ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Elettronica, Informatica e Telecomunicazioni

LINGUAGGI DI PROGRAMMAZIONE AD AGENTI:
JASON COME CASO DI STUDIO

Elaborata nel corso di: Sistemi Operativi

Tesi di Laurea di:
NICOLO' CARPIGNOLI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2013-2014
SESSIONE II

PAROLE CHIAVE

Jason

Agenti

BDI

Sistemi Multi-Agente

AgentSpeak

Alla mia famiglia

Indice

Introduzione

L'informatica è al giorno d'oggi al centro di un incredibile sviluppo e tumulto innovativo; è la scienza che coniuga ogni sviluppo tecnologico, il quale inevitabilmente, d'ora in poi, verrà in qualche modo condizionato dalle innovazioni di tale disciplina.
Fra gli innumerevoli nuovi trend che si sono affacciati prepotentemente negli ultimi anni sul panorama informatico, il paradigma per la programmazione ad agenti è uno dei più interessanti, in accordo con i recenti e prossimi sviluppi della tecnologia in generale.
Questa tesi tratterà tale argomento partendo da un punto di vista generico ed introduttivo volutamente esaustivo, per poi concentrarsi su una specifica tecnologia implementativa, mostrandola infine nella pratica.
Il primo capitolo è un introduzione sui sistemi multi-agente, con definizioni, esempi, accenni di tecnologie; nel secondo capitolo verrà analizzata nel dettaglio una di queste tecnologie, per poi concludere nel terzo capitolo con un esempio applicativo (su un caso di studio concreto) basato sulla tecnologia citata appunto nel capitolo precedente.

È importante sottolineare come in documenti di questo tipo, specie quando si trattano argomenti altamente tecnici, vi sia l'utilizzo di terminologie non ancora d'uso comune rendendo il tutto spesso poco comprensibile a chi non è esperto del settore. In questo documento invece l'autore cercherà di trattare l'argomento in maniera sì tecnica ma allo stesso tempo comprensibile anche per i 'non addetti ai lavori', cercando di mantenere un linguaggio adatto al carattere scientifico del documento stesso, ma accompagnando ugualmente nella lettura e nella comprensione il lettore appassionato e curioso, ma non esperto.
Come Socrate sosteneva, "l'ignoranza è il principio di ogni male", e la conoscenza è da sempre il bene più prezioso da condividere nella società moderna, è di mio auspicio riuscire a coinvolgere il più possibile chiunque verso quella

che sarà l'informatica e quindi il mondo del futuro, divulgare le conoscenze a coloro che, curiosi, vogliono apprendere, di modo che un domani possano utilizzare consciamente e con giudizio tutte le meraviglie che il progresso ci avrà fornito.

Capitolo 1

I Sistemi Multi agente

In questa prima parte verranno definiti e trattati i principi base della programmazione Multi Agente e le peculiarità del mondo degli agenti nello specifico.Verrà chiarito cosa sono gli agenti e i sistemi multi-agente, come nascono, e perché nascono. Verrà fatto un approfondimento sulla differenza fra oggetti ed agenti, di modo che il lettore, che si presume abbia almeno dei rudimenti dell'object oriented programming, possa capir ancora meglio le peculiarità di questo nuovo paradigma. Verranno infine presentati concetti e tecnologie per far comunicare e coordinare più agenti fra loro, e mostrati dei casi d'uso in cui gli agenti già al giorno d'oggi tornano utilissimi e d'uso concreto nell'informatica odierna.

Il primo capitolo è quindi pensato per essere un'introduzione al mondo degli agenti, fruibile a chiunque e necessario per comprendere a pieno Jason e proseguire quindi in maniera consapevole e preparata alla lettura dei restanti due capitoli.

1.1 Agenti e sistemi multi agente

L'approccio classico nel trattare argomenti tecnici come questo è quello di partire con una definizione di modo da fissare subito dei concetti cardine nel lettore, per poi muoversi durante l'approfondimento consapevoli che il lettore potrà ben seguirci nei ragionamenti. Dunque, l'agente, dopo diversi anni di dibattiti sulla definizione formale, viene trattato in informatica come

Un sistema situato in un certo ambiente, capace di azioni autonome sul-

l'ambiente stesso cercando di raggiungere certi obiettivi.

La definizione dice molto più di quel che può sembrare. Le parole chiave sono ambiente, autonomia e obiettivi; sono termini da situare attentamente nel contesto di modo da capire bene di cosa si sta parlando. Parliamo di ambiente nel senso di sistema in cui uno o più agenti (per ora soffermiamoci ad analizzare il singolo) possono percepire informazioni, e in base alle loro azioni, modificarlo. In genere, l'ambiente viene virtualizzato e visto dall'agente simbolicamente, di modo che possa computare e leggere in maniera opportuna le informazioni. Ad esempio, per un agente che deve monitorare un certo isolato l'ambiente sarà la realtà rappresentata sotto forma di griglia dove i vari ostacoli come alberi, case e lampioni verranno modellati come elementi presenti nella griglia. L'agente può quindi percepire informazioni sull'ambiente, informazioni che possono cambiare dinamicamente nel tempo, e modificare lo stesso per raggiungere certi obiettivi. Parlando appunto di obiettivi, sono semplicemente azioni che si vogliono conseguire, e l'agente farà di tutto in suo potere (in base alle sue credenze sull'ambiente, le sue possibili azioni,ecc) per fare ciò. Tali aspetti verranno approfonditi in seguito. Infine, l'autonomia è l'aspetto veramente fondamentale che contraddistingue un agente. Autonomia in questo caso significa essere capaci di prendere decisioni per raggiungere un determinato obiettivo. E' ciò che li contraddistingue da i classici sistemi distribuiti: questi ultimi sono programmati per un obiettivo comune, i sistemi multi agente sono formati da agenti ognuno con i propri obiettivi, ed ognuno dei quali cercherà di conseguirli con determinate azioni; ciò non esclude che obiettivi diversi siano in conflitto, e ciò porta a dover prevedere casistiche di coordinazione tra agenti. Inoltre, l'autonomia prevede che per raggiungere certi obiettivi, tra le varie scelte autonome ci sia anche la cooperazione, quindi l'organizzazione sociale tra diversi agenti.

E' ora semplice capire che un sistema multi agente sia un sistema dove più agenti presenti nello stesso ambiente si muovano con l'intento di raggiungere ognuno i propri obiettivi. Ogni agente è di per sè un sistema complesso da programmare, allo stesso modo e in misura maggiore lo sono i sistemi multi agente (da qui in poi abbreviati come MAS). E' inoltre da puntualizzare che un MAS può avere un alto grado di non determinismo dovuto ai possibili fallimenti delle azioni degli agenti, e che inoltre ogni agente ha una visione parziale e probabilmente distorta dell'ambiente stesso: ciò

rende possibile modellare i MAS ad alto livello rendendoli molto affini con il mondo reale, ma allo stesso tempo aumenta la difficoltà nel progettare gli stessi.

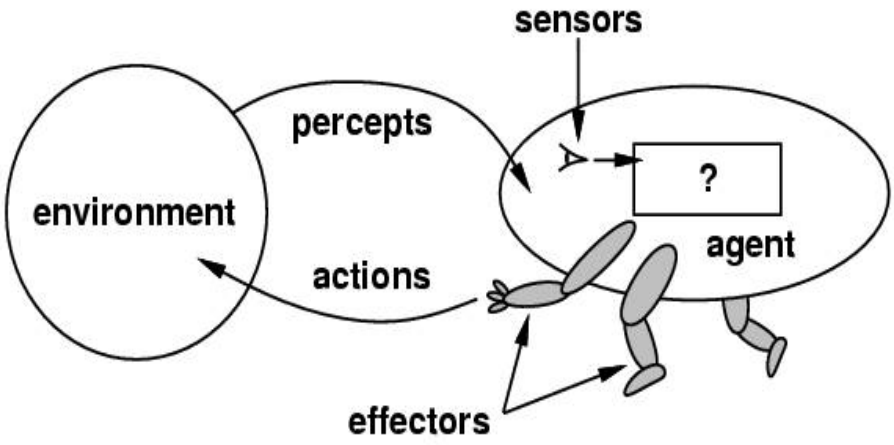

Figura 1.1: Un agente e l'ambiente

L'immagine aiuta subito a capire lo stretto legame fra agente e ambiente, così come noi stessi nella realtà abbiamo continui rapporti con l'ambiente che ci circonda, spostandoci sullo stesso, cambiando la posizione di certi oggetti,ecc.

Andiamo ora brevemente a comprendere la visione che ha portato allo sviluppo degli agenti e i campi in cui questo paradigma potrà giocare un ruolo chiave.

1.1.1 Perché gli agenti

Non è questione immediata individuare uno specifico anno o periodo di nascita del concetto di agente, ma si può senza dubbio collocare le primi idee al riguardo (ovvero di agenti intesi come sistemi intelligenti capaci di computare) con le idee di Turing sulla Intelligenza Artificiale; non a caso, ancora oggi la IA e gli agenti sono concetti strettamente legati, per loro natura. Turing stesso col suo famoso test implicitamente parlava di agente, poichè una macchina (intelligente) capace di arguire con un umano, senza che quest'ultimo riesca a capirne la natura artificiale, ingloba gli aspetti di

un comportamento autonomo, attivo e proattivo tipico degli agenti come li intendiamo oggi. Dal 1969 al 1985 grossi progressi vennero fatti, è da citare STRIPS, un primo algoritmo di planning che si basava sulla rappresentazione simbolica del mondo e sulle decisioni deduttive. Partendo da STRIPS però, diverse community di ricercatori proposero altri approcci per l'IA, come la 'Behavioural IA', in cui il comportamento generale veniva visto come un insieme di piccoli e semplici corsi d'azione e veniva abbandonata l'idea di una forte simbolizzazione dell'ambiente. L'incremento degli studi e pure i dibattiti accesi fra le community non fecero altro che infiammare il focolaio dei ricercatori rendendo l'area delle architetture degli agenti un settore importante dell'IA. Anche i MAS ebbero origine, come concezione, con i primi approcci all'IA post Turing, seppure si parla di primo MAS con il famoso sistema Blackboard, una sorta di (letteralmente) lavagna, dove più attori, in differenti momenti possono persistere la propria conoscenza in una memoria centrale (la lavagna appunto) di modo che ognuno di essi possa attingere al lavoro altrui per il raggiungimento del proprio obiettivo. La lezione di storia potrebbe proseguire con il trattare il modello ad Attori e l'introduzione dei primi standard e linguaggi per la comunicazioni inter agente, come KQML e KIF, ma ci si limita a citarli volendo questa esser soltanto un'introduzione ad un ben più interessante approfondimento, ovvero perché gli agenti sono così tanto seguiti e studiati e cosa potrebbero apportare al mondo dell'informatica (e non solo) nel prossimo futuro.

1.1.2 Il futuro contributo degli agenti

Gli agenti si sono già rivelati tutt'oggi, seppur siano ancora un trend in pieno sviluppo, un'ottima soluzione per applicazioni e sistemi legate ai Web Services, alla gestione di reti distribuite e a marketplace basati sul Web. Ed è indubbio che l'informatica stia virando verso trend come l'ubiquitous computing, i sistemi pervasivi, e l'Internet of Things; ciò che accomuna questi elementi sono caratteristiche come l'autonomia, la capacità collaborativa e comunicativa degli elementi, la costante connettività con la rete e con sè stessi nonostante la loro distribuzione nello spazio. Insomma, è chiaro che nel breve futuro avremo bisogno di un paradigma di programmazione capace di farci vedere il mondo sotto il punto di vista dell'interazione, e sembra proprio che il paradigma ad agenti sia un forte candidato a tale scopo. Approfondiremo meglio le differenze e analogie fra gli agenti e gli oggetti in

seguito, qua ci soffermeremo su alcuni settori in cui gli agenti potranno, in un futuro molto prossimo, dare contributo ancora più importante di quello che oggi, in maniera un pò acerba, stanno già dando.

