrafo indicazioni del tipo:

- `PROPOSTA INIZIALE` quando si descrivono le indicazioni prese in [4];

- `IMPLEMENTAZIONE` quanod si descrive quanto realizzato nell'implementazione;

- `PROPOSTA INIZIALE` $\equiv$ `IMPLEMENTAZIONE` quando le proposte coincidono.

Nel capitolo successivo verrano forniti dettagli e ulteriori motivazioni a supporto di tali scelte.

### Strategia di cooperazione

`PROPOSTA INIZIALE` ≡ `IMPLEMENTAZIONE:` la stategia di cooperazione in Coo-AgentSpeak può essere espressa facilmente con un insieme di "predicati di sistema" (riservati) entro la base dei belief. I predicati di sistema sono:

- `cooAS_planSources(`$Te$`,[`$Id_1$`,...,`$Id_n$`])`,
- `cooAS_retrievalPolicy(`$Te$`,`$Retrieval$`)` e
- `cooAS_acquisitionPolicy(`$Te$`,`$Acquisition$`)`,

e corrispondono a quelli introdotti nella Sezione 5.1.

### Piani

Jason fornisce il mezzo per includere nelle etichette di piano annotazioni arbitrarie e Coo-AgentSpeak sfrutta la struttura dei piani di Jason senza bisogno di modifiche. Si adottano le etichette di piano annotate per specificare entro le annotazioni una struttura `cooAS` di arietà uno; l'argomento di `cooAS` è una lista che contiene due termini di arietà uno: `accSpec` e `source`.

`PROPOSTA INIZIALE:` l'argomento di `accSpec` può assumere valori in {`private, public, only(TrustedAgentSet)`}. L'argomento di `source` può assumere valori in {`self,`$id$}.

`IMPLEMENTAZIONE:` si prevede che `source` abbia come argomento l'identificatore dell'agente "proprietario" del piano oppure, in alcuni passaggi del meccanismo di scambio dei piani, l'identificatore univoco della richiesta di recupero di piani esterni.

### Intenzioni

Si deve tenere traccia delle intenzioni che sono in attesa dell'arrivo di un certo piano da un altro agente.

`PROPOSTA INIZIALE:` un modo per implementare questo in Jason è usare una struttura ad hoc che viene memorizzata nella base dei belief. Il vantaggio di avere queste informazioni nella base dei belief è che gli agenti possono cambiare le informazioni Coo-AgentSpeak dinamicamente (ad esempio eseguendo delle istruzioni interne speciali definite dall'utente da determinate istanze di piano).

Un'intenzione viene associata con una richiesta di piani esterni usando belief di sistema della forma:

```
cooAS_suspendedInt(MsgId, Ev, Ags, Plans).
```

Belief di questo tipo dovrebbero essere usati per aggiornare l'insieme `Plans` dei piani esterni recuperati per l'intenzione identificata entro `Ev`, quando l'agente riceve un messaggio in risposta alla richiesta di piani con identificatore `MsgId`; `Ags` viene istanziata con l'insieme di tutti gli agenti cui è stata inviata la richiesta.

`IMPLEMENTAZIONE:` durante l'implementazione ci si è resi conto che la rappresentazione dell'intenzione entro il belief non è semplice. Un belief deve essere ground per poter essere inserito nella base dei belief e per ottenere questo l'evento, costituito da un triggering event e da un'intenzione, va codificato con una stringa, ma il passaggio inverso, da stringa a evento non è banale.

Inoltre ci sono azioni interne che effettuano delle ricerche sulle intenzioni presenti nel sistema le quali non prevedono l'esistenza di strutture di memorizzazione delle intenzioni diverse da quelle previste da Jason. Memorizzando l'intenzione nel belief questa è inaccessibile a queste azioni la cui esecuzione potrebbe quindi portare a risultati errati. Per questi motivi si è scelta un'alta strada.

Le intenzioni sospese rimangono all'interno dell'insieme delle intenzioni selezionabili e le informazioni relative alla richiesta vengono inserite in cima allo stack dell'intenzione. Sono identificabili rispetto a quelle attive perché in cima allo stack hanno un'istanza di piano particolare.

### Interprete

Gli aspetti relativi all'interprete sono trattati seguendo i quattro passi definiti in Coo-BDI.

1. **Gestione della mailbox**

   `PROPOSTA INIZIALE:` ci sono due tipi di messaggi che devono essere processati prima degli altri: le richieste di piani e le risposte a tali richieste. Il metodo `selectMessage` fornito da Jason è modificato in modo da dare priorità a questi tipi di messaggi; tra i due la precedenza deve essere data alle risposte ricevute per le richieste di piani dell'agente.

   `IMPLEMENTAZIONE:` si è deciso di non dare priorità ad alcun messaggio pur riconoscendo che certi messaggi legati alla richiesta di piani vanno gestiti in modo particolare.

   Si considerano due casi.

   (a) `PROPOSTA INIZIALE:` quando si seleziona per la gestione un messaggio di tipo ⟨`achieve, Ag, cooAS_sendPlansFor(Te)`⟩, l'evento speciale `+!cooAS_sendPlanFor(Te, Ag)` è aggiunto alla event

queue del ricevente.

`IMPLEMENTAZIONE:` si procede come sopra ma si è scelto di indicare entro al predicato `cooAS_sendPlanFor` anche l'identificatore dell'agente che fa la richiesta e l'identificatore della richiesta, per cui il messaggio selezionato sarà del tipo ⟨`achieve, Ag, cooAS_sendPlansFor(Te, MsgId, Ag)`⟩. `Ag` è il mittente della richiesta e `MsgId` è l'identificatore che individua in modo univoco la richiesta.

(b) `PROPOSTA INIZIALE:` quando si seleziona un messaggio di tipo ⟨`tellHow, Ag, P`⟩ in risposta ad un messaggio identificato da `MsgId`, l'informazione sull'intenzione sospesa associata con tale `MsgId` è aggiornata nella base dei belief. L'agente `Ag` è rimosso dall'insieme `Ags` degli agenti da cui si attende una risposta e tutti i piani in `P` sono aggiunti a `Plans` entro a tale predicato. Quando tutti gli agenti hanno risposto l'intenzione può essere ripristinata, selezionando un piano da `Plans`; prima di questo, la libreria dei piani è aggiornata secondo la `acquisitionPolicy` specificata per i triggering event di tutti i piani ricevuti.

`IMPLEMENTAZIONE:` quanto descritto sopra si discosta molto da quanto realizzato effettivamente. L'unico aggiornamento che si ha nelle informazioni relative alla sospensione dell'intenzione riguarda la lista degli agenti da cui si attendono risposte. I piani ottenuti in risposta non sono raccolti in strutture ausiliarie ma sono introdotti direttamente nella libreria dei piani e contengono entro l'annotazione l'identificatore della richiesta in base alla quale sono stati scambiati. Il ripristino dell'intenzione sospesa e le attività che ne conseguono sono trattate in un altro momento.

2. **Gestione della event queue.**

Si considerano due situazioni.

(a) `PROPOSTA INIZIALE:` si seleziona l'evento speciale generato dalla ricezione di una richiesta di piano. Il metodo `selectEvent` fornito da Jason viene modificato in modo da scegliere sempre questo evento prima (solo l'evento `!startExtenalPlanRetrieval(Te)` ha priorità più alta). Coo-AgentSpeak fornisce un piano di sistema con `+!cooAS_sendPlanFor(Te, Ag)` come triggering event; con questo piano l'agente cerca tutti i piani rilevanti per `Te` (lo fa con un'azione interna implementata appositamente per tale scopo, usando l'algoritmo di unificazione presente in Jason per controllare quali piani nella libreria dei piani siano rilevanti) ed

esegue un'azione `.send(Ag, tellHow, P)`, dove `P` è l'insieme dei piani recuperati.

`IMPLEMENTAZIONE:` si è scelto di non dare priorità agli eventi. Il piano di sistema ha triggering event `+!cooAS_sendPlanFor(Te, MsgId, Ag)` e si comporta in modo differente. Una volta recuperati i piani, questi vengono annotati con l'informazione dell'identificatore della richiesta, ma non sono inviati con un solo messaggio di tipo TellHow entro una lista. I piani sono trasmessi con un sequenza di messaggi di tipo TellHow, delimitata da due messaggi di tipo Tell in cui si usa ancora l'identificatore della richiesta per indicare a quale intenzione sospesa si riferisce la trasmissione.

(b) Se si sceglie un evento normale `Te` si possono verificare due casi. Se la strategia di recupero per `Te` è `noLocal` e ci sono piani rilevanti e applicabili per `Te`, o non ci sono agenti fidati per `Te`, allora non vengono emesse richieste di piani esterni e la computazione continua come nell'implementazione di AgentSpeak. Altrimenti, vanno recuperati i piani esterni. L'emissione delle richieste di recupero di piani rilevanti all'esterno avviene come segue.

`PROPOSTA INIZIALE:` si genera un nuovo evento interno `!startExtenalPlanRetrieval(Te)` e il metodo `selectEvent` viene modificato in modo da scegliere sempre tale evento per primo. Oltre a questo, si ha un piano di sistema, che va incluso nella libreria dei piani dell'agente, avente un triggering event coincidente con l'aggiunta di goal `+!startExtenalPlanRetrieval(Te)`, il predicato `cooAS_planSources(Te, AgList)` come contesto e un corpo entro cui eseguire l'azione `.send(AgList, achieve, cooAS_sendPlansFor(Te))`.

`IMPLEMENTAZIONE:` non si genera l'evento prioritario che serve come triggering event per il piano di sistema che avvia la richiesta ma si fa in modo che la `selectOption` renda direttamente tale piano e che si occupi della sospensione dell'intenzione e della gestione delle informazioni ad essa relative. Il contenuto del piano invece resta pressoché invariato.

3. **Gestione delle intenzioni sospese**

`PROPOSTA INIZIALE:` è completata contestualmente alla gestione dei messaggi come descritto in precedenza. Per questo passo di Coo-BDI non era prevista nessuna particolare modifica per l'implementazione con Jason.

`IMPLEMENTAZIONE:` le intenzioni sospese sono ripristinate all'interno della `selectIntention` che si occupa di scegliere un'intenzione sospesa pronta per il ripristino, determina quale istanza di piano mandare in esecuzione in cima all'intenzione e gestisce l'acquisizione dei piani recuperati secondo la politica indicata.

4. **Gestione delle intenzioni attive**

   `PROPOSTA INIZIALE` $\equiv$ `IMPLEMENTAZIONE:` questo passo è gestito seguendo le regole semantiche di AgentSpeak originale.

Qui sono state brevemente accennate alcune scelte implementative che saranno dettagliate, giustificate e messe eventualmente a confronto con possibili alternative nel capitolo successivo.