Parlando di Self Interested Computation, trattiamo di computazioni in cui il sistema (o la macchina stessa) è interessata nella riuscita di tale operazione, magari a scapito di altre. Un classico esempio è un sistema online di e commerce, come Amazon o eBay, dove in un futuro si potrebbe immaginare che a lanciare le contrattazioni per un eventuale acquisto siano macchine programmate ad agenti, e non l'utente che fisicamente impartisce i propri comandi ad un sistema passivo. Lo stesso Web Semantico lanciato come (almeno nel breve termine) un'utopia, è una settore di grande interesse per il paradigma ad agenti. Seguendo la definizione dello stesso Berners Lee

Ho un sogno per il Web in cui i computer siano capaci di analizzare i dati dalla Rete, i contenuti e le interazioni fra computer e utenti; tale cosa deve ancora emergere, ma quando lo farà (...) gli 'agenti intelligenti' che le persone hanno sognati per anni saranno realtà, e sarà possibile vedere trasazioni economiche, pratiche burocratiche e così via realizzate da macchine con altre macchine

è praticamente esplicito il riferimento agli agenti come noi stessi li intendiamo oggi. E' inoltre importante ragionare sulla natura degli agenti: non sono forse essi, con tutti i loro limiti, una nostra rappresentazione? Se si cerca di rappresentare l'essere attivi, proattivi, collaborativi su larga scala, non si cerca dunque di simulare una sorta di società umana? E' quindi nata un'importante visione degli agenti come strumento per studiare in qualche modo la società umana, nei modi più disparati; in genere si parla comunque di simulazione, dalle applicazioni militari, ai piani di evacuazione in certe zone critiche, o alla simulazione di vita di una piccola comunità di agenti, per cercare di prevedere comportamenti in situazioni critiche. Si parla ovviamente di simulazione sommarie e con tantissimi limiti, dato che è impossibile, e probabilmente rimarrà così per sempre, prevedere e controllare tutte le possibilità e varianti in gioco per questi ambiti. Un importante progetto a proposito è EOS (1995), dove si è cercato di simulare la vita di uomini del Paleolitico nel Sud della Francia, per cercare di capire i cambiamenti che hanno portato all'inevitabile maggior complessità della società del tempo. Indubbiamente avremo a che fare con gli agenti, ogni program-

matore nel prossimo futuro ne sentirà parlare, avrà a che fare con essi come le ultime generazioni hanno studiato e lavorato con il paradigma ad oggetti. Gli oggetti, appunto, sono dunque morti? C'è ancora spazio per loro nell'informatica? Cosa li differenzia in fondo dagli agenti? Vedremo che le differenze sono molto più di quelle che si pensano, e che, come spesso accade in ogni ambito, ogni tecnologia ha i suoi settori in cui si esprime al meglio ed altri in cui è meno consigliata rispetto ad altre.

1.2 Agenti vs Oggetti

Si potrebbero vedere gli agenti come una logica evoluzione degli oggetti, così come gli oggetti hanno affiancato ed evoluto, se vogliamo, la programmazione procedurale fino ad oggi. Con tutti i vantaggi che comportano gli agenti, è chiaro che in ogni caso in un futuro non vi sarà probabilmente la morte del paradigma ad oggetti, poichè vi saranno sempre certi tipi di applicazioni e sistemi per cui gli agenti non saranno il miglior paradigma da scegliere.

Ciò detto, riconosciamo che agenti ed oggetti hanno certe analogie, come l' information hiding (nascondere le informazioni sul proprio stato se non è strettamente necessario rivelarle) e il possedere un proprio stato interno e comportamento; ma sono le differenze che rendono sostanzialmente i due paradigmi molto diversi, e per questo, adatti a scopi divergenti. Innanzitutto gli agenti sono autonomi, gli oggetti hanno un comportamento che in nessun modo è definibile autonomo o intelligente; un oggetto espone un metodo, che può essere chiamato da chiunque, e l'oggetto in questione non potrà far altro che eseguirlo passivamente. Gli agenti invece, vedendosi arrivare una richiesta per l'esecuzione di un azione, potranno scegliere se eseguirla o meno; ad esempio, se ciò andrà in conflitto con gli obiettivi dell'agente ad esempio, potranno declinare, ed inoltre potranno decidere, in caso, come eseguire tale azione, scegliendo tra diversi comportamenti disponibili. Ciò si collega al fatto che a differenza degli oggetti, gli agenti hanno inoltre un comportamento attivo e proattivo. Potremo riassumere il concetto con la massima

Objects do it for free; agents do it because they want to.

Inoltre gli agenti sono sostanzialmente multi-threaded, mentre gli oggetti possono sì essere lanciati parallelamente, ma non nacquero con tale peculiarità, introdotta solo successivamente. Parliamo quindi di differenze che, enumerate, non compongono un elenco corposo, ma racchiudono concetti sostanzialmente agli antipodi, rendendo così necessario suddividerli in due paradigmi diversi, che inevitabilmente diverranno pane quotidiano per progettisti di sistemi software del prossimo futuro.

1.3 Programmare ad Agenti

E' indubbio che quindi che gli agenti costituiscano un vero e proprio paradigma di programmazione, il quale comporta una serie di stili e di regole da seguire per produrre software in maniera non eroica. Per la modellazione di sistemi software ad agenti inoltre è stato introdotta un'estensione del linguaggio di modellazione UML (in genere utilizzato per il paradgima ad oggetti), l'AUML, nato nei primi anni 2000. La filosofia è riusare e riadattare meccanismi e concetti dell'UML, e quando non possibile, introdurne di nuovi; ad esempio, si dà importanza al concetto di ruolo, estendendo quello già presente nell'UML, e al supporto per thread di interazione concorrenti. Per la comunicazione e interazione fra agenti l'AUML è strettamente legato agli sviluppi del protocollo FIPA, di cui discuteremo oltre. In generale poi, possono essere individuati dei pilastri nella programmazione ad agenti, ovvero semplici linee guida e reminder che aiutano il progettista a non perdere di vista la visione e i concetti di questo nuovo paradigma.

Innanzitutto è bene tenere presente che gli agenti non sono, e non saranno la soluzione definitiva ad ogni problema software d'ora in avanti: gli altri paradigmi di programmazione avranno campi di interesse che difficilmente moriranno. Inoltre è bene ricordarsi del fatto che gli agenti sono pur sempre software, e come tale hanno tutte le limitazioni che i sistemi software hanno. Fanno uso delle possibilità offerte dall'IA, ma non danno contributi in questo campo e le loro abilità di intelligenza e autonomia sono limitate allo stato dell'arte dell'IA.

Bisogna ricordarsi che i sistemi multi-agente sono sistemi multi-threaded, e con ciò tenere bene a mente le lezioni imparate su questi sistemi poichè bisogna trattarli come tali, seppure programmiamo 'ad agenti'. Chiaramente

è inutile adottare una soluzione ad agenti, intrinsecamente concorrente, per problemi che non hanno bisogno affatto di un approccio multi-threaded.

In generale poi sono da evitare un numero troppo elevato di agenti nel proprio sistema, poichè oltre ad aumentare la difficoltà di progettazione e gestione dello stesso, non sempre si è capaci di prevedere l'esatto numero di agenti effettivamente utili; d'altro canto è bene non caricare troppe responsabilità e azioni su un numero limitato di agenti.

Ecco quindi presentati alcuni consigli per una buona programmazione ad agenti, i quali assieme ad un uso sapiente di un linguaggio di modellazione adatto e a determinati pattern (di cui se ne parlerà nella seconda parte di questo documento) aiuteranno sicuramente nello sviluppo software.

Analizziamo ora brevemente le tipologie più importanti di Agenti, per poi fare un focus sulla coordinazione e comunicazione inter-agente.

1.4 Tipologie di Agenti

E' già stata data una definizione di agente nei primi periodi di questa parte introduttiva, evidenziando l'importanza degli aspetti autonomi quali reattività, proattività, attività e abilità sociali; abbiamo praticamente parlato in maniera implicita di quel tipo di agenti definiti 'Intelligenti'. Esistono però altri tipi di tale entità, caratterizzati da proprietà diverse più o meno importanti.

1.4.1 Deductive reasoning agents

L'approccio tradizionale per realizzare sistemi intelligenti ed in particolare l'ambiente in cui essi operano, è definire una simbologia e una rappresentazione simbolica dello stesso, di modo che le entità ivi operanti vedano soltanto tale rappresentazione. Quando le rappresentazioni sono formule logiche, la manipolazione sintattica diventa deduzione logica, o 'proving' di teoremi. Il progettista avrà quindi due compiti principali da assolvere per muoversi adeguatamente, ovvero tradurre il mondo reale in un'esatta rappresentazione simbolica, e far sì che gli agenti siano in grado di capirla ed operarvi. Tali agenti, che agiranno come una sorta di 'theorem provers', avranno una teoria logica (una sorta di specifica) tramite cui sapranno come operare con il resto delle rappresentazioni logiche presenti per raggiungere

i propri obiettivi. Parliamo insomma dei cosidetti 'Deductive Reasoning Agents'.

1.4.2 Practical reasoning agents

Ci si accorge però che noi stessi, come umani, non pratichiamo tutti i giorni soltanto ragionamenti logici, tutt'altro; la maggior parte del tempo cerchiamo di risolvere problemi in maniera pratica, con ragionamenti non puramente logici. Il 'pratical reasoning', ragionamento pratico, è quello che ci fa decidere ogni giorno oltre al 'cosa' fare, anche 'come' farlo, è ciò che ci permette ad esempio di scegliere il miglior mezzo pubblico per raggiungere un determinato appuntamento, e così via. I 'Practical Reasoning Agent' hanno appunto queste priorità, basate su un ragionamento pratico che consiste in una fase di deliberazione (ovvero l'obiettivo, le intenzioni dell'agente) e in un'altra fase più pratica detta 'means-end reasoning', ovvero 'come' svolgere tali intenzioni (i cosiddetti piani d'azione).

1.4.3 Agenti Reattivi e Ibridi

Nella seconda metà degli anni ottanta peò, molti ricercatori hanno messo in dubbio l'estrema simbolizzazione dell'ambiente e in generale l'IA simbolica, sostenendo non fosse il giusto approccio per agenti che dovevano muoversi in ambienti strettamente legati al tempo. In generale, gli agenti Reattivi portano novità come approcci 'behavioural', ovvero un comportamento definito da altri molto più semplici, 'situato', dato che ogni agente dovrebbe essere legato strettamente all'ambiente in cui si trova, e 'reattivo' appunto, con la pretesa che esso semplicemente risponda ai cambiamenti dell'ambiente senza ragionarvici sopra.

Altre tipi di agenti da citare sono gli Ibridi, ovvero agenti in cui nel loro progetto si cerca di dividere in layer ben separati gli aspetti attivi e proattivi del loro comportamento.