# Capitolo 6

# Realizzazione di Coo-AgentSpeak

Quello che si descrive di seguito sono le fasi seguite per la progettazione e la realizzazione di un prototipo dell'interprete per Coo-AgentSpeak realizzato in Jason. La costruzione dell'interprete è subordinata al vincolo per cui l'estensione che consente l'integrazione dell'idea Coo-BDI in Jason deve essere fornita come una patch all'attuale versione 0.5 di Jason in modo che anche una volta installato il paccheto per l'estensione cooperativa, i MAS programmati secondo il modello BDI originale continuino a funzionare (conservatività).

Ci si è impegnati per fare in modo che l'estensione dell'interprete si ottenga con modifiche al codice che interessino solo quelle parti dell'interprete originale predisposte per la ridefinizione del comportamento di un agente da parte dell'utente e con azioni interne nuove definite in un package apposito.

Nel seguito verrà prentata la fase preliminare alla progettazione in cui ci si è occuppati di studiare e testare il funzionamento dell'interprete Jason. I problemi rilevati in questa fase preliminare sono stati riportati agli autori di Jason i quali hanno integrato i suggerimenti proposti per risolverli nella versione corrente.

Segue la descrizione delle scelte progettuali ad alto livello che vengono motivate e talvolta messe a confronto con possibili alternative.

## 6.1 Studio e test sul funzionamento di Jason

Lo studio del funzionamento dell'interprete è cominciato con la versione 0.2 del gennaio 2004, la prima ad essere resa pubblica. Tale versione era eseguibile solo sul sistema operativo Linux e aveva un'interfaccia a linea di comando. Nell'aprile 2004 è stata pubblicata la versione 0.3, eseguibile anche su sistemi Windows, che introduce un'interfaccia utente di tipo grafico. A luglio 2004 è uscita la versione 0.4 nella quale l'interfaccia grafica è stata

migliorata, è stata introdotta l'opzione `retrieve` per la gestione degli eventi in assenza dei piani rilevanti o applicabili corrispondenti ed è cambiato il modo in cui si specifica l'ambiente definibile dall'utente. Con la versione 0.4 l'utente può decidere come saranno le percezioni per ogni agente.

È con la versione 0.4 che si cominciano i primi test con esempi scritti in Coo-AgentSpeak. Da questi test emergono alcuni problemi importanti generati nell'esecuzione di meccanismi cardine rispetto all'implementazione dello scambio di piani.

Il primo errore che abbiamo riscontrato è di tipo sintattico ed è legato alla scansione dei piani etichettati che sono un elemento indispensabile per la definizione di agenti Coo-AgentSpeak. Quando il primo piano che compare nella libreria dei piani è etichettatto, il parser non si accorge della fine della base dei belief dell'agente e tenta di leggere l'etichetta del piano, definita dalla `atomic_formula` rappresentata con un `literal` , come se fosse l'ennesimo belief della base dei belief, rappresentato anch'esso come `literal`.

La soluzione al problema richiede una modifica della grammatica che nella versione 0.4 è

```
agent          → beliefs plans
beliefs        → ( literal "." )*
                 N.B.: a semantic error is generated if the
                 literal was not ground.
plans          → ( plan )+
plan           → [ atomic_formula "->" ]
                 triggering_event ":" context "<-" body "."
```

Abbiamo suggerito agli sviluppatori di Jason diverse possibili modifiche:

- si può introdurre un terminale che faccia da terminatore per la base dei belief dell'agente
  `beliefs -> (literal ‘‘.’’)* ‘‘#’’`
- si può introdurre un terminale che funga da separatore tra la base dei belief e la libreria dei piani dell'agente
  `agent -> beliefs ‘‘#’’ plans`
- si può introdurre un terminale per indicare il punto in cui inizia la definizione della libreria dei piani
  `plans -> ‘‘#’’(plan)+`
- si può introdurre un terminale che funga da prefisso per indicare che il `literal` che segue è un etichetta di piano e non un belief
  `plan -> [ ‘‘#’’atomic_formula ‘‘->’’ ]`
  `triggering_event ‘‘:’’ context ‘‘<-’’ body ‘‘.’’`

È quest'ultima modifica quella adottata dagli sviluppatori di Jason e introdotta nella versione 0.5 del novembre 2004, che ha portato a ridefinire in questo modo alcune regole della grammatica

```
agent          → beliefs plans
beliefs        → ( literal "." )*
                 N.B.: a semantic error is generated if the
                 literal was not ground.
plans          → ( plan )+
plan           → [ "@" atomic_formula ]
                 triggering_event ":" context "<-" body "."
```

arrivando alla sintassi definitiva vista nel capitolo precedente in Figura 5.1.

L'ultima versione ha inoltre introdotto:

- l'uso di operatori relazionali infissi entro il contesto di un piano $(<, <=, >=, ==, \backslash ==, = (unifica))$;
- indicazioni sul modo in cui le classi Java vanno inserite entro gli esempi da eseguire e sul modo in cui vanno riferite entro i file di configurazione `.mas2j` del MAS;
- una nuova disposizione dei package contenti il codice dell'interprete Jason per cui la classe costruita dall'utente per l'ambiente dovrà importare alcuni package

  ```
  import jason.asSyntax.*;
  import jason.environment.*;
  ```

- una nuova azione interna `.broadcast(<ilforce>,<content>)` entro la libreria standard.

Un altro problema che abbiamo riscontrato durante i test su Jason, rilevato anche da altri utilizzatori, in maniera indipendente e contemporanea a noi, riguarda il funzinamento dell'azione interna `.send` nel caso di invio di messaggio avente come forza illocutoria `tellHow` e quindi come contenuto un piano. La soluzione del problema è fondamentale proprio perché il meccanismo di scambio di piani usa messaggi di questo tipo per trasmettere i piani rilevanti all'agente che ne ha fatto richiesta.

L'azione interna `.send`, indipendentemente dal tipo di forza illocutoria indicata dal messaggio, tratta il contenuto come se fosse un belief o un goal tentando di ricondurlo ad un letterale (istanza della classe `Literal`) che viene istanziato usando l'unificatore corrente e poi codificato con una stringa (ovvero con un dato di tipo `String` di Java) per essere inserito nel messaggio da inviare. Nel caso indicato sopra però la stringa che rappresenta il contenuto del messaggio codifica un piano che non può essere ricondotto al letterale atteso e quindi viene generato un errore in fase di esecuzione.

Nell'attuale versione dell'interprete il problema è stato risolto fornendo una soluzione che è valida solo se si considera l'invio del messaggio con forza illocutoria `tellHow` ma non la sua ricezione. Quando il messaggio viene inviato l'ostacolo può essere aggirato inserendo il piano tra doppi apici in modo che dal punto di vista sintattico equivalga ad un termine, ovvero ad un "oggetto" riconducibile ad un letterale. Quando il messaggio raggiunge la mailbox e viene scelto per essere gestito però si ottiene un nuovo errore in quanto si tenta di decodificare come piano la stringa rappresentate il contenuto che ora codifica un letterale.
Per questo problema non abbiamo proposto alcuna soluzione agli autori di Jason. Siamo a conoscenza del fatto che a breve sarà pubblicata la versione 0.6 nella quale sono previste importanti innovazioni relative ai meccanismi di gestione dei messaggi.
Per poter procedere nell'implementazione, tuttavia, abbiamo studiato due possibili soluzioni di "ripiego": o ridefinire il comportamento tenuto al momento della ricezione del messaggio, o ridefinire l'azione interna `.send` in modo che tratti il contenuto in modo diverso in base alla forza illocutoria del messaggio. Come si vedrà in seguito, combinando queste due proposte siamo riusciti a proggettare un meccanismo per la trasmissione di un insieme di piani che però risulta complesso e inefficiente. Il numero di messaggi inviati è maggiore di quello dei piani da trasmettere. Con una gestione dei messaggi diversa avremmo potuto inviare anche diversi piani con un unico messaggio.

Abbiamo rilevato anche un errore di compilazione nel caso in cui più agenti entro un MAS usino la stessa classe per ridefinire il proprio comportamento. La possibilità di avere più agenti che ridefiniscono il proprio comportamento riferendo la stessa classe è però fondamentale per lo sviluppo del prototipo di Coo-AgentSpeak. Infatti il modo che si usa per indurre gli agenti a cooperare è quello di riferirli ad una stessa classe che ne estenda il comportamento.

Dato questo esempio che modella la situazione in cui gli agenti `ag0` e `ag1` si riferiscono alla stessa classe `myPkg.MyAgClass` e alla stessa architettura `myPkg.MyAgArch`:

```
MAS testCentDuplex {
    architecture: Centralised
    environment:
        myPkg.testEnv at localhost
        // testEnv is the class that implements the environment,
        // it is in the myPkg java package

    agents:
        // runs two instances of the agent ag0
        ag0 ./as/ag0.asl [events=retrieve,verbose=3]
          agentArchClass myPkg.MyAgArch
          agentClass myPkg.MyAgClass
```

```
        #2 at localhost;

        ag1 ./as/ag1  //.asl default extension for AgentSpeak files
          agentArchClass myPkg.MyAgArch
          agentClass myPkg.MyAgClass
          at localhost;
}
```

il messaggio di errore notificato da Jason è il seguente:

```
Compiling user class with
C:\j2sdk1.4.2_05\bin\javac -classpath
.;C:\Jason/ulibs/;C:\Jason/bin/classes/;
  C:\saci/bin/saci.jar;C:\Jason\SimpleCorrectErrDuplicato\;
  %CLASSPATH% myPkg/MyAgArch.java myPkg/MyAgArch.java
    myPkg/MyAgClass.java myPkg/MyAgClass.java myPkg/testEnv.java

myPkg/MyAgArch.java:6: duplicate class: myPkg.MyAgArch
public class MyAgArch extends CentralisedAgArch {
       ^
myPkg/MyAgClass.java:7: duplicate class: myPkg.MyAgClass
public class MyAgClass extends Agent {
       ^
2 errors
```

La spiegazione della causa di questo errore, che è stato notificato agli autori di Jason, e la relativa correzione al momento non sono disponibili. Per procedere nella progettazione dell'interprete abbiamo aggirato il problema compilando la classe una prima volta manualmente, in quanto il meccanismo di compilazione delle classi riferite nel file di configurazione si attiva solo se non esiste una versione aggiornata del file `.class` corrispondente.

## 6.2 Progettazione del prototipo

Nel capitolo precedente abbiamo descritto la proposta implementativa considerando quattro aspetti: definizione della strategia di cooperazione, estensione dei piani, sospensione delle intenzioni e funzionamento dell'interprete. Descriviamo, invece, la progettazione del prototipo seguendo le tre fasi che costituiscono il meccanismo di recupero dei piani esterni. Cominciamo col descrivere l'avvio del meccanismo di recupero, proseguiamo con la risposta ad una richiesta di piani pervenuta e concludiamo con la raccolta dei piani ottenuti in risposta e con il ripristino delle intenzioni sospese in attesa su di una richiesta.

Si vedono in dettaglio alcune delle scelte implementative indicate nella sezione conclusiva del capitolo precedente.

### 6.2.1 Avvio della richiesta di recupero dei piani

Il recupero di piani esterni ha origine nell'ambito della gestione della coda degli eventi (event queue). Parlando della event queue si sono individuati due tipi di eventi:

- *eventi speciali*: aventi triggering event `+!cooAS_sendPlanFor(`*`lista di argomenti`*`)`. Questo triggering event corrisponde all'aggiunta nella event queue di un achieve goal per la richiesta di piani rilevanti da parte di un altro agente.
- *eventi normali*: tutti gli eventi non speciali.

#### Gestione evento

Quando quello selezionato è un evento normale con triggering event `te` si procede con la ricerca dei piani rilevanti e applicabili per `te` e poi con la selezione dell'istanza di piano applicabile da mandare in esecuzione. L'istanza di piano è costituita da un piano e dall'unificatore usato per istanziarlo. La selezione di questo piano istanziato è affidata ad un metodo (il metodo `selectOption` della classe `Agent`) che nella nostra implementazione si comporta diversamente a seconda di quali sono le strategie di cooperazione dell'agente definite mediante i predicati `cooAS_planSources`, `cooAS_acquisitionPolicy` e `cooAS_retrievalPolicy` discussi nel capitolo precedente.

Entro il metodo di selezione delle istanze di piano da mandare in esecuzione si possono verificare diverse situazioni:

- se la strategia di recupero è `noLocal` e ci sono piani rilevanti e applicabili per `te` si restituisce uno di questi piani istanziato; se non ci sono piani rilevanti o non ci sono piani applicabili per `te` si guarda se la lista degli agenti fidati per `te` è vuota: in caso affermativo si restituisce null, altrimenti parte la richiesta di recupero di piani esterni.
- se la strategia di recupero è `always`, indipendentemente dalla presenza di piani rilevani e applicabili in locale si guarda se la lista degli agenti fidati per `te` è vuota: in caso affermativo, se c'è, si restituisce un'istanza di piano applicabile, altrimenti parte la richiesta di recupero di piani esterni.

Abbiamo implementato la richiesta di recupero di piani esterni generando un piano di sistema istanziato e gestendo la sospensione dell'intenzione legata a `te`. Il piano di sistema sarà restituito dal metodo di selezione dei piani istanziati stesso per essere mandato subito in esecuzione.

Il piano di sistema è:

```
+!startExternalPlanRetrieval(te, reqId) :
    cooAS_planSources(te, AgL) <-
```

```
cooASlib.sendAll(achieve, AgL,
cooAS_sendPlanFor(te, ReqId, myId)).
```

`reqId` è una stringa con cui una richiesta di recupero piani viene identificata univocamente. `AgL` è la lista contenente gli agenti fidati cui inviare le richieste di piano. `cooASlib.sendAll` è un'azione interna della libreria `cooASlib` contenente quelle azioni interne che servono per far funzionare Coo-AgentSpeak, che sarà presentata nella sezione successiva. `myId` è l'identificatore dell'agente che avvia la richiesta.

Abbiamo implementato l'*identificatore univoco della richiesta*, `reqId` nella descrizione del piano data sopra, come una stringa della forma `cooAS_ReqId_<agent name><numero progeressivo>` dove `<agent name>` è il nome dell'agente che genera la richiesta e `<numero progressivo>` è un intero che cambia ad ogni richiesta.

### Sospensione dell'intenzione

In Jason le intenzioni si trovano in diversi contesti:

- nell'insieme delle intenzioni dentro una circostanza sono incluse le intenzioni selezionabili per essere processate e quindi attive;
- sono associate ad un triggering event per un evento inserito nella event queue;
- sono legate ad un'azione in esecuzione inserita nell'insieme delle azioni pendenti.

La proposta di usare un belief per memorizzare le intenzioni sospese descritta in [4] garantiva la possibilità di modificare dinamicamente, mediante azioni interne definite appositamente, le informazioni relative alla sospensione. Si pensava ad esempio di usare un'azione interna per cancellare dalla lista degli agenti da cui si attendevano le risposte, gli identificatori di quegli agenti che avevano già risposto.

Secondo una nostra valutazione questa soluzione era però resa inefficiente dal vincolo imposto sui belief entro la base in quanto essi devono essere ground. Inoltre un belief è implementato con un predicato che può avere come argomenti solo termini. Per questi motivi il belief per la sospensione doveva contenere solo costanti il ché richiedeva la codifica in stringhe dei suoi argomenti, in modo che fossero tutti termini. Il passaggio di decodifica da stringa costante ad oggetto però per alcuni di essi, come ad esempio l'intenzione che compone l'evento, non era banale.

Di conseguenza per quanto riguarda la sospensione dell'intenzione abbiamo considerato strade alternative.

*Alternativa 1.* Le intenzioni sospese sono inserite in un'apposita struttura

aggiunta allo stato dell'agente Coo-AgentSpeak realizzato con un HashMap di Java. La chiave per l'oggetto é la stringa `reqId`. Le altre informazioni legate alla sospensione dell'intenzione quali la lista degli agenti fidati da cui si attendono risposte alla richiesta, il triggering event per cui si è generata la richiesta e l'identificatore della richiesta sono inserite in cima allo stack.

Questo modo di procedere può generare dei problemi nel corso dell'esecuzione dell'azione interna `.desire` di Jason. Tale azione esamina tutte le intenzioni presenti nel sistema alla ricerca di un determinato achieve goal (desiderio). Oltre alle intenzioni attive, sono considerate quelle legate ad azioni pendenti e quelle inserite nella coda degli eventi in associazione ai triggering event. A questo punto le intenzioni inserite nella struttura di cui sopra sarebbero escluse da questa ricerca pregiudicando il corretto funzionamento dell'azione interna che potrebbe fallire erroneamente in quanto in realtà il desiderio c'è.

*Alternativa 2.* Si propone di tenere l'intenzione all'interno dell'insieme delle intenzioni mettendo in cima ad essa, implementata con uno Stack di Java, le informazioni relative alla richiesta come la lista degli agenti fidati da cui si attendono risposte, il triggering event che l'ha generata la richiesta e l'identificatore della stessa. Le intenzioni sospese si distinguono da quelle attive proprio perché l'elemento in cima allo stack non è un'istanza di piano ma la stringa che identifica univocamente la richiesta.

Questo modo di procedere può creare problemi in fase di esecuzione di alcune azioni interne, quali `.desire` e `.dropDesire`, che esaminano il contenuto delle intenzioni (di tutte, anche di quelle non contenute nell'insieme delle azioni) dando per scontato che queste contengano solo istanze di piano.

Per questo motivo abbiamo proposto una variante di quest'ultima alternativa.

*Alternativa 3.* Anche qui l'intenzione sospesa resta nell'insieme delle intenzioni insieme a quelle attive. Al fine di distinguerle si inserisce in cima all'intenzione da sospendere un'istanza di piano costruita ad hoc in modo che contenga anche le informazioni relative alla sospensione. Il piano istanziato è composto da una coppia (piano, unificatore) in cui l'unificatore è vuoto e il piano è

```
@reqId [AgL] te: true <- true;
```

Le intenzioni sospese si riconoscono in quanto l'etichetta del piano al top corrisponde all'identificatore della richiesta. La lista degli agenti che devono rispondere viene messa come annotazione e come triggering event si ha il trigger che ha originato la richiesta. In questo modo queste informazioni sono accessibili senza richiedere particolari meccanismi di codifica e decodifica.

*Alternativa 4.* Si propone una soluzione ideale che è un ibrido tra l'uso del predicato di sistema ipotizzato nella proposta iniziale e la variante della seconda alternativa, vista sopra. Il predicato di sistema contiene solo alcune delle informazioni, ovvero la lista degli agenti da cui si attendono risposte e l'identificatore della richiesta. Tra queste la lista degli agenti da cui si attendono i piani è modificabile dinamicamente. Le intenzioni sospese sono messe nell'insieme delle intenzioni e vengono distinte da quelle attive in quanto in cima hanno un'istanza di piano particolare. La particolarità del piano istanziato è data dalla sua etichetta dalla quale si può recuperare l'identificatore della richiesta collegata. Questo identificatore é usato come chiave primaria della relazione che lega l'intenzione sospesa alle informazioni ad essa relative.

Se avessimo adottato questa soluzione ideale il piano di sistema avrebbe avuto questa forma:

```
+!startExternalPlanRetrieval(te, reqId) :
cooAS_planSources(te, reqId, AgL) <-
    +cooAS_suspendedInt(reqId, AgL);
      cooASlib.sendAll(achieve, AgL,
      cooAS_sendPlanFor(te, reqId, myId)).
```

Inoltre, avremmo dovuto prevedere l'esistenza di un piano di sistema per gestire l'evento con triggering event `+cooAS_suspendedInt(reqId, AgL)` in modo opportuno in modo da evitare un'ulteriore ricerca di piani che non avrebbe avuto successo. Questo piano

```
@cooAS_suspInt +cooAS_suspendedInt(X, Y) : true <- true.
```

lo avremmo aggiunto alla libreria dei piani dell'agente al momento dell'istanziazione dell'agente.

La soluzione che abbiamo scelto per l'implementazione è quella che lascia le intenzioni sospese nell'insieme delle intenzioni (Alternativa 3) distinguendole dalle altre con l'istanza di piano etichettata con l'identificatore della richiesta. Si sacrifica la possibilità di aggiornare dinamicamente le informazioni concentrandosi sullo sviluppo di un unico meccanismo per la gestione della sospensione.

Si evidenzia come nell'idea iniziale presentata in [4] il predicato per la sospensione dell'intenzione avesse come quarto argomento la lista dei piani ottenuti in risposta alla richiesta. La soluzione adottata non ha più bisogno di questa struttura in quanto, come si vedrà in seguito, i piani recuperati saranno aggiunti direttamente alla libreria dei piani dell'agente.

#### Piano di sistema per l'avvio della richiesta

In base alle scelte fatte ai passi precedenti procediamo come spiegato nel seguito.

Il piano di sistema adottato (mostrato nel paragrafo "Gestione evento") ha come contesto `cooAS_planSources(te, AgL)`. Se la condizione del contesto è verifiata, nella variabile `AgL` si recupera la lista degli agenti a cui inviare le richieste. Il corpo del piano contiene l'azione interna riferita come `cooASlib.sendAll`, inclusa nella libreria per Coo-AgentSpeak, che si occupa di inviare un messaggio di tipo

```
<achieve, myId, agId, cooAS_sendPlanFor(te, reqId, myId)>
```

ad ogni agente `agId` incluso nella lista in `AgL`, in cui si richiedono piani rilevanti per `te` a fronte della richiesta identificata da `reqId` emessa dall'agente `myId`.

L'azione interna `cooASlib.sendAll` prepara il contenuto del messaggio di tipo `achieve` riconducendolo ad un letterale che viene istanziato esattamente come fa l'azione `.send` della libreria standard e poi genera e inivia per ogni agente incluso in `AgL` un messaggio con quello stesso contenuto.