1.4.4 Agenti Mobile

Infine, una tipologia molto importante per le comunità focalizzate sui linguaggi di programmazione e le community object-oriented development è senz'altro quella degli agenti Mobile. L'idea di base è che tali agenti siano

indirizzabili nella rete e che possano riprendere la propria computazione in nodi differenti, trasmettendo il programma e il loro stato. Inizialmente nacque con la necessità di sostituire le chiamate di procedura remote, ovvero una chiamata sincrona, quindi bloccante per colui che la esegue, verso procedure presenti in un altro nodo della rete; il problema è appunto relativo alle performance e al fatto che il chiamante si sblocchi soltanto al ritorno della procedura (con il relativo risultato). Inviando invece un programma vero e proprio, quindi l'agente, la chiamata diventa asincrona e l'agente può computare indipendentemente ed autonomamente nel nodo richiesto, per poi ritornare con il risultato. I problemi qua, più o meno risolti da linguaggi come Java ad esempio, riguardano la serializzazione, ovvero cosa deve essere inviato o meno tramite la rete, i problemi relativi all'ambiente di esecuzione (diversi sistemi operativi e piattaforme possono portare a incompatiblità) e la sicurezza. Proprio quest'ultimo aspetto è particolarmente spinoso dato che un giusto compromesso sulle informazioni da passare o meno è arduo da conseguire, visto che chiaramente con un numero elevato di dati si ha una maggior qualità dell'esecuzione, ma al tempo stesso una maggior probabilità di vedersi rubati da terzi dati sensibili.

1.5 Comunicazione e coordinazione inter-agente

In generale, l'interazione è l'aspetto cruciale nei sistemi multi-agente, e concerne la comunicazione fra gli stessi nonch è la coordinazione, spesso per conseguire in maniera adeguata obiettivi in comune.

1.5.1 Comunicazione

Inanzitutto è cruciale per ogni tipo di comunicazione fra due entità che vi sia una sorta di comune accordo nella terminologia da usare; non si può pretendere che due agenti, nell'atto di trattare la compravendita di un bene ad esempio, utilizzino termini diversi nel definire le proprietà dello stesso, pur intendendo lo stesso concetto. Parliamo di ontologie come una rappresentazione formale di un dominio di interesse, ed è la base per ogni dialogo inter-agente. Le ontologie poi possono essere di diverso tipo, dai vocabolari ai glossari, fino ad arrivare ai thesaurus (sinonimi) o alle ontologie formali come le proprietà. Molti dei linguaggi per ontologie tra i più diffusi e in via di sviluppo al giorno d'oggi nascono come supporto a quello che sarà il

web semantico, ma aiutano sicuramente anche nel campo della programmazione ad agenti; RDF ad esempio è stato proposto da W3C per la codifica e lo scambio di metadati strutturati fra le applicazioni web, mentre OWL (estensione di RDF descritta in XML) per rappresentare esplicitamente significato e semantica dei termini con vocabolari e relazioni fra essi.

Se si è quindi definito un insieme di termini in comune per determinati settori di interesse, quale linguaggio e concetti ci mancano ancora per riuscire a far comunicare più agenti fra loro? Ci si accorge subito che ad esempio non è possibile riusare i concetti della comunicazione nell'object oriented programming; se l'oggetto 'a' per comunicare invoca un metodo sull'oggetto 'b', quest'ultimo non potrà far altro che eseguirlo senza il minimo controllo sul proprio comportamento. Ciò non è chiaramente possibile nella comunicazione fra agenti, dove se l'agente 'b' vede arrivare una richiesta (quindi nella fase di comunicazione, appunto) può declinarla o meno in base ai suoi interessi. La comunicazione quindi è da intendere come azione volta a influenzare le credenze degli altri agenti. La teoria che tratta la comunicazione come particolare tipo di azione è detta Speech Act Theory.

Nel tempo sono stati sviluppati alcuni linguaggi sulla base di questa teoria, il KSE (Knowledge Sharing Effort, fondato dalla DARPA) ha creato sostanzialmente KQML e KIF, dove il primo definisce il formato e la semantica di un messaggio, con alcuni parametri ognuno col proprio significato, lasciando libera la gestione del contenuto (il KIF inizialmente doveva occuparsi di quest'ultimo aspetto). Ogni messaggio in KQML (knowledge query and manipulation language) ha una performativa (ovvero una sorta di classe del messaggio) e un numero di parametri con determinati valori; segue un esempio di messaggio per una richiesta di prezzo su un oggetto generico IBM e una lista (parziale) delle performative possibili.

```
(ask-one
   :content  (PRICE IBM ?price)
   :receiver stock-server
   :language LProlog
   :ontology NYSE-TICKS
)
```

Figura 1.2: Esempio di messaggio KQML

Performative	Meaning
achieve	S wants R to do make something true of their environment
advertise	S is particulary suited to processing a performative
ask-about	S wants all relevant sentences in R s VKB
ask-all	S wants all of R s answers to a question
ask-if	S wants to know if the sentence is in R s VKB
ask-one	S wants one of R s answers to a question
break	S wants R to break an established pipe
broadcast	S wants R to send a performative over all connections
broker-all	S wants R to collect all responses to a performative
broker-one	S wants R to get help in responding to a performative
deny	the embedded performative does not apply to S anymore
delete	S wants R to remove a ground sentence from its VKB
delete-all	S wants R to remove all matching sentences from its VKB
delete-one	S wants R to remove om matching sentence from its VKB
discard	S will not want R s remaining responses to a previous performative
eos	end of a stream of responses to an earlier query
error	S considers R s ealier message to be mal formed

Figura 1.3: Alcune performatives del KQML

Nel 1995 la FIPA (Foundation for Intelligent Physical Agents) iniziò i propri lavori per sviluppare un suo standard per i sistemi ad agenti; ne conseguì la creazione di un linguaggio di comunicazione (detto anche FIPA ACL). Tale linguaggio ha una sintassi molto simile al KQML, la differenza principale sta nelle performative offerte, che nella FIPA ammontano a venti. Con il linguaggio FIPA poi si è cercato di ovviare alla principale critica al KQML, ovvero la mancanza di una semantica adeguata. L'approccio per tale scopo è stato produrre una semantica che rispettasse un linguaggio formale chiamato SL, che permette di rappresentare i belief, desires e uncertain beliefs degli agenti (rispettivamente credenze, desideri e credenze non certe) così come le azioni da performare. Inoltre le due principali performative nell' ACL FIPA sono inform e request, dato che tutte le altre sono in qualche modo definite in termini di queste due.

Parameter	Category of Parameters
performative	Type of communicative acts
sender	Participant in communication
receiver	Participant in communication
reply-to	Participant in communication
content	Content of message
language	Description of Content
encoding	Description of Content
ontology	Description of Content
protocol	Control of conversation
conversation-id	Control of conversation
reply-with	Control of conversation
in-reply-to	Control of conversation
reply-by	Control of conversation

Figura 1.4: Parametri di un messaggio FIPA

1.5.2 Coordinazione

Dopo la comunicazione, la coordinazione è un altro aspetto cruciale nei sistemi multi-agente. Il problema principale nel lavoro cooperativo è appunto la coordinazione espressa in termini di gestione delle dipendenze fra le attività di più agenti. Abbiamo bisogno essenzialmente di un minimo meccanismo per poter gestire tali dipendenze quando esse si presentano. Possiamo trattare in maniera generica alcuni tipi principali di coordinazione multi-agente, come segue.

Parlando di Partial Global Planning (sviluppato inizialmente da Durfee nel 1988) gli agenti cooperano scambiandosi informazioni per raggiungere conclusioni sul problem-solving. Il planning è parziale poichè il sistema non può generare una soluzione per l'intero problema, ed è allo stesso tempo detto globale poichè gli agenti formano una soluzione non locale scambiandosi informazioni su soluzioni locali. Ogni agente quindi in base ai propri obiettivi genera soluzioni parziali e locali, vengono scambiate informazioni per determinare come tali soluzioni possono interagire e in caso vengono modificate per coordinarsi. Una struttura dati detta PGP (partial global plan),

generata coperativamente, contiene informazioni sull'obiettivo globale del sistema.

Un altro approccio è basato sui modelli per il lavoro di squadra umano, e vanno qua distinti con chiarezza i concetti di azioni (semplicemente) coordinate e azioni coordinate e cooperative. Ad esempio, c' è differenza fra un gruppo di persone che simultaneamente usa il proprio ombrello quando arriva un acquazzone, ed un gruppo di ballerini che assieme seguono una coreografia. Vi è insomma, nel secondo caso, una sorta di responsabilità di ognuno nei confronti degli altri. Chiarito tale aspetto, l'approccio prevede nei dettagli dei 'commitment' e 'convention', dove i primi sono intenzioni, mentre i secondi specificano quando una commitment va abbandonata o meno e in caso come agire da soli e coperativamente per soddisfarla, tenendo sempre presente il fatto che si trattano azione cooperative.

Viene poi la coordinazione per mutua modellazione, legata strettamente all'approccio sopra citato. Se ad esempio, due persone hanno intenzione di oltrepassare una porta larga abbastanza per una persona alla volta, è necessario un meccanismo di coordinazione per gestire tale risorsa. Un'idea è mettersi ognuno nei panni dell'altro, ovvero costruire un modello con beliefs e intentions e coordinare le attività sulla base delle previsioni del modello stesso. Così facendo, si può prevedere che uno dei due voglia favorire l'altro e permettere quindi di oltrepassare la porta per primo, gestendo la risorsa in maniera adeguata.

Ultimo approccio qui presentato riguarda l'affidamento alle norme e alle cosiddette regole sociali. Una norma è un'aspettativa di comportamento, mentre una regola sociale differisce dalla prima soltanto per una presunta autorità nella stessa. Tali vincoli possono aiutare sensibilmente nella costruzione di modelli di coordinazione inter-agenti, tral'altro semplificando il carico di lavoro e il processo di decision-making per gli agenti i quali dovranno perlopiù confrontarsi con regole già presenti in qualche struttura.

1.6 Applicazioni

Analizziamo più nel dettaglio alcune applicazioni pratiche possibili per il paradigma ad agenti, tra stato dell'arte e futuri sviluppi. In generale, i principali domini di applicazione riguardano i sistemi distribuiti, dove gli agenti rappresentano nodi del sistema, e trattiamo qui chiaramente con sistemi

multi-agente, e i Personal Software Assistants, dove gli agenti individuali giocano il ruolo di assistenti proattivi per l'utente.

1.6.1 Sistemi distribuiti

- Agenti per il Workflow e Business Process Management: tali sistemi di workflow e simili hanno lo scopo di automatizzare il processo di business assicurandosi che task diversi siano affidati alle persone giuste al momento giusto. ADEPT è un sistema simile organizzato ad agenti modellato come una società di negoziazione e di servizi offerti ad agenti.

- Agenti per il Sensing distribuito: è un approccio classico ad agenti, ovvero utilizzarli per gestire una rete di sensori distribuiti nello spazio. Ad esempio, nella gestione di sensori ottici per il passaggio di un veicolo in un determinato settore, l'analisi può essere praticata al meglio se i vari sensori sono agenti capaci di cooperare fra loro, prevedendo magari l'arrivo di un veicolo da una zona appena analizzata ad un altra adiacente, e così via.

1.6.2 Personal Software Assistant

- **Agenti per l'Information Retrieval:** il web così com'è è chiaramente una fonte enorme di dati, in continuo aumento e per questo, per un utente in procinto di effettuare una ricerca più o meno accurata, si ha il costante rischio di non riuscire nell'intento vista l'eterogeneità e la qualità dei dati presenti. Un information agent è un agente che in questo caso può accedere ad una o più risorse, gestendole al meglio per riportare all'utente il risultato di una sua specifica richiesta. Ad esempio, un personal information agents sviluppato da Pattie Maes è MAXIMS, il quale impara a gestire (quindi cancellare, inviare, ordinare) mail al posto dell'utente. Esso gestisce gli eventi con un'azione conseguente ad essi, ed ogni evento ha particolari proprietà ed attributi che standardizzano il suo comportamento. Rimanendo in tema, trattiamo i Web agent, ovvero agenti come i Tour guides (i quali possono aiutare l'utente nella navigazione web sulla base delle sue preferenze, oppure gli Indexing agents) che forniscono un layer extra

di astrazione sopra servizi quali Google per fornirne di extra e ancor più personalizzati.