In [4] l'avvio del meccanismo di recupero avveniva generando e inserendo nella event queue un evento speciale `+!startExternalPlanRetrieval(te, reqId)`. La `selectEvent` doveva scegliere poi tale evento con priorità massima rispetto ad altri seguendo il normale ciclo di ragionamento dell'agente per gestirlo e restituire il piano di sistema mediante la `selectOption`. Abbiamo ritenuto inutile questo modo di procedere e deciso di far restituire il piano di sistema dalla `selectOption` non appena si rileva la necessità di avviare il recupero di piani.

### 6.2.2 Risposta ad una richiesta di piani

La risposta alla richiesta di piani è costituita da un meccanismo che viene attivato dalla ricezione di un messaggio del tipo

```
<achieve, ag, cooAS_sendPlanFor(te, reqId, ag)>
```

che indica che l'agente `ag`, mittente del messaggio, vuole che l'agente ricevente soddisfi il goal `cooAS_sendPlanFor(te, reqId, ag)` impegnandosi a cercare piani rilevanti per `te` che gli devono essere inviati in risposta alla richiesta identificata da `reqId`.

Il messaggio ha forza illocutoria `achieve` e quindi nella event queue viene inserito un evento con intenzione nuova, e quindi vuota, e triggering event

```
+!cooAS_sendPlanFor(te, reqId, ag)
```

Abbiamo scelto di includere nel predicato `cooAS_sendPlanFor` le informazioni relative al mittente della richiesta e all'identificatore della richiesta. Questa scelta consente di avere memorizzate delle informazioni utili nel corso della gestione della richiesta che non potranno più essere accessibili in seguito oppure che non saranno più memorizzabili a causa dei vincoli tecnici dettati dall'implementazione dell'interprete.

### Ricezione messaggio di richiesta

Nella proposta originale data in [4] i messaggi di questo tipo avevano priorità massima su ogni altro messaggio e pertanto la funzione di selezione dei messaggi (`selectMessage`) andava ridefinita in modo da dare la precedenza a tali messaggi.

Nell'implementazione abbiamo scelto di non assegnare alcuna priorità ai messaggi. Le ragioni sono due: una di natura tecnica l'altra di natura concettuale.

Dal punto di vista tecnico la scelta prioritaria di un messaggio entro la mailbox implica l'applicazione di un algoritmo di ricerca dell'elemento da selezionare. La mailbox è implementata con una semplice lista linkata e quindi non supporta alcun meccanismo per la gestione della priorità in modo efficiente. Il vantaggio che si ottiene dallo scegliere prima un certo messaggio è vanificato dal tempo "perso" per trovarlo.

Concettualmente si pensa che l'uso di Coo-AgentSpeak come strumento di programmazione di MAS ad alto livello implichi un massiccio uso del meccanismo di recupero dei piani da parte degli agenti, per cui i messaggi di richiesta si presenteranno spesso entro la mailbox di un agente. Adottare una strategia che dà precedenza a messaggi di un certo tipo rischia di rimandare a tempo indeterminato la gestione di altri messaggi generati nell'ambito dell'esecuzione di operazioni, diverse da quelle di scambio di piani, che costituiscono parte dell'attività dell'agente e che devono a loro volta proseguire.

### Gestione evento di richiesta piani

Quando la funzione di selezione degli eventi restituisce un evento il cui triggering event è quello corrispondente a una richiesta di piani rilevanti si ottiene un piano di sistema studiato appositamente per gestire questa situazione. Il piano ha come triggering event `+!cooAS_sendPlanFor(te, reqId, ag)`, risulta sempre applicabile e il suo corpo comprende azioni interne. Queste azioni interne si occupano del recupero dei piani rilevanti per `te` e della trasmissione degli stessi all'agente richiedente `ag` indicando che questa trasmissione avviene in risposta alla richiesta identificata da `reqId`.

Il piano di sistema è della forma

```
+!cooAS_sendPlanFor(te, reqId, ag) : true
<- cooASlib.searchAndSendPlans(ag, reqId, te).
```

L'azione interna riferita da `cooASlib.searchAndSendPlans` è inclusa nella libreria delle azioni per Coo-AgentSpeak e si occupa di tutto il processo di ricerca e trasmissione dei piani rilevanti.

**Ricerca piani.** Si recuperano dalla libreria i piani rilevanti per `te` e vengono raccolti in una lista senza essere istanziati. Vengono eliminati dalla lista quei piani la cui sorgente è diversa dall'agente in quanto nella libreria dei piani potrebbero essere presenti piani che non appartengono all'agente ma che gli sono arrivati in risposta a richieste di piani pendenti. Si tolgono dalla lista quei piani aventi specificatore d'accesso *private* oppure *only(ListaAgentiFidati)* quando l'identificatore `ag` dell'agente che ha fatto la richiesta non è incluso in *ListaAgentiFidati*. I piani rimasti nella lista saranno trasmessi.

**Trasmissione piani.** Si procede con la trasmissione dei piani inclusi nella lista.

- L'inizio della trasmissione della sequenza di piani rilevanti in risposta ad una richiesta è notificato all'agente richiedente `ag` inviandogli un messaggio con forza illocutoria `tell` contenente il belief `cooAS_initTrasm(reqId)`.
- Per ogni piano incluso nella lista dei piani rilevanti si modifica la sorgente del piano inserendo `reqId` al posto dell'identificatore dell'agente nell'annotazione del piano che la indica. Il piano così modificato viene quindi inviato ad `ag`, senza essere istanziato, entro un messaggio avente forza illocutoria `tellHow`.
- Si segnala la fine della trasmissione inviando all'agente richiedente `ag` un messaggio con forza illocutoria `tell` contenente il belief `cooAS_endTrasm(reqId)`.

A fronte dell'aggiunta di questi belief alla base sono generati degli eventi che andrebbero gestiti per evitare che anche per questi venga avviato il meccanismo di recupero di piani senza successo. Usiamo due piani di sistema aventi il solo scopo di gestire questa situazione che vengono aggiunti alla libreria dei piani al momento dell'istanziazione dell'agente.

Una variante non implementata prevede che i passi di ricerca piani e trasmissione piani siano gestiti da due azioni interne differenti.

Nella proposta originale l'evento di tipo `+!cooAS_sendPlanFor(te, reqId,`

ag) era selezionato con priorità rispetto ad altri eventi modificando il comportamento della funzione di selezione `selectEvent`.

Abbiamo scelto anche questa volta di non assegnare priorità agli eventi e le motivazioni sono le stesse indicate per la funzione di selezione dei messaggi.

### 6.2.3 Recupero piani e ripristino intenzione sospesa

Una volta ricevuti i piani in risposta alla richiesta si passa al ripristino dell'intenzione sospesa e alla gestione dei piani recuperati in base alla politica di acquisizione stabilita per l'evento che ha generato la richiesta.

#### Recupero risposte

Secondo le scelte implementative che abbiamo adottato sin qui, tra i messaggi che arrivano alla mailbox si distinguiamo quelli relativi alle risposte in arrivo dagli agenti a cui si era inviata la richiesta. Questi sono gestiti in modo particolare distinguendo tre tipi di messaggi contemplati nella sequenza di risposta.

- Messaggio con contenuto `cooAS_initTrasm(reqId)` e forza illocutoria `tell`: si recupera dal contenuto `reqId` e si guarda se c'è un'intenzione sospesa avente in cima un piano istanziato etichettato con esso. In caso negativo il messaggio viene scartato, altrimenti si procede in modo che il contenuto venga inserito nella base dei belief. A fronte di tale inserimento viene generato un evento e inserito nella event queue.

- Messaggio con forza illocutoria `tellHow` contenente un piano: si recupera l'identificatore della sorgente del piano dalle annotazioni. Se questo corrisponde ad un identificatore `reqId` di richiesta di piani si fanno i seguenti controlli: se nella libreria dei piani c'è un predicato `cooAS_initTrasm(reqId)` e non c'è il predicato contenuto `cooAS_endTrasm(reqId)` il messaggio è valido e il piano viene aggiunto alla libreria dei piani, altrimenti il piano viene scartato.

- Messaggio con contenuto `cooAS_endTrasm(reqId)` e forza illocutoria `tell`: si recupera dal contenuto `reqId` e si verifica che nella base dei belief ci sia il predicato `cooAS_initTrasm` avente come argomento `reqId`. In caso negativo il messaggio viene scartato. In caso affermativo il messaggio è valido e si recupera l'identificatore dell'agente che ha inviato il messaggio. Questo identificatore viene usato per aggiornare la lista degli agenti da cui si attendono risposte. L'informazione relativa a questa lista è memorizzata entro l'annotazione del piano in cima all'intenzione sospesa per la richiesta. Infine, il predicato `cooAS_endTrasm(reqId)` è inserito nella base dei belief. Anche per questo inserimento viene generato un evento da porre nella event queue.

Se nella fase di sospensione dell'intenzione avessino adottato la soluzione ibrida (Alternativa 4), avremmo potuto gestire l'evento generato dalla ricezione del messaggio di fine della trasmissione con un piano di sistema. Questo piano avrebbe previsto un'azione interna da usare per togliere il mittente del messaggio dalla lista degli agenti da cui attendere risposta. Per poter fare questo il belief che indica la fine della trasmissione avrebbe dovuto avere un secondo argomento contentente l'indicazione dell'identità dell'agente che lo ha inviato. Tale belief quindi sarebbe stato `cooAS_endTrasm(reqId, ag)` con `ag` identificatore dell'agente mittente.

Originariamente era previsto che i piani rilevanti trovati venissero raccolti in una lista da inviare entro un unico messaggio di tipo `tellHow` come risposta. Questo modo di procedere avrebbe però generato un errore in fase di lettura del messaggio in quanto a fronte della ricezione di un messaggio di questo tipo ci si aspetta di dover aggiungere alla libreria dei piani un unico piano. Per evitare l'errore avremmo dovuto ridefinire il comportamento del metodo che aggiunge alla libreria un unico piano in modo che potesse aggiungere liste di piani. Tale metodo però non è tra quelli predisposti per la modifica da parte dell'utente.

L'uso della sequenza di trasmissione dei piani che abbiamo adottato nell'implementazione ha proprio lo scopo di evitare problemi nella fase di ricezione di un piano senza dover ridefinire metodi non modificabili.

#### Ripristino intenzione sospesa

Inizialmente era prevista contestualmente alla ricezione dell'ultima risposta attesa per la richiesta.

Nell'implementazione abbiamo collocato l'individuazione di un'intenzione sospesa da ripristinare entro la fase di selezione dell'intenzione da processare. Si controllano tutte le intenzioni sospese entro l'insieme delle intenzioni. Queste intenzioni hanno in cima un piano istanziato la cui etichetta è un identificatore di richiesta `reqId` come detto altre volte in precedenza.

Se l'annotazione del piano è vuota si ha che tutti gli agenti hanno risposto alla richiesta e quindi l'intenzione può essere riattivata togliendo questo piano dalla cima della stessa. Oltre a `reqId` dal piano istanziato si recupera il triggering event che identifica l'evento che aveva generato la richiesta.

Questo triggering event è usato per recuperare dalla libreria dei piani il piano da mandare in esecuzione che verrà messo in cima all'intenzione, la quale sarà reinserita nell'insieme delle intenzioni. Dall'insieme aggiornato con l'intenzione ripristinata viene scelta un'intenzione da eseguire.

Si poteva anche restituire per l'esecuzione proprio l'intenzione ripristinata assegnando così priorità maggiore alle intenzioni appena riattivate rispetto a quelle attive, cosa che avrebbe potuto portare ad una situazione di stallo nella gestione delle intenzioni attive.

### Scelta del piano istanziato da eseguire

Tenendo conto delle scelte implementative fatte in precedenza procediamo al rispristino di un'intenzione sospesa e contestualmente alla selezione di un piano applicabile da istanziare e inserire in cima all'intenzione.

In modo analogo a quanto si fa nella gestione di un evento normale, si cercano nella libreria i piani rilevanti e quindi quelli applicabili.

Dalla lista di piani applicabili si tolgono quelli la cui sorgente non è né l'identificatore dell'agente, né l'identificatore della richiesta per cui l'intenzione era stata sospesa. Tra i piani rimanenti si sceglie l'istanza di piano per l'esecuzione usando la stessa funzione di selezione usata nella normale fase di gestione degli eventi. Questa istanza di piano viene quindi messa in cima all'intenzione riattivata.

I dettagli relativi al completamento di questa fase nella proposta di implementazione iniziale non erano specificati così come non lo erano quelli relativi al trattamento dei piani recuperati secondo la politica di acquisizione data per il triggering event. Per la gestione della politica di acquisizione abbiamo deciso di seguire con qualche variazione le indicazioni date nella descrizione del comportamento dell'interprete Coo-BDI che applicava il criterio di acquisizione solo al piano scelto dopo che questo era stato eseguito.

### Politica di acquisizione

La politica di acquisizione, recuperata dal predicato di sistema `cooAS_acquisitionPolicy`, stabilisce cosa fare con tutti i piani rilevanti recuperati. Abbiamo scelto di occuparsi dell'acquisizione dei piani recuperati solo dopo che il piano istanziato da eseguire è stato selezionato.

Il modo di procedere è determinato dal valore assegnato alla politica di acquisizione. Ci sono tre valori possibili.

- *add*: tutti i piani contenuti nella libreria dei piani annotati con `reqId` nella sorgente che sono stati inseriti tra i piani rilevanti nella fase di ricerca dell'istanza di piano, restano nella libreria dei piani ma la loro sorgente viene modificata con l'identificatore dell'agente. L'agente ora è il possessore di tali piani.

- *discard*: tutti i piani contenuti nella libreria dei piani annotati con `reqId` nella sorgente che sono stati inseriti tra i piani rilevanti nella fase di ricerca dell'istanza di piano sono eliminati dalla libreria.

- *replace*: si considerano i piani inseriti nella lista di quelli rilevanti nella fase di ricerca del piano istanziato. Tra questi si gestiscono quelli di proprietà dell'agente che vengono eliminati dalla libreria dei piani. Finita questa fase di eliminazione si gestiscono gli altri piani rilevanti che restano nella libreria dei piani con la sorgente che cambia da `reqId`

all'identificatore dell'agente stesso. In pratica i piani di proprietà dell'agente sono sostituiti da quelli recuperati con la richiesta.

## 6.3 Implementazione di Coo-AgentSpeak

L'interprete Jason è realizzato in Java. Tra le sue caratteristiche, presentate nel capitolo precedente, è risultata utile la possibilità di estendere in parte il comportamento dell'interprete lasciando all'utente la possibilità di ridefinire alcuni metodi e di creare nuove azioni interne.

I metodi predisposti per essere modificati appartengono a classi che definiscono il comportamento di agenti, architetture e ambienti.

L'implementazione che abbiamo realizzato si concentra sulla variazione del comportamento dell'agente. A tal proposito abbiamo definito una classe `CooASAgent` che estende la classe `Agent` in cui è definito il comportamento degli agenti AgentSpeak(L). Abbiamo inoltre definito la libreria `cooASlib` avente al suo interno alcune azioni interne usate dal meccanismo di scambio di piani.

### 6.3.1 Classe CooASAgent

La classe `CooASAgent` è inserita nel package `jason.asSemantics`. Inizialmente avevamo pensato di inserirla in un package distinto, creato apposta per l'estensione. Il fatto che molti oggetti in Jason abbiano attributi e metodi non accessibili da package diversi ci ha costretto ad una scelta diversa.

Abbiamo esteso lo stato dell'agente con l'inserimento di un campo statico usato per implementare l'identificatore unico della richiesta di recupero di piani esterni.

All'interno della classe si sono ridefiniti i metodi di selezione dell'istanza di piano da mandare in esecuzione `selectOption` e di selezione dell'intenzione `selectIntention`. Oltre ai metodi di selezione, un'agente ha metodi usati per decidere se un messaggio in arrivo da un dato agente è accettabile e può quindi essere gestito. Tra questi è ridefinito il metodo di gestione dei messaggi di tipo `tell`/`tellHow` in ingresso `acceptTell`.

Prima di procedere con la descrizione delle modifiche ricordiamo che parte dello stato di un agente in esecuzione è costituito dalle circostanze. In ogni circostanza sono memorizzate varie informazioni tra cui l'insieme delle intenzioni (attive e sospese), l'evento e l'intenzione correntemente selezionati.

#### Gestione dell'identificatore di richiesta

Entro la classe `CooASAgent` è aggiunto un attributo di tipo `int` con cui si realizza un contatore usato per implementare l'identificatore unico di richiesta di piano.

Ogni volta che viene attivata una richiesta il contatore viene incrementato. Il valore ottento viene concatenato ad una stringa costituita dal prefisso `cooAS_ReqId_` e dal nome dell'agente che ha avviato la richiesta.

Abbiamo definito anche un metodo che presa una stringa verifica che questa sia un identificatore di richiesta.

#### selectOption

Il metodo `selectOption` è quello che avvia il meccanismo di recupero dei piani. Questo metodo è stato ridefinito in modo da funzionare diversamente a seconda che ci si trovi in una situazione precedente o successiva all'avvio della richiesta di recupero.

La prima azione fatta all'interno di `selectOption` ha il compito di determinare in quale delle due modalità si sta operando.

Abbiamo implementato un metodo ausiliario che ricerca entro l'insieme delle intenzioni un'intenzione sospesa ripristinabile per l'evento attualmente selezionato nella circostanza. La ricerca avviene scorrendo a partire dal primo elemento tutta la lista che rappresenta l'insieme delle intenzioni. Per ogni stack nella lista si preleva l'elemento al top. Se l'etichetta del piano in cima allo stack è un identificatore di richiesta l'intenzione è sospesa e si passa al controllo delle annotazioni. Se le annotazioni sono costituite dalla lista vuota tutti gli agenti hanno risposto alla richiesta e l'intenzione è ripristinabile. Si controlla quindi il triggering event e se coincide con quello correntemente selezionato l'intenzione è quella cercata.

Se l'intenzione è trovata, allora la fase di recupero di piani per quell'intenzione è stata ultimata e il metodo si comporta restituendo il primo elemento della lista di piani istanziati che ha come parametro. Se non si trova questa intenzione si è nella fase di normale selezione di un piano istanziato da mandare in esecuzione. È questa la fase da cui può nascere una richiesta di piani applicabili.

L'***avvio della richiesta di piani*** prevede:

- la generazione dell'identificatore unico della richiesta;
- la costruzione del piano di sistema per l'avvio della richiesta e la sua istanziazione;
- la costruzione del piano ad hoc per la sospensione dell'intenzione e la sua istanziazione;
- la sospensione dell'intenzione correntemente selezionata nella circostanza mediante inserimento del piano ad hoc istanziato in cima allo stack.

Per recuperare la lista degli agenti fidati e la politica di recupero per un dato triggering event abbiamo definito dei metodi appositi. All'interno

di questi si scandisce la base dei belief alla ricerca dei belief di sistema usati per definire le scelte cooperative per il triggering event corrente. Per poter inserire il triggering event nel belief questo va codificato come stringa. Nel momento del recupero delle informazioni di cui sopra si deve tenere conto di questo e la stringa per il triggering event va "ripulita". Una volta ripulita la stringa viene decodificata e il triggering event risultante può essere confrontato con quello correntemente selezionato nella circostanza. Se i due coincidono, l'informazione può essere recuperata.

### selectIntention

Con il metodo `selectIntention` si seleziona un'intenzione da mandare in esecuzione. A questo metodo abbiamo affidato anche il compito di ripristinare un'eventuale intenzione sospesa che attende di essere riattivata.

Si fa una ricerca entro l'insieme delle intenzioni per trovare un'intenzione riattivabile. Se non la si trova si restituisce la prima intenzione non sospesa dell'insieme.

Se si trova l'intenzione si recupera dal piano l'informazione relativa al triggering event che verrà usata per selezionare il piano applicabile da eseguire tra quelli applicabili.

La ricerca dell'intenzione riattivabile è fatta con un metodo che abbiamo appositamente definito. Il metodo scorre a partire dal primo elemento tutta la lista che rappresenta l'insieme delle intenzioni. Per ogni stack nella lista si preleva l'elemento al top. Se l'etichetta del piano in cima allo stack è un identificatore di richiesta l'intenzione è sospesa e si passa al controllo delle annotazioni. Se le annotazioni sono costituite dalla lista vuota tutti gli agenti hanno risposto alla richiesta e l'intenzione è ripristinabile.

Abbiamo implementato dei metodi ausiliari che vengono usati per scegliere i piani rilevanti e applicabili tra tutti quelli inclusi nella libreria dei piani compresi quelli recuperati esternamente. Questi metodi di fatto usano gli stessi algoritmi utilizzati in Jason per scegliere piani rilevanti e applicabili nel normale processo di applicazione delle regole.

Un altro metodo ausiliario è usato per togliere dall'insieme dei piani applicabili tutti quelli che non appartengono all'agente o che non sono riconducibili alla richiesta di piani per cui l'intenzione era stata sospesa. L'insieme "ripulito" si ottiene creando una lista in cui vengono copiate le istanze di piano "buone".

Si usa la `selectOption` nella modalità in cui restituisce il primo piano della lista di piani applicabili disponibili.

A questo punto si gestiscono i piani che sono entrati nella libreria dei piani in seguito alla richiesta seguendo le indicazioni fornite dalla politica di acquisizione.

Il piano istanziato scelto viene messo al posto di quello ad hoc in cima all'intenzione riattivandola.

Infine, si prende la prima intenzione non sospesa dall'insieme delle intenzioni e la si restituisce.

Anche il recupero della politica di acquisizione ha richiesto la definizione di un metodo ausiliario apposito. Lo abbiamo realizzato in modo analogo a quanto descritto per ottenere la politica di recupero e la lista degli agenti fidati.