- Simili a questi ultimi abbiamo gli Agenti per l'E-Commerce, dove essi diventano veri e propri assistenti durante la compravendita online, come ad esempio i Comparison shopping agents, i quali possono catalogare e fornirci dettagliati elenchi di vari oggetti sulla base dei nostri criteri di ricerca- quindi in base al miglior prezzo, o magari sulla base di prezzo, livello di usura ed altro quando si tratta di particolari acquisti come veicoli usati.

- **Agenti per l'interazione Uomo-Macchina:** il classico approccio con cui noi utenti al giorno d'oggi interagiamo con le nostre macchine è detto direct manipulating, ovvero l'impartire ordini precisi (tramite interfaccia grafica, pulsanti, ecc.) e attendere la risposta passiva del computer. Quello che potrebbero apportare gli agenti a questa interazione è un aspetto proattivo, ovvero un'interfaccia che prenda l'iniziativa e ci guidi passo passo verso i nostri obiettivi, consigliandoci/indirizzandoci verso determinate azioni. Sono detti anche interface-agents.

- Infine, come già detto gli agenti possono aiutare nella simulazione sociale (progetto EOS già citato), oppure più semplicemente possono essere utilizzati in ambienti virtuali quali mondi simulati e videogiochi ad esempio, fornendo un'entità definita nientemeno come believable agents. Quest'ultimo concetto è tutto fuorchè banale, si parla di agenti con vere e proprie emozioni, di modo che l'utente quando vi interagirà avrà la sensazione di rapportarsi con un essere umano. Tale campo è sicuramente molto all'avanguardia e per questo, più di altri, è destinato a rimanere per ancora molto tempo soltanto in via di sviluppo.

1.7 Stato dell'arte

Sono numerose le tecnologie in fase di sviluppo e ricerca per la programmazione orientata agli agenti, ma non sarebbe interessante un elenco sostanzioso di nomi e sigle di progetti vari e non è oggetto di questa tesi. Come per ogni disciplina, l'ideale è focalizzarsi su un aspetto e diventarne per così dire

specializzati; in questa tesi, dopo un primo macro-capitolo di introduzione (doveroso) verso i sistemi multi-agente, verrà fatto un ampio focus su quello che è Jason, un progetto tra i più sostenuti e apprezzati dalla community di appassionati e ricercatori del settore. Inoltre verrà trattato per completezza (e mostrato come può interagire con lo stesso Jason) JADE, un framework scritto in Java (con tutti i benefici che ciò comporta) che offre strumenti utilissimi durante la progettazione e diagnostica di tali sistemi.

1.8 Conclusione

Come auspicato nell'introduzione, a questo punto della lettura (che mi auguro sia stata interessante e soprattutto leggera per chiunque) si dovrebbe avere una visione abbastanza chiara e a grandi linee di quello che sono i sistemi multi-agente.

Si è visto quando sono nati, perché già utilizzati in molti campi dell'informatica, pur rimanendo sostanzialmente ancora un argomento d'elite della comunità di ricerca di tale settore.

Concentrandoci ora su una specifica tecnologia (almeno nella seconda parte del secondo capitolo e in quello finale) si cercherà nel limite del possibile di continuare a trattare gli argomenti in maniera fruibile a chiunque, pur non potendo esplicare ogni singolo aspetto tecnico presente, in particolar modo tale premessa ha senso in vista della parte finale dove son presenti modelli UML e codice.

Capitolo 2

Jason come linguaggio di programmazione ad agenti

2.1 Primo approccio a Jason

2.1.1 Un linguaggio di programmazione ad agenti

Avendo visto nella prima parte di questo documento gli aspetti principali e le peculiarità dei sistemi multi-agente, potremo già avere un'idea di quelle che dovrebbero essere le caratteristiche di un linguaggio di programmazione cosidetto 'ad agenti'. innanzitutto, tale linguaggio dovrebbe poter gestire la nozione di obiettivi; non vogliamo comunicare con gli agenti fornendo loro istruzioni dettagliate su come agire, bensì vorremo poter ragionare in termini di obiettivi, passando loro tali informazioni e lasciandoli liberi di scegliere il corso d'azione migliore. Il linguaggio inoltre dovrebbe permettere la produzione di sistemi che rispondono all'ambiente cui sono immersi, così come il fornire un comportamento 'responsive' per gli agenti stessi. Per ultimo, ma non meno importante di certo, va gestita la capacità di interazione e coordinazione fra gli agenti di modo da fare enfasi sulle abilità sociali, loro caratteristica cardine per definizione. Jason si propone come un linguaggio che cerca (e di certo riesce egregiamente) ad assolvere tutti questi aspetti quasi nella loro totalità. Jason, creato tra gli altri da R.H.Bordini, è un interprete per il linguaggio di programmazione (o meglio, una sua versione modificata) AgentSpeak, il quale si basa sull'architettura per il comportamento degli agenti detta BDI (beliefs-desires-intentions).

Figura 2.1: Icona di Jason

Jason è scritto in Java, rendendolo multi-piattaforma ed è offerto sotto licenza GNU LGPL. E' possibile utilizzarlo con framework e ambienti di programmazione come JADE, con il quale si integra per offrire supporti per sistemi multi-agente e la loro diagnostica. Viene fornito principalmente come plugin per J-Edit ma è disponibile anche per Eclipse. Andiamo ora ad analizzare un 'Hello World' per avere un primo impatto col linguaggio, per poi approfondirne uno alla volta gli aspetti fondamentali.

2.1.2 Hello World in AgentSpeak

Un semplice programma di stampa della stringa 'Hello World', può essere così presentato in AgentSpeak (ricordiamo che Jason è un interprete di questo linguaggio, che è (assieme all'architettura BDI) fondamentale da capire per poter programmare in Jason), come mostrato nell'immagine.

innanzitutto, ciò potrebbe definire il comportamento di un singolo agente (sebbene potrebbero esserci più linee di codice, fino a formare una lista di statement) ed andrebbe quindi salvato in un singolo file con estensione .asl. La definizione di un singolo agente viene suddivisa negli initial beliefs (e initial goals se vi sono) e nei plans dell'agente; qui 'started' (seguito dal separatore, ovvero il punto) rappresenta un belief, ovvero una credenza che ha l'agente, che può riguardare sé stesso o l'ambiente. Tale belief deve iniziare necessariamente con una lettere minuscola. Nella linea successiva abbiamo un piano (plan) dell'agente; può essere letto come 'ogni volta che hai un belief started, esegui l'azione specificata' che in questo caso è la stampa della stringa 'Hello World'. Il simbolo + prima di started sta ad indicare l'acquisizione di tale belief da parte dell'agente.

Il piano è così 'triggerato', poiché attivato dal fatto che l'agente ha disponibile il belief started (perché noi stessi glielo abbiamo dato come belief

```
started.

+started <- .print("Hello World!").
```

Figura 2.2: AgentSpeak's Hello World

iniziale). In ogni caso, prima di eseguire il piano, l'agente controlla che il contesto del piano sia appropriato: difatti possiamo definire azioni complesse nella gestione del contesto, di modo che un agente possa avere diverse risposte allo stesso evento, e possa scegliere l'azione migliore a seconda della situazione cui si trova. In questo caso chiaramente il piano viene eseguito sempre poiché il contesto è vuoto e viene quindi considerato sempre applicabile in esso. L'azione 'print' è preceduta dal punto ed è un'azione interna, ovvero che non modifica l'ambiente (lo modificano invece le normali azioni). Online è facilmente reperibile nella documentazione di Jason l'elenco di azioni interne standard. Prima di entrare nel vivo con i dettagli riguardo a Jason come linguaggio di programmazione (quindi costrutti, sintassi e quant'altro) viene ora presentato il modello ad agenti BDI.

2.2 Architettura BDI

2.2.1 Beliefs, Desires, Intentions

AgentSpeak si basa su un architettura particolare, denominata BDI, ovvero 'beliefs, desires, intentions'. E' stata pensata per svariati motivi, ma principalmente aiuta a progettare sistemi multi-agente e quindi a vederli come un insieme di entità che ci simulano, e come noi hanno uno stato mentale. Possiamo così programmarli andando a modificare quello che è il loro stato mentale, e non fornendo loro un elenco di istruzioni specifiche sui compiti loro assegnati. La realtà è più complessa di quello che sembrerebbe, dato che una prima visione prevede che il progettista non conosca nulla di come gli agenti andranno poi a conseguire i propri obiettivi, limitandosi soltanto a gestire i loro aspetti organizzativi e intenzionali. Come si evince da una prima analisi, ciò aiuta a vedere appunto gli agenti come esseri umani. Entrando nel dettaglio, i beliefs rappresentano quelle che sono le credenze

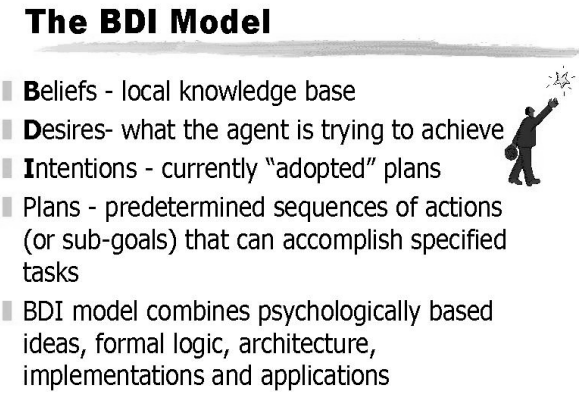

Figura 2.3: Architettura BDI

dell'agente sul sistema, le quali possono essere, così come lo sono le nostre, sulla realtà che ci circonda, incomplete ed imprecise. Sono quindi una rappresentazione di quello che un agente 'crede' dell'ambiente cui è immerso. I desires sono obiettivi, stati generici, o una qualunque situazione che l'agente vorrebbe eseguire. Ciò influenza di molto le prossime mosse dell'agente, in maniera diretta, offrendogli effettivamente opzioni selezionabili per le future azioni pratiche. Chiaramente più desires possono essere in conflitto fra loro, così come negli esseri umani certi obiettivi sono irrealizzabili assieme e/o si escludono a vicenda. Infine le intentions sono desires che l'agente ha deciso di prendere in carico e quindi di portare a termine in qualche modo. Tali opzioni diventano quindi automaticamente intentions.