#### acceptTell

Questo metodo è invocato quando si ricevono messaggi con forza illocutoria `tell` o `tellHow`. Nel primo caso il contenuto del messaggio è un belief rappresentato da un letterale; nel secondo è un piano.

L'`acceptTell` viene usata per fare i controlli sui messaggi in arrivo al fine di riconoscere e gestire quelli facenti parte di una trasmissione di piani giunta in risposta ad una richiesta.

Abbiamo ridefinito questo metodo in modo che tenga un comportamento differente a seconda della tipologia del contenuto del messaggio.

- Se il messaggio contiene un letterale, allora il messaggio è di tipo `tell`. A questo punto si controlla il contenuto del letterale alla ricerca dei predicati che delimitano la trasmissione.

  Quando si trova il predicato che segnala l'inizio della trasmissione per una data richiesta si cerca nell'insieme delle intenzioni un'intenzione sospesa per la stessa richiesta. Se non si trova tale intenzione si scarta il messaggio, altrimenti il messaggio è accettato.

  Quando il predicato trovato è quello di fine trasmissione si guarda se nella base dei belief c'è una segnalazione di inizio trasmissione per la stessa richiesta. In caso affermativo si aggiorna la lista degli agenti da cui si aspettano risposte togliendo da essa l'agente mittente del messaggio.

- Se il messaggio contiene un piano, allora il messaggio è di tipo `tellHow`. Si procede verificando che entro l'annotazione per la sorgente ci sia un identificatore di richiesta. Se questo non c'è il messaggio è accettato. Se invece si trova l'identificatore si controlla che il messaggio sia valido.

  Perché il messaggio sia valido nella base dei belief deve trovarsi il predicato per l'inizio della trasmissione ma non quello per la fine. In caso di validità del messaggio questo è accettato, in caso contrario viene scartato.

Abbiamo definito un metodo per la ricerca dei predicati di delimitazione della trasmissione nella base dei belief. Questo metodo semplicemente scorre tutta la lista che rappresenta i belief positivi entro la base alla ricerca dei predicati indicati facendo dei confronti sui loro funtori.

Anche la verifica dell'esistenza di un'intenzione sospesa sulla stessa richiesta per cui si è ricevuto il segnale di inizio trasmissione ha richiesto la realizzazione di un metodo apposito. Il metodo in questione funziona in maniera analoga a quello visto in precedenza per l'individuazione di un'intenzione riattivabile. Differisce da esso in quanto si limita a controllare che l'etichetta del piano in cima allo stack dell'intenzione sia un'identificatore unico di richiesta.

#### Istanziazione piani di sistema

Nel corso dell'esecuzione del meccanismo di recupero di piani si sono utilizzati più volte piani di sistema che a seconda della loro tipologia sono gestiti in modo differente.

- Il *piano di sistema per l'avvio della richiesta* viene generato dinamicamente entro la `selectOption`.
- Il *piano di sistema per la gestione di una richiesta* viene generato e inserito entro la libreria dei piani dell'agente al momento dell'istanziazione dello stesso. Questo meccanismo si ottiene ridefinendo metodo pubblico `parseAS` usato appunto per istanziare il piano.
- I *piani di sistema per gestire l'arrivo dei limitatori di trasmissione* sono anch'essi generati e inseriti nella libreria dei piani nel momento in cui il piano viene istanziato, mediante l'invocazione dl metodo `parseAS`.

### 6.3.2 Libreria cooASlib

All'interno della libreria sono definite due azioni interne. Entrambe sono usate all'interno dei piani di sistema usati per la gestione del meccanismo di scambio di piani tra gli agenti.

- La `cooASlib.sendAll` è eseguita nel corpo del piano di sistema che si occupa dell'avvio della richiesta di piani. Uno dei suoi argomenti è la lista degli agenti a cui inviare il messaggio, l'altro è il contenuto. Per ognuno degli agenti inclusi nella lista si costruisce il messaggio di tipo `achieve`. Il contenuto del messaggio viene reso ground prima di essere codificato nella stringa da spedire.
- La `cooASlib.searchAndSendPlans` è eseguita nel corpo del piano di sistema che si occupa di rispondere ad una richiesta di piani rilevanti. Ha tre argomenti: l'agente che ha mandato la richiesta a cui sono destinati i piani recuperati; l'identificatore della richiesta e il triggering event che ha generato la richiesta.

  Con un algoritmo di unificazione già presente in Jason, si recuperano i piani rilevanti per il triggering event specificato.

Con l'identificatore della richiesta si costruiscono i belief per delimitare la trasmissione. Si invia il messaggio di inizio trasmissione al destinatario della risposta di tipo `tell`.

Per ogni piano recuperato si controlla l'accessibilità: se il piano è pubblico oppure se il richiedente può accedervi il piano viene preparato per l'invio, altrimenti viene scartato. Il piano accessibile viene modificato in modo che nell'annotazione per la sorgente venga inserito l'identificatore del messaggio al posto di quello dell'agente proprietario. Tale piano viene codificato in una stringa e inviato con un messaggio di tipo `tellHow`.

Infine, si invia il messaggio di tipo `tell` con il belief che indica la fine di trasmissione.

Per il controllo sull'accessibilità dei piani abbiamo implementato un metodo che presi un piano ed un agente verifica se l'agente può accedere al piano.

All'interno di questo metodo abbiamo anche implementato il controllo sul formato che deve avere l'annotazione tipica dei piani in Coo-AgentSpeak.

Ogni piano infatti deve avere un'annotazione del tipo

```
cooAS([accSpec( accesso ), source( id )])
```

Se l'annotazione con funtore `cooAS` non rispetta questa struttura, viene lanciata l'eccezione `ErrCooASStructException`.

Il controllo sulla forma della struttura e quello sul formato dell'identificatore univoco di richiesta sono gli unici controlli fatti sul formato di costrutti di sistema. Si assume che l'utente usi i belief di sistema per la descrizione della strategia cooperativa in modo corretto e che l'utente non definisca predicati o azioni interne con lo stesso nome di quelli di sistema. Si intende usare il prefisso `cooAS` proprio per distringuere i costrutti di sistema specifici di Coo-AgentSpeak dagli altri.

### Esecuzione di un esempio Coo-AgentSpeak

Ogni agente è definito in modo che per ogni triggering event che vuole gestire sia specificata la strategia di cooperazione.

Per ogni piano nella libreria dei piani dell'agente deve essere fornita un'etichetta annotata con la struttura di funtore `cooAS` in modo da indicare l'identità del suo possessore e il suo livello di accessibilità.

Nel file di configurazione del MAS si deve specificare il valore `retrieve` in corrispondenza dell'opzione per la gestione degli eventi non trattati. Per ogni agente si deve indicare come classe di riferimento in `agentClass` la classe `jason.asSemantics.cooASAgent`.

Di seguito mostriamo un semplice esempio di file di configurazione per un MAS definito in Coo-AgentSpeak.

```
MAS mars {

environment:
        marsMEnv

agents:
  r1 r1.asl [events = retrieve]
    agentClass jason.asSemantics.CooASAgent;

  r2 [events = retrieve]
    agentClass jason.asSemantics.CooASAgent;

  r3 [events = retrieve]
    agentClass jason.asSemantics.CooASAgent;

}
```

### Installazione di Coo-AgentSpeak

Il file `CooASAgent.java` viene inserito nella directory `./jason/asSemantics`. La directory `cooASlib` per la libreria delle azioni interne, contenente i file `sendAll.java` e `searchAndSendPlans.java`, viene inserita allo stesso livello della directory `./jason`. Questi file vengono compilati e collocati nei package opportuni manualmente.

# Conclusioni e sviluppi futuri

I sistemi basati sugli agenti stanno prendendo sempre più piede nell'ambito delle diverse discipline considerate dalla Tecnologia dell'Informazione facendo sì che venga riconosciuta la crescente importanza del paradigma orientato agli agenti.

Il successo di tali sistemi si basa sul fatto che l'agente incarna concetti cognitivi che consentono di rappresentare entità complesse con nozioni ad alto livello in modo semplice.

Lo sviluppo di sistemi ad agenti è quindi al centro di studi e lavori di ricerca, molti dei quali si occupano della definizione dei sistemi multi-agente concentrandosi, talvolta sugli aspetti di definizione del singolo agente e dei suoi aspetti cognitivi, altre volte sull'aspetto sociale del sistema.

Questa tesi si colloca proprio in questo ambito di ricerca occupandosi dell'estensione di un linguaggio di specifica per agenti BDI, AgentSpeak(L), con la nozione di cooperazione introdotta con il modello Coo-BDI. Da questa estensione si ottiene un nuovo linguaggio Coo-AgentSpeak per il quale si è progettato un interprete. La cooperazione in Coo-BDI si realizza tramite un meccanismo di scambio di piani che consente all'agente di accrescere o modificare la propria conoscenza procedurale.

Il lavoro di tesi comincia con una rassegna sui linguaggi di specifica per agenti in cui si sono esaminati alcuni linguaggi con caratteristiche BDI e altri linguaggi facenti comunque uso di concetti cognitivi. Da questo esame si è visto come tra questi solo alcuni prevedano la possibilità di modificare dinamicamente la conoscenza procedurale (ovvero, i piani e o le regole) dell'agente.

Un discorso a parte si è fatto per il linguaggio AgentSpeak(L), uno dei primi linguaggi BDI proposti in letteratura, per il quale si sono descritte due estensioni: con la prima si introduce il concetto di azione interna, eseguita internamente all'agente senza influire sull'ambiente esterno; con la seconda si introducono nel linguaggio le primitive di comunicazione. Entrambi i concetti sono indispensabili per la costruzione del meccanismo di scambio di piani.

Una breve panoramica sui due linguaggi di comunicazione per agenti

più noti in letteratura è servita per evidenziare come la comunicazione tra agenti non abbia solo lo scopo di scambiare dati ma possa essere anche un valido strumento per comunicare qual è l'atteggiamento che produce un determinato modo di agire dell'agente.

Usando come pretesto il problema, non affrontato dalla quasi totalità dei linguaggi visti in rassegna, dell'assenza di conoscenza procedurale per la gestione di un determinato evento si è introdotto il modello Coo-BDI quale parziale soluzione ad esso.

Mediante la descrizione di un esempio si è visto come i due modelli BDI e Coo-BDI si pongono di fronte al problema del non saper gestire un evento, dando risalto a come il modello cooperativo di Coo-BDI porti il modo di agire dell'agente ad essere il più possibile simile al modo di agire umano.

Da un'osservazione di come l'estensione comunicativa di AgentSpeak(L) sia funzionale alla cooperazione si è pensato di sfruttare le caratteristiche di questo linguaggio esteso ed abbinarle al meccanismo di scambio di piani teorizzato in Coo-BDI al fine di ottenere un'unica architettura per agenti BDI altamente cooperativi, scritta in un linguaggio che sarà chiamato Coo-AgentSpeak.