2.2.2 Practical Reasoning

Ciò ci offre quindi una maniera di progettare tali sistemi complessi astraendo da un certo livello di implementazione, di modo che sia possibile più facilmente (anche per non esperti di aspetti algoritmici) progettarli dato l'alto livello che ci offre un sistema BDI. La visione è quindi quella di programmare i nostri agenti gestendo i loro stati mentali, lasciandoli più o meno liberi di scegliere le azioni effettive migliori per arrivare ai loro obiettivi, e quindi basandoci su un particolare meccanismo di ragionamento interno

degli stessi. Tale meccanismo non è nient'altro che il Practical Reasoning, già accennato nel primo capitolo di questa tesi nell'approfondimento sui diversi tipi di agenti in circolazione. Come già accennato, si divide in una fase di deliberation e means-end reasoning. Con la deliberation sostanzialmente gli agenti scelgono le intentions, spesso nella maniera ovvia in cui le vediamo come pro-attitude per certe azioni. Ovvero, se l'agente (o come astrazione, un qualunque ragazzo) ha intenzione di giocare a calcio, ciò lo porterà all'azione effettiva di giocare; non è un desiderio, ma un'intenzione, che comporta il fatto che un eventuale imprevisto lo porti a riprovare a portare a termine l'obiettivo, magari in un'altro spazio o tempo successivo. Ciò tral'altro comporta che adottare un'intenzione modifica inevitabilmente il nostro comportamento futuro, finché tale obiettivo non è stato raggiunto oppure siamo stati impossibilitati nell'opera. Il means-end reasoning non è altro che il processo con cui usando i means (ovvero le azioni possibili) decidiamo di conseguire un obiettivo, un'opzione tra i vari desires quindi. Non si tratta altro di planning, ovvero un algoritmo che in base al nostro goal (obiettivo), ai beliefs dell'agente sull'ambiente e alle azioni possibili, produce un plan, un corso d'azione che andrà eseguito nell'immediato futuro. In AgentSpeak introduciamo ora un interessante modello per questo tipo di meccanismo di reasoning.

2.2.3 Practical Reasoning in BDI

L'approccio adottato in AgentSpeak è quello di lasciare allo sviluppatore il fornire all'agente, in design-time (prima dell'esecuzione), delle collezioni di plans parziali, lasciando poi all'agente il compito a run-time di unire più plan sulla base di beliefs, per raggiungere i propri goals. E' un approccio diverso dal fornire piani d'azione parziali, ma si è rivelato nella pratica e nelle performance decisamente potente e migliore di altri, ed è quindi stato adottato in AgentSpeak, riflettendosi in Jason, nostro vero campo d'interesse in questa tesi. Il Procedural Reasoning System (PRS) è un approccio simile a quello appena esposto, ma prevede un loop in cui durante l'esecuzione di plan o il conseguire un intention, si possa controllare che le condizioni per cui tale esecuzione (pendente) sia ancora la miglior 'cosa da fare'. Ovvero, si è capaci di fermarsi o cambiare corso d'azione e/o obiettivo anche durante l'esecuzione dello stesso. Se ad esempio durante l'esecuzione di un intention acquisiamo un belief che mette in discussione i belief (e l'intention) prece-

denti, possiamo interrompere il tutto; oppure se acquisiamo un goal che va in contrasto con il plan in esecuzione, oppure non sussistono più le condizioni per eseguirlo, ecc. Ciò rende il flusso incredibilmente flessibile e più simile al comportamento di un essere umano, dato che noi stessi tutti i giorni abbandoniamo corsi d'azione o idee per i più svariati motivi, per cedere il nostro interesse ad altro. AgentSpeak è quindi un linguaggio che cerca in qualche modo di fornire tali aspetti del PRS non previsti inizialmente nel Reasoning; quando verrà analizzato il ciclo d'esecuzione dell'interprete Jason si avrà chiaro il loop adottato dal nostro linguaggio.

2.2.4 Comunicazione: KQML e FIPA

Il modello BDI prevede la gestione dello stato mentale di un singolo agente, tra ciò che crede, ciò che vorrebbe fare, e ciò che sta effettivamente per attuare. Non prevede quindi meccanismi né teorie per la comunicazione e coordinazione inter-agente. La teoria adottata come visione in AgentSpeak è quella già citata nel primo capitolo, la Speech Act Theory, ovvero una visione in cui la comunicazione è vista come un particolare tipo di azione. I vari messaggi contengono una performative, tra le altre proprietà, che identifica il messaggio sulla base del suo 'significato'. Adottando quindi linguaggi come KQML e FIPA (già discussi in precedenza), abbiamo quindi messaggi che viaggiano in rete completi di tutte le informazioni, tra cui appunto una performative; un messaggio del tipo 'tell' modificherà i beliefs dell'agente ricevente, un messaggio con performative 'achieve' ne modificherà invece i goals. Non verrà di nuovo approfondita tale questione che verrà invece affrontata nelle prossime sottosezioni. Entriamo ora nel dettaglio e nello specifico di Jason, dopo aver esordito in maniera esauriente con tutto ciò che fa da compendio per comprendere a fondo tale linguaggio.

2.3 Uno sguardo dettagliato

Precisiamo innanzitutto la differenza fra agent architecture e agent program; un agent architecture è un framework con le sue regole e vincoli ben definiti, dove va in esecuzione un agent program. Il PRS è un esempio di agent architecture, e abbiam visto come l'architettura BDI si rifà a quest'ultimo, e Jason interpreta un linguaggio AgentSpeak esteso che attinge dalla BDI. Gli agenti BDI sono reactive planning system, ovvero hanno un ciclo continuo

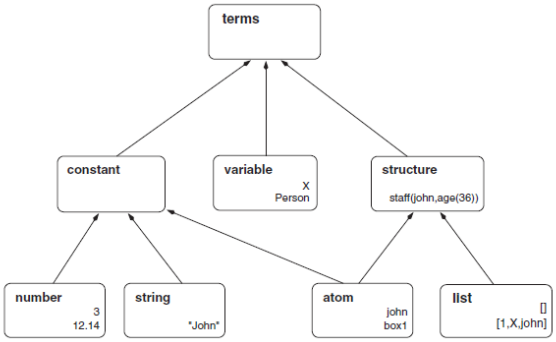

Figura 2.4: Term presenti in Jason

(quindi non sono programmi che terminano dopo l'esecuzione di un task o funzione) in cui percepiscono informazione dall'ambiente, fanno delle scelte ed eseguono plans (quindi azioni) per modificare l'ambiente, di conseguenza. Analizziamo ora nel dettaglio gli aspetti base di Jason, iniziando da quelli che sono i beliefs.

2.3.1 Beliefs

La beliefs base non è altro che un elenco iniziale di belief che il designer affida all'agente prima dell'effettiva esecuzione; chiaramente tale elenco può espandersi o modificarsi in ogni modo durante il ciclo di vita dell'agente. Ci troviamo, con AgentSpeak, a trattare di programmazione logica, e in questa le informazioni sono rappresentate in maniera simbolica dai predicati. Ad esempio, 'tall(jhon)' è un predicato che ci informa di una proprietà di jhon, che qui è un 'term' che probabilmente definisce un'entità denominata Jhon. Un sequenza del tipo 'likes(jhon,music)' rappresenta un 'literal', ovvero un predicato o la sua negazione, e ci informa chiaramente che a Jhon piace la musica. E' da sottolineare che se un agente ha nella sua beliefs base l'ultima sentenza, ad esempio, ciò non significa che essa sia una verità assoluta, tutt'altro: significa semplicemente che tale agente pensi che ciò sia vero, ma potrebbe non esserlo. L'immagine seguente illustra la terminologia e i tipi di 'term' in Jason, fondamentale per proseguire nella lettura e nella comprensione del codice.

E' ora d'obbligo una breve divagazione sulla programmazione logica e su un pò di terminologia utilizzata da qui in poi, per coloro che non sono avvezzi con linguaggi per la programmazione logica come il Prolog.

2.3.2 Cenni di programmazione logica

Come nel Prolog, ogni sequenza di caratteri cui inizia con lettera minuscola è chiamata atom; ad esempio, nel caso precedente, jhon era un atom. Se invece la sequenza di caratteri inizia con lettera maiuscola si tratta di una variabile, che può essere free o non istanziata; una volta istanziata (o anche bound) ad un particolare valore, lo mantiene fino alla durata del suo scope, che in AgentSpeak sono rappresentati dai plans. Ad esempio, la variabile Person può assumere il valore jhon. Inoltre, atomi, numeri e stringhe sono visti e classificati anche come costanti. Una structure è un tipo di dato complesso, come ad esempio 'staff('Paolo Rossi', 123234, sposato)', quindi un elenco di atomi detti argomenti, separati da virgole e racchiusi da parentesi; l'atom all'esterno (ovvero staff) è detto functor. Notare inoltre che i predicati (facts in Prolog) hanno la stessa sintassi delle structure, difatti differiscono soltanto per la semantica. Il numero di argomenti è importante ed è definito come arity della struttura. Una struttura appunto è definita dal functor e dalla sua arity. Se una struttura è racchiusa tra parentesi quadre è invece chiamata lista, e vi si possono usare particolari costrutti come '—', il quale separa il primo elemento dagli altri, e può essere usato per unire diverse liste creando matching particolari e precisi tra gli elementi. Per comprendere meglio il tutto, una rilettura di questo paragrafo è consigliata assieme all'immagine sopra che definisce i terms e le relazioni fra di essi.

2.3.3 Annotations

Una differenza importante fra Jason e i linguaggi di programmazione logica classici è l'uso delle annotations, ovvero particolari strutture associate ad un belief, che conferiscono informazioni aggiuntive, spesso fondamentali. 'busy(jhon)[expires(autumn)]' ci dice che l'atom jhon (associato presumibilmente ad un'entità Jhon) è occupato finché è autunno. Ciò ci conferisce due vantaggi: innanzitutto l'eleganza della forma, da non sottovalutare per molti sviluppatori, ed inoltre aiuta di molto la gestione della base beliefs, dato che può aiutare a scegliere quanto tempo tale belief deve permanere

nella base. Di per sé, così com'é presentata sopra, l'interprete Jason non ha chiaramente la nozione di autunno e non sa quando rimuovere il belief dalla base, ma con delle personalizzazioni possibili con poco codice Java ciò è possibile ed ammissibile dal framework che circonda Jason. Ci sono poi annotations che per l'interprete hanno invece significati specifici, come 'source', che è tra l'altro l'annotations inizialmente prevista (l'unica) dal sistema. Con source si ha chiaramente informazioni sulla fonte che ha causato la presenza di tale belief nella beliefs base dell'agente. L'annotations source è di tre tipi: percept, ovvero il belief è stato acquisito a seguito di una percezioni tramite sensori (virtuali o meno) dell'ambiente, communication, ovvero l'informazione ci è stata comunicata da un altro agente, ed infine mental notes, ovvero informazioni che l'agente decide di memorizzare e che possono tornare utili in futuro per determinati obiettivi. Con source(percept) denotiamo la percezione, con source(self) l'utilizzo delle nostre mental notes, mentre con source(NomeAgente) il fatto che il belief ci è stato comunicato da un agente. Rimanendo in tema, è possibile l'uso di nested annotations, ovvero annotations annidate. Ad esempio, se nei belief dell'agente bob abbiamo 'loves(maria,bob)[source(jhon) [source(maria)]]' vediamo chiaramente una annotations annidata; questo può capitare se ad esempio jhon manda un messaggio a bob con significato 'maria mi ha detto che lei ama bob', o meglio bob riceve un messaggio in cui jhon dice di aver saputo da maria che lei ama bob stesso. Può sembrare cervellotico ma rileggendo con calma la questione, tutto risulterà chiaro.

2.3.4 Negation

La negazione è un aspetto che causa spesso situazioni erronee nella programmazione logica. Un primo tipo di negazione è chiamata Closed world assumption (o anche negation as failure) e può essere così espressa:

Ogni cosa che non è conosciuta come vera, né derivabile da fatti noti usando le regole del programma, è da assumere come falsa.

Quindi l'unica negazione possibile, definita con l'operatore not, vi è quando è vera la negazione di una formula che l'interprete ha fallito nel derivare, stando ovviamente alle regole del programma (a proposito di regole e quindi Rules, esse sono un aspetto avanzato della programmazione

logica, presente anche in Jason, ma non trattato qui poiché in molti ambiti, come nelle nostre casistiche, è tranquillamente evitabile). Un altro tipo di negazione rappresentato dall'operatore tilde è detto Strong Negation, ed esprime una situazione in cui l'agente crede esplicitamente che qualcosa sia falso. Ovvero, se l'agente crede che (operatore tilde)colour(car,red), significa che crede fermamente che il colore della macchina non sia rosso. Sotto le assunzioni del closed world sopracitato, tutto ciò che non è esplicitamente creduto vero è falso; ciò è troppo restrittivo, specialmente quando si tratta di sistemi aperti; in generale è bene avere la possibilità di modellare il fatto che un agente creda che un qualcosa sia vero, falso, oppure che non possa con certezza dire nessuna delle due opzioni. Se quindi se sono vere not p e not (tilde)p, significa che l'agente non può sapere se p è vero o falso. Inoltre, se se si usa la closed world assumption, è da ricordare di non usare strong negation e di fare in modo che i beliefs percepiti dall'ambiente contengano tutti literal positivi. Infine, ecco un breve esempio per ricapitolare gli ultimi argomenti:

colour(box1,blue)[source(bob)] (tilde)colour(box1,white)[source(jhon)] colour(box1,red)[source(percept)]

Siamo quindi prima informati da bob che esso crede che il box1 sia di colore blu, mentre poi siamo informati da jhon che egli creda fermamente che il box1 non sia bianco. Percepiamo poi noi stessi che il box1 è rosso; ciò ci mette in condizione di credere che bob sia un bugiardo, oppure non sappia riconoscere i colori, dato che in genere si segue la convenzione nel dare priorità alle percezioni piuttosto che alla comunicazione (una sorta di enfasi del fidarsi è bene, non fidarsi è meglio). Potremo poi continuare producendo belief con fonte self (noi stessi) in cui etichettiamo bob come bugiardo o come daltonico assegnando magari diverse probabilità ai due fatti. Chiaramente quest'ultima aggiunta non è gestibile automaticamente dall'interprete così come non lo era prima la nozione di tempo basata sulla stagione autunno.

2.3.5 Goals

Il concetto di Goals è fondamentale; se i belief rappresentano ciò che l'agente crede del sistema cui è immerso, i goals rappresentano lo stato dell'ambiente

che l'agente vorrebbe. L'agente con il goal g agirà per far sì che l'ambiente si porti nella condizione definita, appunto, da g. E' un modo particolare di vedere i goals definiti come declarative goals. In AgentSpeak abbiamo la suddivisione dei suddetti in achievement goals e test goals. I primi sono denotati dal simbolo ! e rappresentano, come suggerisce il termine, uno stato cui si vuole giungere. L'esempio !own(car) ci suggerisce che l'agente dovrà fare in modo di arrivare ad avere tale automobile. I test goals, rappresentati dal simbolo ?, ci aiutano a capire cosa un agente crede di un literal o un gruppo di literal, e quindi avere informazioni sulla sua base belief. Ad esempio, un test goals del tipo '?bankbalance(B)' fa in modo di memorizzare in B il saldo bancario dell'agente, posto chiaramente che tra i suoi belief ci sia tale informazione. Entrando ancor più nei dettagli, gli achievement goals si possono usare in maniera procedurale o dichiarativa: la maniera procedurale li rende simili a metodi e/o procedure come nei normali linguaggi di programmazione, mentre la maniera dichiarativa, come già ribadito, esprime uno stato relativo all'ambiente (quindi appunto, lo dichiara) cui si vuole arrivare. Si possono inoltre prevedere goals iniziali (initial goals) che vengono inseriti nell'agente non a run-time, un pò come i belief iniziali presenti nella belief base; ad esempio, un agente con l'initial goal !goout(house) cercherà in ogni modo, a lungo termine, di uscire di casa.

2.3.6 Plans

I plans sono l'ultimo concetto veramente fondamentale assieme a belief e goal presente in Jason, almeno in una prima fase di iniziazione al linguaggio. Un plan in AgenSpeak è diviso in tre parti ben definite separate da una certa sintassi, ovvero:

triggeringevent : context <- body.

Il triggering event è appunto un evento relativo all'ambiente, e un insieme di triggering event specifica in un plan quando esso dev'essere eseguito (ovvero quando matcha con uno degli eventi). Ciò delinea quello che è l'aspetto proattivo e reattivo dell'agente, poiché esso gestisce l'esser immerso nell'ambiente tramite percezione di eventi. Gli eventi possono essere di tipo addition o deletion, quindi aggiungere o rimuovere belief o goal da quelli già presenti nell'agente. Se quindi il plan matcha l'evento, si dice che è

rilevante per esso, e se sussistono certe condizioni (il contesto) esso verrà eseguito. Continuando un primo briefing sugli aspetti base dei plans, il contesto rappresenta delle particolari condizioni per cui un plan può esser eseguito o meno. La visione dietro a tutto ciò è quella di far sì che l'agente sia il più reattivo possibile, cercando di tardare al massimo l'esecuzione effettiva di un corso d'azione per assicurarci che quest'ultimo sia la miglior scelta al momento, evitando che un cambiamento repentino nell'ambiente possa cambiare le nostre performance. Difatti la prassi è fornire all'agente più plans per lo stesso goal, di modo che esso possa scegliere a dovere il miglior corso d'azione al momento. Un plan con un contesto da scegliere si ha dopo una cosidetta conseguenza logica, ovvero se esso in uno statement ritorna true; se così è, esso viene denominato applicabile e diventa diretto candidato per l'esecuzione. Il body infine è la parte più semplice del plan, poiché contiene formule e corsi d'azione da eseguire direttamente. Da notare però che potrebbe contenere allo stesso tempo anche goals, quindi obiettivi da aggiungere alla lista, e per questo denominati subgoals. Entriamo ora ancor più nel dettaglio di questi tre elementi per comprenderli a dovere

2.3.7 Triggering Events

Esistono esattamente sei tipologie di triggering events, generiche, previste da Jason: possiamo aggiungere e rimuovere beliefs (+l e -l, dove l è un belief), o aggiungere e rimuovere achievement goal e/o test goal (+!l per l'achievement, +?l per aggiungere un test goal). Aggiungere o rimuovere belief spesso capita quando l'agente percepisce qualcosa dall'ambiente, l'aggiunta di goal quando si esegue un plan (con un subgoal magari) oppure quando lo si comunica ad un altro agente. Se ad esempio abbiamo un agente che deve preparare una cena, avremo un goal del tipo prepared(dinner), e magari un plan lanciato da un triggering event come +!prepared(dinner) (ovvero un plan che farà di tutto per far sì che l'agente creda di aver preparato la cena, come da obiettivo). Potremo anche utilizzare la variabile Meal generica ed instanziarla nel nostro caso con l'atom dinner, rendendo più indipendente la soluzione. Questo era un achievement goal. Riguardo i test goals, essi possono essere usati come triggering event quando abbiamo bisogno di recuperare informazioni dalla belief base (magari nell'esempio controllare che ci sia abbastanza pasta per X persone), e in caso non sia direttamente disponibile tale informazione, procedere con un'azione aggiuntiva. Difatti

l'interprete quando trova un test goal, agisce prima cercando l'informazione nella beliefs base e se fallisce lancia un evento che cerca di fare in modo che sia eseguito il plan per il test goal, se possibile.

2.3.8 Context

Ricordandoci che il contesto decide se un plan è applicabile o meno, e che il concetto base è quello di tardare il più possibile l'esecuzione del plan, entriamo nel dettaglio nella costruzione di un contesto. Esso è quasi sempre definito dalla congiunzione di literal (ovvero statement sull'ambiente) di un certo tipo. Utilizziamo innanzitutto il prefisso not per definire semplicemente che tale literal non è nella beliefs base dell'agente (il literal può prevedere strong negation oppure no). Non credere che ad esempio l sia falso non è lo stesso di credere che sia vero, per intenderci. Schematicamente: la sintassi 'l' ci dice che l'agente crede che l sia vero, tilde l che sia falso, not l che l'agente non crede che l sia vero ed infine not tilde l, che l'agente non crede che l sia falso. Sono tipi di literal diversi fra loro la cui sottile differenza è da cogliere a pieno. Un importante appunto è da fare riguardo il fatto che il contesto di un plan è subito controllato con la beliefs base, dove avviene un matching automatico se ad esempio nella belief base si ha stock(pasta,5), nel contesto stock(Something,S) e la variabile Something è già instanziata a pasta. Ciò viene checkato senza altre operazioni su variabili. Esistono anche le azione interne, che non vengono checkate con la beliefs base, e contengono codice Java per eseguire determinati task; da notare però che sono eseguite per controllare se un piano è applicabile, ciò significa che vengono eseguite quando viene scelto il miglior piano, ovvero anche se il plan non risulta essere la miglior scelta. Sono quindi da evitare agenti carichi di internal actions poiché diventerebbero troppo carichi e poco performanti. Una maniera elegante di scrivere un plan con il suo context potrebbe essere:

canafford(Something): price(Something,P) and cash(B) and B>P

nella belief base per controllare se in generale l'agente può permettersi un certo acquisto, e prevedere un plan del tipo:

+!buy(Something):not tilde legal(Something) and canafford(Something) <- (...)

il quale verrà lanciato se vi è un evento (achievement goal) per acquistare qualcosa, ed effettivamente eseguito se l'agente non crede che l'oggetto sia illegale e crede che (sulla base del precedente belief) possa permetterselo.

2.3.9 Body

Il body contiene un corso d'azione che verrà eseguito se si è presentato un triggering event del plan e se il rispettivo context ha reso il plan applicabile; esistono sei tipi di formule che è possibile inserire in un body, separate da punto e virgola, e vengono qui presentate.

- Actions: Un agente dovrà chiaramente prima o poi, dopo il ciclo di reasoning, performare azioni, ed ecco che esse vengono presentate all'interno del body, sottoforma di predicati ground; essi si distinguono dalle condizioni del plan semplicemente per il posizionamento all'interno del body, dato che la sintassi è esattamente la stessa. Per avere un feedback sull'effettiva esecuzione, il plan che ha richiesto l'azione è sospeso finché essa non è eseguita (o meglio, esso non è a conoscenza del fatto che sia stata eseguita). Ciò ci dice soltanto se l'azione è stata eseguita o meno, in caso negativo il plan fallisce; chiaramente ciò non ci dice nulla sulle conseguenze che ha avuto la nostra azione, tali informazioni possono essere recuperate soltanto con delle percezioni sull'ambiente.

- Goals: Possiamo avere anche achievement goal o test goal. Riguardo gli achievement, avere un goal di questo tipo significa avere un evento che porta all'esecuzione di un plan; se un plan ha un obiettivo, e di conseguenza viene lanciato un altro plan per gestirlo, il corso d'azione ricomincia se quest'ultimo ha successo, altrimenti viene sospeso. Ad esempio, se si ha 'a1; !g2; a3' in un body plan, l'agente eseguirà a1, poi dovrà arrivare ad avere g2 e lancierà altri plan per questo; riprenderà con l'esecuzione e con a3 una volta gestito g2, dato che il plan principale è appunto stato sospeso. Possiamo avere anche formule precedute da !!, un operatore che denota il fatto che abbiamo un obiettivo da raggiungere ma non vogliamo aspettare la sua esecuzione per continuare il nostro corso d'azione. E' importante in genere, ritardare la scelta del corso d'azione ed anche l'assegnamento di valori (atom e non solo)

alle variabili, per gestire quella visione reattiva già citata. I test goals invece non vengono solo usati per ricavare informazioni dalla beliefs base (e non solo), ma sono utili anche perché se si cerca di eseguire un plan con un testo goal all'interno e quest'ultimo fallisce, l'intero plan fallisce ed è gestito dall'interprete. Se un test plan è inserito come triggering event, e non v' è tale informazione nella beliefs base, l'interprete prima di fallire crea un evento che cerca di recuperare i dati utili (utilizzando plan già creati dal progettista nell'ottica di tale situazione). Il plan è fallito soltanto se anche quest'ultimo tentativo non va a buon fine.