Il modello Coo-BDI è stato adattato alle caratteristiche dell'interprete per AgentSpeak(L). Questo interprete è Jason e lo si è scelto principalmente in quanto in esso erano già presenti alcuni costrutti per la comunicazione, anche di conoscenza procedurale. Inoltre la sintassi di Jason supporta interamente la sintassi per la definizione degli agenti Coo-AgentSpeak. La fase di progettazione dell'interprete per Coo-AgentSpeak è stata preceduta da una fase di studio e test di Jason.

Lo studio fatto aveva come scopo l'acquisizione di un'approfondita conoscenza del funzionamento dell'interprete di Jason in modo da aver chiaro quali fasi dell'esecuzione potessero essere toccate nell'integrazione di Coo-BDI. Questa fase è stata utile in quanto ci ha permesso di rilevare alcuni bachi, prontamente segnalati agli autori dell'interprete, contribuendo alla sua messa a punto nelle versioni successive.

Nel progettare l'interprete abbiamo seguito, quando possibile, le indicazioni date in [4]. Quando abbiamo fatto scelte diverse queste sono state descritte valutando eventuali alternative e motivando le decisioni prese.

Le maggiori difficoltà che abbiamo incontrato nell'implementazione erano legate alla rappresentazione dei dati. In certi casi si sono usati dei "trucchi" per poter inserire elementi di un certo tipo in contesti in cui non erano previsti.

La fase di progettazione ci ha condotto alla realizzazione di un primo prototipo che deve ancora essere testato.

Nel complesso questo lavoro è riuscito ad evidenziare l'importanza della cooperazione applicata ai sistemi multi-agente. Jason è pensato per lavorare in ambiente distribuito. Recentemente si tende ad assimilare il sistema

distribuito allo schema generale del sistema multi-agente. I concetti tipici di agente autonomo, situato e sociale si trovano spesso all'interno di sistemi moderni quali sistemi di controllo, sistemi mobili, sistemi P2P, applicazioni Internet, sistemi aperti e altri.

Allo stesso modo il meccanismo di cooperazione basato sullo scambio di conoscenza procedurale può essere estremamente importante al fine di risolvere problemi tipici di questi sistemi.

Alcuni esempi di sistemi in cui la possibilità di modificare o accrescere la propria conoscenza procedurale trova applicazione sono quelli in cui le risorse a disposizione sono limitate come i PDA. Nei PDA lo spazio a disposizione è limitato ed è quindi utile un meccanismo che permetta di recuperare automaticamente delle informazioni utili destinate ad essere scartate dopo l'utilizzo.

Un altro esempio di sistema per cui il meccanismo di scambio di conoscenza può essere utile è dato dai sistemi con capacità riflessive. In questi sistemi il programma, ovvero l'agente, identifica come dato parte del proprio codice e può decidere di cambiarlo rimpiazzandolo dinamicamente con altro codice recuperato automaticamente.

Infine, la possibilità di accrescimento della propria conoscenza procedurale ha un ottimo campo di applicazione nei maggiordomi digitali. Un maggiordomo digitale adatta dinamicamente il suo comportamento alle indicazioni dell'utente. L'utente mostra attraverso un interfaccia le azioni da compiere che il sistema acquisisce come se fossero azioni di un piano recuperate esternamente.

Nel febbraio 2005 è stata pubblicata la release 0.6 di Jason. In essa sono presenti alcune innovazioni: l'esecuzione può essere sincrona o asincrona ed è presente un meccanismo per il debug. Sono stati introdotti molti cambiamenti che vanno a toccare diversi elementi dell'interprete.

I più interessanti dal punto di vista della realizzazione dello scambio di piani riguardano:

- *la gestione dei messaggi in arrivo.* Quando arriva un messaggio, indipendentemente dal suo tipo viene generato un evento che va gestito con un piano AgentSpeak(L) standard. I metodi `accepTell` e `acceptAchieve` inoltre sono stati sostituiti da `socAcc`.

- *la sintassi AgentSpeak(L).* Le variabili possono essere usato nei contesti in cui si ci aspetta un letterale.

In particolare, la modifica della gestione dei messaggi in ingresso potrebbe risolevere alcuni dei principali problemi rilevati nella realizzazione di Coo-AgentSpeak, come quelli legati alla ricezione di messaggi di tipo `tellHow`.

Un'estensione proposta è quella di adottare la più recente versione di

Jason per completare il funzionamento del prototipo e verificarlo sviluppando una applicazione in uno dei domini descritti precedentemente (PDA, ecc.).

# Bibliografia


[1] Agent Oriented Software Pty. Ltd. JACK$^{TM}$ Intelligent Agents Agent Manual. `http://www.agent-software.com/shared/demosNdocs/JACK_Manual_WEB/jack.html`.

[2] Agent Oriented Software Pty. Ltd. JACK$^{TM}$ Intelligent Agents Agent Manual - Events. `http://www.agent-software.com/shared/demosNdocs/JACK_Manual_WEB/events.html`.

[3] D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI model with cooperativity. In J. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *"Declarative Agent Languages and Technologies", First International Workshop, DALT 2003*, pages 109–134. Springer Verlag, 2003.

[4] D. Ancona, V. Mascardi, J. F. Hübner, and R. H. Bordini. Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Proceedings di AAMAS 2004 (Int. Conf. on Autonomous Agents and Multiagent Systems)*, pages 696–705, 2004.

[5] J. L. Austin. *How to Do Things with Words*. Oxford University Press, London, 1962.

[6] R. H. Bordini, A. L. C. Bazzan, R. de O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling. In C. Castelfranchi and W. L. Johnson, editors, *Proc. of the 1st International AAMAS Joint Conference*, pages 1294–1302. ACM Press, 2002.

[7] R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melburne, Australia, 14-18 July*, pages 409–416, 2003.

[8] R. H. Bordini, J. F. Hübner, et al. ***Jason**: A Java-based agentSpeak interpreter used with SACI for multi-agent distribution over the net*, manual, fifth release edition, November 2004.

[9] R. H. Bordini and A. F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3), 2004. To appear.

[10] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349 – 355, 1988.

[11] P. Busetta. Paolo Busetta's Curriculum Vitae - Experience. `http://www.cs.mu.oz.au/~paolo/curr.html#exp`.

[12] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents – components for intelligent agents in Java. *AgentLink News Letter*, 2, 1999.

[13] Cognitive Robotics Group Home Page. `http://www.cs.toronto.edu/cogrobo/`, 2002.

[14] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42, 1990.

[15] M. Dastani. 3APL platform. `http://www.cs.uu.nl/3apl/download/java/userguide.pdf`.

[16] M. Dastani. A Prototype of 3APL Interpreter. `http://www.cs.uu.nl/3apl/prototype.html`.

[17] W. H. Davies and P. Edwards. Agent-K: An integration of AOP & KQML. In T. Finin and Y. Labrou, editors, *Proceedings of the Workshop on Intelligent Information Agents. Held in conjunction with the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, 1994.

[18] G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog: A concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[19] G. De Giacomo, Y. Lespérance, H. J. Levesque, and S. Sardiña. On the semantics of deliberation in IndiGolog – from theory to implementation. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M. A. Williams, editors, *Proceedings of the 8th International Conference in Principles of Knowledge Representation and Reasoning (KR'02)*, pages 603–614. Morgan Kaufmann, 2002.

[20] K. S. Decker and V. R. Lesser. Quantitative Modeling of Complex Environments. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 2(4):215–234, January 1993.

[21] M. d'Inverno, K. V. Hindriks, and M. Luck. A formal architecture for the 3APL agent programming language. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *Proc. of the 1st International ZB Conference*, pages 168–187. Springer Verlag, 2000. LNCS 1878.

[22] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. Rao, and M. Wooldridge, editors, *Proc. of the 4th International ATAL Workshop*, pages 155–176. Springer Verlag, 1997. LNAI 1365.

[23] T. W. Finin. UMBC KQML web. `http://www.cs.umbc.edu/kqml/`.

[24] T. W. Finin, R. Fritzson, D. McDay, and R. McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference of Information and Knowledge Management (CIKM '94)*, pages 456–463. ACM Press, November 1994.

[25] M. Fisher and R. Owens. An Introduction to Executable Modal and Temporal Logics. In M. Fisher and R. Owens, editors, *Proceedings of the Workshop on Executable Modal and Temporal Logics. Held in conjunction with the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1–20. Springer Verlag, 1993. LNAI 897.

[26] Foundation for Intelligent Physical Agents. FIPA home page. `http://www.fipa.org/`.

[27] M. P. Georgeff and D. Morley AAII. dMARS, Agentis and deimos. Slides, URL:`http://www.agents.org.au/19990625AAII-UoM990624.pdf`.

[28] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

[29] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. Agent Tcl. In W. Cockayne and M. Zyda, editors, *Mobile Agents: Explanations and Examples*. Manning Publishing, 1997.

[30] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[31] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming with declarative goals. In *Intelligent Agents VI - Proceedings of the 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL'2000)*, pages 228–243. Springer, 2000. Lecture Notes in AI.

[32] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Semantics of communicating agents based on deduction and abduction. Technical report, 2000.

[33] M. J. Huber. Intelligent reasoning systems. `http://www.marcush.net/IRS/index.html`.

[34] M. J. Huber. Jam agents in a nutshell. `http://www.marcush.net/IRS/Jam/Jam-man-01Nov01.doc`.

[35] M. J. Huber. JAM: a BDI-Theoretic Mobile Agent Architecture. In *Agents*, pages 236–243, 1999.

[36] J. F. Hübner and J. S. Sichman. *SACI — Simple Agent Communication Infrastructure*, 2003. SACI Home Page: `http://www.lti.pcs.usp.br/saci/`.

[37] J. F. Hübner and J. S. Sichman. *SACI — Simple Agent Communication Infrastructure*, July 2003. Programming Manual. `http://www.lti.pcs.usp.br/saci/doc/programmingGuide.pdf`.

[38] J. F. Hübner, J. S. Sichman, and O. Boissier. *Moise+ Tutorial*. São Paulo, Brazil, 2002. URL: `http://www.lti.pcs.usp.br/moise/doc/tutorial.pdf`.

[39] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.

[40] S. A. Kripke. Semantical Analysis of Modal Logic I: Normal Modal Propositional Calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[41] S. A. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[42] S. A. Kripke. Semantical Analysis of Modal Logic II: Non-Normal Modal Propositional Calculi. In Addison, Henkin, and Tarski, editors, *The theory of models*, pages 206–220, Amsterdam, 1965. North Holland Publishing Co.

[43] Y. Labrou, T. W. Finin, and Y. Peng. Agent communication languages: The current landscape. *IEEE Intelligent Systems*, 14(2):45–52, 1999.

[44] J. Lee, M. J. Huber, P. G. Kenny, and E. H. Durfee. UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS'94)*, pages 842–849, Houston, Texas, 1994.

[45] Legolog Home Page. `http://www.cs.toronto.edu/cogrobo/Legolog/index.html`, 2000.