- Mental Notes: innanzitutto è da evitare in Jason l'aggiungere beliefs alla base dopo l'esecuzione di un azione; tali beliefs in genere sono gestiti in automatico tramite percezione dell'agente sull'ambiente. E' comunque utile creare beliefs, tramite fonte noi stessi (source[self]) per ricordarsi ad esempio di determinate informazioni passate. Viene usata una sintassi del tipo -+ per rimuovere e allo stesso tempo inserire un'istanza di una nota mentale, poiché bisogna esplicitamente aggiornare la nostra base belief, in genere per avere in mente l'ultima copia aggiornata (o visione aggiornata) della realtà.

- Internal Actions: Le azioni interne sono particolari tipi di azioni che non modificano l'ambiente, e sono precedute dall'operatore '.' ; si possono utilizzare metodi Java di package esterni, come ad esempio da una libreria per gestire le azioni possibili per robot o droni. Oppure utilizzare le internal actions (caldamente consigliato imparare almeno le più usate e conosciute) fornite dal framework Jason, facilmente reperibili online e nella documentazione Jason. Le internal actions esterne sono recuperabili con la sintassi 'packageesterno.metodo(parametri)' mentre quelle interne semplicemente con '.metodo(parametri)'. Un esempio di uso di un'internal action (prevista tra l'altro tra quelle standard di Jason) è questo:

  +!leave(home) : not raining and not tilde raining <- .send(mum,AskIf,raining); (...)

Ovvero se si ha un plan con triggering event di lasciare casa, si controlla prima nel contesto una condizione che semplicemente ci dice che l'agente non sa se piove e non sa nemmeno se non piove; chiederà quindi all'agente mum se sta piovendo, con l'azione interna send (richiamata appunto con l'operatore dot).

- Expressions e Plan Labels: Due ulteriori argomenti riguardano le Expressions e le label dei Plans. Nei plan context usiamo spesso espressioni che ritornano un booleano, per decidere in maniera logica se tale plan è applicabile o meno. Potremmo inserire espressioni del tipo X >= Y*2, ed eseguire il plan soltanto se essa è valida. Possiamo inoltre come in Prolog, legare dei predicati alle variabili piuttosto che legarvi terms; possiamo così memorizzare una formula come term nella beliefs base e usare una variabile per recuperarla ed eseguirla. O usare una variabile per matchare ogni evento, una sorta di 'agire se true <- true'. Attenzione con tali metodi, possono portare a comportamenti indesiderati nelle situazioni particolari, spesso nemmeno previste dal progettista. Un'occhiata alla sintassi utilizzata per le espressioni si ha nell'immagine seguente, e un approfondimento è disponibile online nella documentazione Jason, seppur non si tratti di nulla di trascendentale né di diverso per coloro già abituati alla programmazione in generale (ad esempio, == e $\neq$ confrontano se due elementi sono esattamente uguali o diversi rispettivamente, sapendo che la seconda è valida soltanto se sono del tutto diversi).

  I plan labels sono semplicemente dei riferimenti, delle nomenclature che vengono assegnate ai plans. In automatico Jason ne assegna uno per plan, se noi non lo facciamo in maniera esplicita. La sintassi è la seguente, ed è utile non soltanto per ritrovare e identificare i plan ma anche per inserire nelle label delle meta-informazioni sul plan.

  @label tevent: context <- body;

  E' possibile fornire predicati e informazioni aggiuntive nelle label, più o meno complesse, ma non è oggetto della tesi entrare troppo nel merito di argomenti così dettagliati; basti sapere che Jason prevede annotations standard riconosciute dall'interprete con determinati significati, che possono tornare certamente utili.

```
agent          → init_bels init_goals plans
init_bels      → beliefs rules
beliefs        → ( literal "." )*
rules          → ( literal ":-" log_expr "." )*
init_goals     → ( "!" literal "." )*
plans          → ( plan )*
plan           → [ "@" atomic_formula ] triggering_event
                  [ ":" context ]
                  [ "<-" body ] "."
triggering_event → ( "+" | "-" ) [ "!" | "?" ] literal
literal        → [ "~" ] atomic_formula
context        → log_expr | "true"
log_expr       → simple_log_expr
               | "not" log_expr
               | log_expr "&" log_expr
               | log_expr "|" log_expr
               | "(" log_expr ")"
simple_log_expr → ( literal | rel_expr | <VAR> )
body           → body_formula ( ";" body_formula )*
               | "true"
body_formula   → ( "!" | "!!" | "?" | "+" | "-" | "-+" ) literal
               | atomic_formula
               | <VAR>
               | rel_expr
atomic_formula → ( <ATOM> | <VAR> )
                  [ "(" list_of_terms ")" ]
                  [ "[" list_of_terms "]" ]
list_of_terms  → term ( "," term )*
term           → literal
               | list
               | arithm_expr
               | <VAR>
               | <STRING>
```

Figura 2.5: La sintassi di Jason

2.3.10 Esempio: Collecting Garbage

Quello presentato ora è un esempio disponibile in un documento presente nel sito ufficiale di Jason, una sorta di guida/tutorial che introduce brevemente al linguaggio. Tale esempio mostra tutto ciò che abbiam visto finora, e sebbene viene qui presentato prima del ciclo di reasoning dell'interprete, aiuta certamente a fare mente locale sui costrutti appena illustrati. Difatti, è opinione del sottoscritto che non v' è miglior metodo dell'imparare un linguaggio del vederlo in opera (e metterlo noi stessi in pratica chiaramente), anche se in questo caso è stata dovuta un'introduzione esaustiva data la natura logica del paradigma di programmazione, ben diverso da ciò che spesso è abituato un normale studente di informatica o appassionato casual. Viene quindi mostrato il codice AgentSpeak (eseguibile in Jason assieme ad altri file a sostegno, come quello che descrive l'ambiente) che definisci due robot che collezionano spazzatura in un ambiente cui sono immersi.

Il robot numero 1 dovrà muoversi nella griglia cercando spazzatura, e quando la trova, portarla al robot numero 2, il quale rimane fisso in una posizione pronto a bruciarla. Andando per ordine, nel codice notiamo innanzitutto che a design time definiamo una rule nella beliefs base, in cui controlliamo che il parametro P corrisponda alla nostra posizione, e in tal caso la sentenza è valida. Nei goals iniziali abbiamo semplicemente il compito di checkare i vari slot alla ricerca di spazzatura.

Il primo plan viene lanciato subito, dato che il triggering event è già presente nella belief base; quindi, se l'agente crede di non avere spazzatura in quello slot, si muoverà al prossimo, checkando (ma senza bloccare il plan). Il plan subito sotto invece, ha lo stesso t.event ma non fa nulla; ciò infatti modella il fatto che dobbiamo spostarci verso un nuovo slot soltanto se crediamo di non avere spazzatura.
Se il mio triggering event è invece garbage(r1), controllo con l'internal action desire che non abbia già il goal di portare la spazzatura al robot r2: se così non è, l'aggiungo fra i goal. Quest'ultimo obiettivo prevede diverse fasi: innanzitutto ricavo con un test goal la mia attuale posizione, memorizzandola e aggiornando tale valore alla copia più attuale denominata last. Porto così la spazzatura a r2 aggiungendo un achievement goal take, e torno poi alla posizione precedente continuando a checkare slots senza sospendere il plan. Il plan che ha come t.event take(S,L), con context vuoto (ovvero

```
at(P) :- pos(P,X,Y) & pos(r1,X,Y).

!check(slots).

+!check(slots) : not garbage(r1)
   <- next(slot);
      !!check(slots).
+!check(slots).

+garbage(r1) : not .desire(carry_to(r2))
   <- !carry_to(r2).

+!carry_to(R)
   <- ?pos(r1,X,Y);
      -+pos(last,X,Y);
      !take(garb,R);
      !at(last);
      !!check(slots).

+!take(S,L) : true
   <- !ensure_pick(S);
      !at(L);
      drop(S).

+!ensure_pick(S) : garbage(r1)
   <- pick(garb);
      !ensure_pick(S).
+!ensure_pick(_).

+!at(L) : at(L).
+!at(L) <- ?pos(L,X,Y);
           move_towards(X,Y);
           !at(L).
```

Figura 2.6: Codice AgentSpeak per i due agenti

true), si assicura che il robot r1 abbia preso effettivamente la spazzatura, si assicura che ci si trovi in L, e poi lascia la spazzatura S. Per gestire ciò abbiamo due subgoals: ensurepick per il primo task citato, che constatato il fatto che abbiamo la spazzatura in quella posizione, la prende, continuando ricorsivamente nell'operazione finché ci accorgiamo che la spazzatura non è più presente, ed è quindi stata presa. Ciò ci serve poiché il robot ha un livello di incertezza e non determinismo nella presa, che aiuta a modellare in maniera più simile alla realtà il nostro sistema. Infine il secondo subgoal controlla ricorsivamente che ci si trovi nella posizione L, e in caso contrario compie l'azione movetowards per farlo.

Il secondo robot ha soltanto un plan, attivato se viene percepita spazzatura nel luogo cui è il robot, e reso applicabile sempre dato il contesto vuoto. In quel caso, la spazzatura viene semplicemente bruciata. Questo primo esempio ha riepilogato un pò quello visto finora, tralasciando chiaramente la modellazione dell'ambiente e codice Java per le personalizzazioni in Jason. Vedremo questi aspetti nel proseguio della tesi, mentre ora ci si concentra sul reasoning cycle dell'interprete Jason, che ci aiuterà a capire come effettivamente funziona il framework decisionale per la gestione degli agenti nel nostro linguaggio.

2.4 Ciclo di Reasoning dell'interprete

2.4.1 Il ciclo nel dettaglio

E' essenziale capire come procede l'interprete nel suo ciclo di reasoning per poter programmare in maniera adeguata in Jason, poiché ci fornisce una visione dettagliata e allo stesso tempo d'insieme del framework cui la nostra applicazone ad agenti andrà a girare. L'immagine qui presentata è considerata la bibbia per il seguente capitolo, e verrà esplicata a dovere nel proseguio.

Il processo è divisibile in diversi step, e l'immagine può essere compresa meglio sapendo che quadrati, rombi e cerchi sono funzioni dell'interprete, personalizzabili dall'utente tranne quelle rappresentate da un cerchio. I rombi sono sostanzialmente delle selezioni di funzioni.