[46] H. J. Levesque and M. Pagnucco. Legolog: Inexpensive Experiments in Cognitive Robotics. `http://www-i5.informatik.rwth-aachen.de/LuFG/cogrob2000/AcceptedPapers.html`.

[47] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[48] R. Machado and R. H. Bordini. Running AgentSpeak(L) agents on SIM_AGENT. In *Pre-Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001),August 1-3, 2001, Seattle,WA*, pages 158–174, 2001.

[49] V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent sogtware agents. TLP 4(4):429–494, 2004.

[50] Massachusetts Institute of Tecnology. The Scheme Programming Language. `http://www.swiss.ai.mit.edu/projects/scheme/`.

[51] J. Mayfield, Y. Labrou, and T. W. Finin. Evaluation of KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Proc. of the 2nd International ATAL Workshop*, pages 347–360. Springer Verlag, 1995.

[52] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in *Semantic Information Processing*, M. Minsky ed., The MIT Press, 1968, 110–117.

[53] A. F. Moreira and R. H. Bordini. An operational semantics for a BDI agent-oriented programming language. In J.-J. Ch. Meyer and M. Wooldridge, editors, *Proceedings of Workshop on Logics for Agent-Based Systems (LABS-02), held in with the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, pages 45–59, Toulouse, 2002.

[54] A. F. Moreira, R. Vieira, and R. H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In João Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies, Proceedings of the First International Workshop (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia (Revised Selected and Invited Papers)*, number 2990 in Lecture Notes in Artificial Intelligence, pages 135–154, Berlin, 2004. Springer-Verlag.

[55] K. L. Myers. User Guide for the Procedural Reasoning System. Technical report, SRI International AI Center Technical Report, SRI International, Menlo Park, CA, 1997.

[56] K. L. Myers. User guide for the procedural reasoning system. Technical report, Artificial Intelligence Center, Menlo Park, CA, 1997.

[57] K. L. Myers and D. E. Wilkins. The Act Formalism, Version 2.2. Technical report, SRI International AI Center Technical Report, SRI International, Menlo Park, CA, 1997.

[58] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Proceedings of Computer-Aided Verification, CAV'96*, pages 411–414. Springer Verlag, 1996. LNCS 1102.

[59] F. Pirri and R. Reiter. Some Contributions to the metatheory of the Situation Calculus. *Journal of the ACM*, 46(3):325–361, 1999.

[60] PLT People. DrScheme. `http://www.drscheme.org/`.

[61] 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: Procedings of the Seventh European workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55. Springer-Verlag, 1996. LNAI 1038, URL: `http://www.aaii.oz.au/bios/rao.html`.

[62] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI–architecture. In R. Fikes and E. Sandewall, editors, *Proc. of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484, San Mateo, CA, 1991. Morgan Kaufmann.

[63] A. S. Rao and M. P. Georgeff. BDI agents: from theory to practice. In V. Lesser and L. Gasser, editors, *Proc. of the 1st International ICMAS Conference*, pages 312–319, 1995.

[64] R. Reiter. On Knowledge-Based Programming with Sensing in the Situation Calculus. *ACM Transactions on Computational Logic (TOCL)*, 2(4):433–457, 2001.

[65] R. L. Rivest. SEXP—(S-expressions). `http://theory.lcs.mit.edu/~rivest/sexp.html`.

[66] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, 1969.

[67] S. Shapiro, Y. Lespérance, and H. J. Levesque. The Cognitive Agent Specification Language and Verification Environment for Multiagent Systems. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 19–26. ACM Press, 2002.

[68] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1), 1993.

[69] A. Sloman and R. Poli. SIM_AGENT: A toolkit for exploring agent design. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II – Proceedings of the Second International Workshop on Agent Theories, Architectures, and Languages (ATAL'95), held as part of IJCAI'95, Montréal, Canada, August 1995*, pages 392–407, Berlin, 1995. Springer-Verlag. LNAI 1037.

[70] J. M. Spivey. *The Z Notation: A Reference Manual, 2nd edition.* Prentice Hall International (UK) Ltd., 1992.

[71] L. Sterling and E. Y. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, 1986.

[72] S. R. Thomas. The PLACA agent programming language. In *ECAI Workshop on Agent Theories, Architectures, and Languages*, pages 355–370. Springer Verlag, 1994. LNCS 890.

[73] B. van Riemsdijk, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in Dribble: from beliefs to goals using plans. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proc. of AAMAS-03*, pages 393–400. ACM Press, 2003.

[74] T. Wagner, A. Garvey, and V. R. Lesser. Criteria-directed heuristic task scheduling. *International Journal of Approximate Processing, Special Issue on Scheduling*, 19(1-2):91–118, 1998.

[75] G. Weiss. *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence.* The MIT Press, 1999.

[76] M. Winikoff. AgentTalk Home Page. `http://goanna.cs.rmit.edu.au/~winikoff/agenttalk`, 2001.

# Appendice A

# Linguaggio di specifica Z

Il linguaggio di specifica formale, Z, è basato sulla teoria degli insiemi e sul calcolo dei predicati del prim'ordine. Estende l'uso di questi linguaggi fornendo un tipo matematico aggiuntivo conosciuto come *schema type*.

Lo schema Z ha due parti: la parte superiore dichiarativa, in cui si dichiarano le variabili e i loro tipi, e la parte inferiore dei predicati, in cui si collegano e vincolano queste variabili. Il tipo di ogni schema può essere considerato come il prodotto cartesiano dei tipi di ognuna di queste variabili, senza alcun ordine, ma limitato dallo schema dei predicati.

In Z la modularità è semplificata fornendo gli schemi da includere entro altri schemi. Si può scegliere una variabile di stato, *var*, di uno schema, *schema*, scrivendo *schema.var*.

Per introdurre un tipo in Z, per il quale si vuole astrarre dal contenuto reale degli elementi del tipo, si usa la nozione di *given set*. Si scrive $[NODE]$ per rappresentare l'insieme di tutti i nodi. Se si vuole stabilire che una variabile assuma un insieme di valori o una coppia ordinata di valori si scrive $x : \mathbb{P}NODE$ e $x : NODE \times NODE$, rispettivamente.

Un tipo *relation* esprime delle relazioni tra due tipi esistenti, conosciuti come i tipi *source* e *target*. Il tipo di una relazione con sorgente $X$ e target $Y$ è $\mathbb{P}(X \times Y)$. Una relazione è quindi un'insieme di coppie ordinate.

Quando nessun elemento del tipo della source può essere collegato a due o più elementi del tipo del target, la relazione è una *funzione (function)*. Una funzione *totale* ($\rightarrow$) è una per cui ogni elemento nell'insieme source è legato, mentre una funzione *parziale* ($\nrightarrow$) è una per cui non ogni elemento nella source è legato. Una sequenza (*seq*) è un tipo speciale di funzione dove il dominio è l'insieme di numeri contigui da 1 al numero di elementi nella sequenza.

Il *dominio* (dom) di una relazione o funzione comprende quegli elementi nell'insieme source che sono legati. La *range* (ran) comprende quegli elementi nell'insieme target che sono legati.

Gli insiemi di elementi possono essere definiti usando la comprensione tra insiemi. Il modo di scrivere predicati in Z è non-standard.

L'espressione, $\mu a : A | P$, sceglie l'unico elemento dal tipo $A$ che soddisfa il predicato $P$.

Un estratto della notazione da usare è dato nella Tabella A.1. Per una descrizione più completa del linguaggio Z si può fare riferimento a [70].

**Definitions and declarations**

| | |
|---|---|
| $a, b$ | Identifiers |
| $p, q$ | Predicates |
| $s, t$ | Sequences |
| $x, y$ | Expressions |
| $A, B$ | Sets |
| $R, S$ | Relations |
| $d;\ e$ | Declarations |
| $a == x$ | Abbreviated definition |
| $[a]$ | Given set |
| $A ::= b\langle\langle B \rangle\rangle$ $\mid c\langle\langle C \rangle\rangle$ | Free type declaration |
| $\mu\, d \mid P$ | Unique value ascription |
| **let** $a == x$ | Local variable definition |

**Logic**

| | |
|---|---|
| $\neg\, p$ | Logical negation |
| $p \wedge q$ | Logical conjunction |
| $p \vee q$ | Logical disjunction |
| $p \Rightarrow q$ | Logical implication |
| $p \Leftrightarrow q$ | Logical equivalence |
| $\forall X \bullet q$ | Universal quantification |
| $\exists X \bullet q$ | Existential quantification |

**Sets**

| | |
|---|---|
| $x \in y$ | Set membership |
| $\varnothing$ | Empty set |
| $A \subseteq B$ | Set inclusion |
| $\{x, y, \ldots\}$ | Set of elements |
| $(x, y, \ldots)$ | Ordered tuple |
| $A \times B \times \ldots$ | Cartesian product |
| $\mathbb{P}\, A$ | Power set |
| $\mathbb{P}_1\, A$ | Non-empty power set |
| $A \cap B$ | Set intersection |
| $A \cup B$ | Set union |
| $A \setminus B$ | Set difference |
| $\bigcup A$ | Generalized union |
| $\#A$ | Size of a finite set |
| $\{d;\ e \ldots \mid p \bullet x\}$ | Set Comprehension |

**Relations**

| | |
|---|---|
| $A \leftrightarrow B$ | Relation |
| $\mathrm{dom}\, R$ | Domain of a relation |
| $\mathrm{ran}\, R$ | Range of a relation |
| $R^{\sim}$ | Inverse of a relation |
| $A \lhd R$ | Anti-domain restriction |
| $R \oplus S$ | Relational overriding |

**Functions**

| | |
|---|---|
| $A \nrightarrow B$ | Partial function |
| $A \rightarrow B$ | Total function |

**Sequences**

| | |
|---|---|
| $\mathrm{seq}\, A$ | Set of finite sequences |
| $\mathrm{seq}_1\, A$ | Non-empty set of sequences |
| $\langle\rangle$ | Empty sequence |
| $\langle x, y, \ldots \rangle$ | Sequence |
| $s \frown t$ | Sequence concatenation |
| $\mathit{head}\ s$ | First element of sequence |
| $\mathit{tail}\ s$ | All but first element |

**Schema notation**

| | |
|---|---|
| $S$<br>$d$<br>$p$ | Vertical schema |
| $d$<br>$p$ | Axiomatic definition |
| $S$<br>$T$<br>$d$<br>$p$ | Schema inclusion |
| $\Delta S$<br>$S$<br>$S'$ | Operation schema |
| $z.a$ | Component selection |

**Conventions**

| | |
|---|---|
| $a?$ | Input to an operation |
| $a$ | State component before operation |
| $a'$ | State component after operation |
| $S$ | State schema before operation |
| $S'$ | State schema after operation |
| $\Delta S$ | Change of state ($S \wedge S'$) |
| $\Xi S$ | No change of state |
| $OP_1 \fatsemi OP_2$ | Operation composition |

Tabella A.1: Notazione Z