Notiamo che tutto inizia principalmente da una percezione dell'ambiente

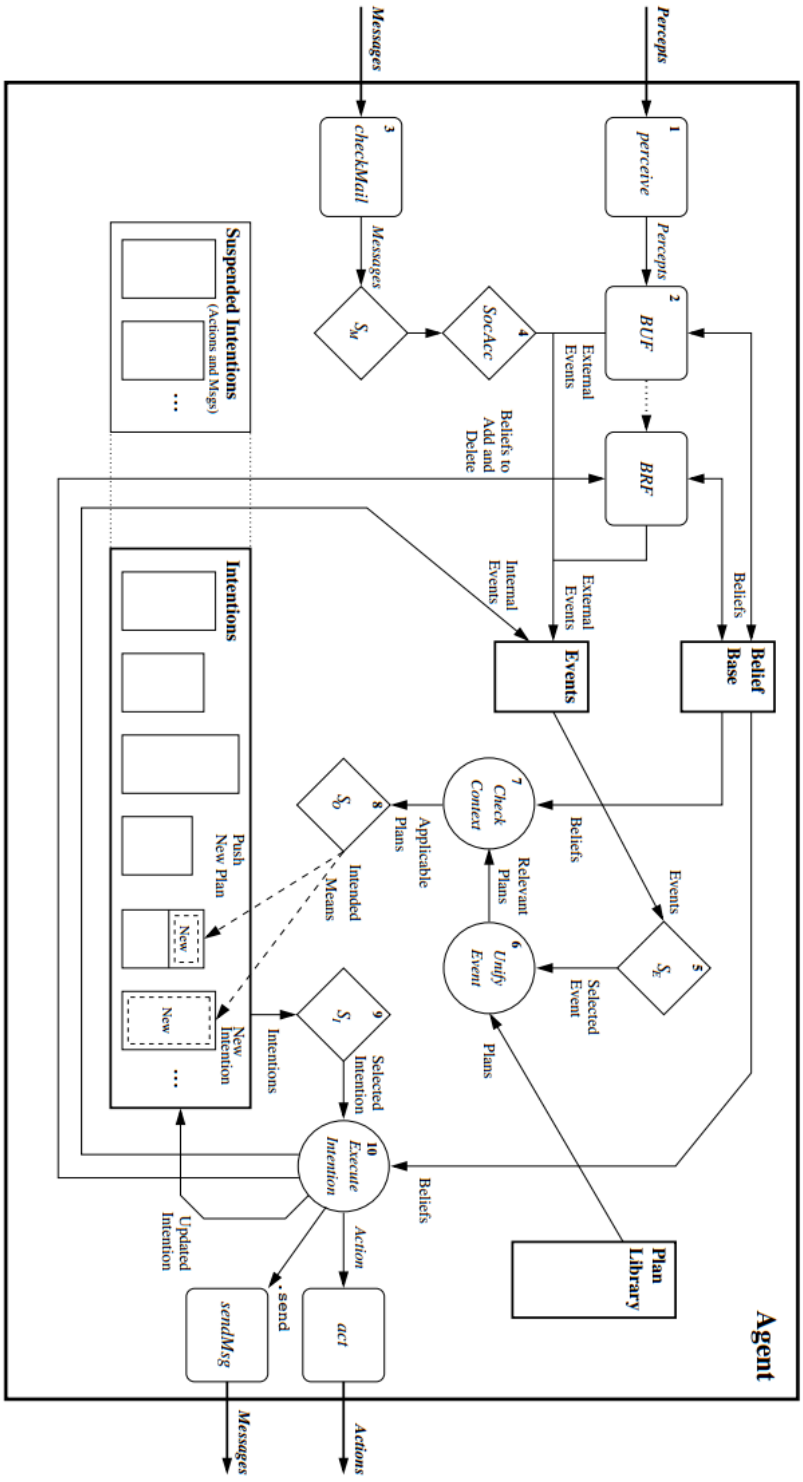

Figura 2.7: Il ciclo dell'interprete Jason

o da un messaggio. Nel primo caso, il metodo perceive serve a recuperare informazioni su una rappresentazione simbolica dell'ambiente, come ad esempio una classe Java nel nostro caso; se si usasse un agente nel mondo reale, con sensori reali, il metodo perceive dovrebbe interfacciarsi con tali dispositivi. In una visione avanzata della programmazione ad agenti, si potrebbe prevedere (anche un ambiente virtualizzato) la percezione diversa per gli stessi segnali da parte di agenti diversi, magari per simboleggiare la capacità dei loro sensori virtuali.

Dopo la percezione, è il momento di aggiornare la belief base, con la funzione BUF (belief update function), anch'essa personalizzabile; in genere l'aggiornamento di default fa sì che ogni literal presente nella percezione sia aggiunto alla base se non presente, e viceversa rimosso se presente nella base ma non nell'ultima percezione. Ogni aggiornamento nella base genera un evento, in questo caso denominato external. Se ad esempio l'agente percepisce il colore di una scatola vi sarà aggiunto un belief del tipo colour(box1,red)[source(percept)] e generato un evento con associata un intention vuota (denominata T, che approfondiermo in seguito). Le annotations giocano un ruolo chiave nella generazione di eventi in questo momento; se ad esempio è aggiunto un belief con annotations [a1,a2] e v' è già presente un belief con [a1,a3,a4], l'evento generato sarà soltanto del tipo +(..)[a2], ovvero soltanto sull'effettivo cambiamento; ciò massimizza le performance dell'interprete e dell'esecuzione stessa. Possiamo nell'altro caso ricevere un messaggio da un altro agente, messaggio che a sua volta genererà un evento non prima di esser passato per la funzione SocAcc, ovvero una gestione sociale degli agenti e dei messaggi che inviano fra di essi. I messaggi sono in genere processati nell'ordine cui sono ricevuti, ed ogni messaggio ha la stessa priorità degli altri; possiamo però personalizzare l'aspetto sociale, creando magari complesse relazioni tra più agenti, dove alcuni di essi vedono di buon occhio altri, magari favorendo la ricezione dei loro messaggi a discapito d'altri. L'aspetto è avanzato ma potenzialmente molto utile e flessibile, e potrebbe aiutare a gestire il comportamento da spammer o non desiderato (magari dovuto anche da errori di progettazione) di altri agenti presenti nello stesso sistema MAS.

Viene poi il crocevia fondamentale del ciclo in cui viene selezionato un evento, dalla funzione event selection (anch'essa personalizzabile), che ne

seleziona uno e uno soltanto per ogni reasoning cycle. Questa funzione di default seleziona dalla lista degli eventi l'ultimo, funzionando come una struttura a FIFO; spesso ciò non è consigliabile, ad esempio se un agente che rappresenta un soccorritore è pensato per dare priorità ad agenti gravemente malati, si dovrebbe modificare la sua funzione di selezione per far sì che tali eventi vengano prelevati anche se si trovano in fondo alla coda. L'evento selezionato è quindi rimosso dalla lista e si passa ad un fase successiva; se la lista è vuota si passa invece direttamente alla selezione dell'intention.

Vanno ora scelti i plan rilevanti, e ciò viene fatto controllando l'evento selezionato e i triggering event dei vari plan nella lista dell'agente. Il matching avviene in maniera molto precisa, ovvero si controlla per un evento di tipo colour(box1,blue)[source(percept)] che il triggering event preveda un colore ed oggetto generico (colour(Object,red) matcha soltanto con il colore rosso, troppo specifico) e matchi pure l'annotations, non considerando source di tipo mental notes o comunicazioni. Un matching per l'evento sopra presentato potrebbe essere il plan colour(Object,blue)[source(percept)]. Se non si trova un plan rilevante, l'evento è abbandonato; tale scelta è stata fatta poiché in genere un agente percepisce di continuo informazioni più o meno rilevanti, e si evita così di caricarlo troppo non considerando gli eventi di nostro interesse. L'ideale è avere quindi una lista di plan a design time che contenga plan con triggering event che riguardano tutto ciò cui vorremo che il nostro agente reagisca. Si raggiunge così un set, un insieme di plan applicabili. La fase appena descritta è detta di Unify Events.

Per scegliere i plan applicabili dobbiamo controllare chiaramente i loro contesti; un plan applicabile è in pratica un plan che ha la maggior probabilità (e a quanto pare tutte le condizioni) per avere successo. Si controlla quindi che le sentenze logiche presenti nel contesto siano vere, che le condizioni siano soddisfatte. Senza entrare troppo nel merito, magari con esempi tediosi, basta essere al corrente del fatto che si controllano quindi che certi statement siano corretti e che certi belief siano presenti nella beliefs base. Inoltre è da ricordare che in questa parte la funzione assegna dei valori ad eventuali variabili presenti, e tali valori permangono anche durante il passaggio dell'interprete dall'head al body del plan.

Se quindi si ha ora una lista di plan applicabili, ognuno dei quali può ragionevolmente portare ad un'esecuzione con successo, l'interprete deve per

forza scegliere uno fra i plan per l'effettiva azione, ed è il plan che andrà in un insieme denominato Set of Intentions dell'agente. Il piano è detto intented means, un corso d'azione che andrà eseguito di li a breve. La funzione che sceglie il plan è anch'essa personalizzabile, e di default li seleziona in maniera sequenziale; ciò è utile se ad esempio abbiamo dei plan ricorsivi e a design time li posizioniamo in modo che di default vengano eseguiti nell'ordine in cui li abbiam scritti. Abbiamo due modi per modificare il set di intentions: un evento esterno, quindi ad esempio una percezione dell'ambiente, crea una nuova intention, mentre un evento interno ne aggiunge una parziale in cima ad un altra. Questo perché l'evento interno è un evento causato durante l'esecuzione di un altro plan, magari con un subgoal all'interno, e aiuta quindi a creare uno stack di intention facendo capire all'interprete che sono corsi d'azione annidati e correlati fra loro. La plan library dell'agente non viene modificata, ma ciò che l'interprete utilizza in questa fase è soltanto una sua istanza (una copia) creata ad hoc. Va ora selezionata un intention ben definita dal set, e la funzione di selezione in questo caso va personalizzata per rendere gli agenti diversi fra loro e per assegnare ad ognuno di esse certe responsabilità; difatti, la priorità di certi goals rispetto ad altri identifica fortemente un agente e il suo ruolo nel sistema. Le intentions sono viste come una lista, e l'interprete di default gestisce lo scheduling con un algoritmo simil-round-robin, ovvero a turno si concentra su un intention ben precisa; di default quindi l'interprete garantisce un ottimo grado di fairness fra le intentions presenti.

Viene infine scelta l'intention da mandare in esecuzione, e abbiamo quella che è la fase finale del ciclo, dove dopo aver percepito l'ambiente, gestito eventi e ragionato su di essi e sui plans, è venuto il momento di agire e modificare in qualche modo l'ambiente cui siamo immersi. Ricordiamoci ora che un intention è un corso d'azione parziale, poiché include i goal, ma scegliamo il corso d'azione effettivo soltanto dopo aver percepito l'ambiente, per garantire maggior reattività possibile. L'esecuzione di un intention dipende dal tipo di formule presente nel body del plan che si trova in cima nell'intention scelta; le formule sono di sei tipi come già visto, e si agisce differentemente per ognuna di esse.

Per le environment action, ovvero le azioni sull'ambiente, esse sono realizzate da effectors, ovvero rappresentazioni virtuali di ciò che l'agente ha

per modificare effettivamente l'ambiente. E' stato pensato un meccanismo di sospensione durante tale esecuzione degli effectors, di modo che un'altra azione debba aspettare che quella precedente sia conclusa prima di procedere, e far sì che intanto l'agente possa far altro. L'intention sospesa viene posta in una struttura particolare aspettando di riprendere dopo un messaggio di feedback. I goals invece vengono sospesi ma non rimossi immediatamente dal body del plan, ma soltanto quando il plan finisce di eseguire. Lo stesso accade appunto sia negli achievement goal sia nei test goals quando non viene trovata l'informazione richiesta nella belief base, ma si cerca a quel punto un plan che soddisfi dale obiettivo. Nulla di particolare avviene per le mental notes, se non che viene loro assegnata una source di tipo self se non specificato altrimenti. Nelle internal actions invece viene eseguito direttamente il loro codice Java, ed è per questo che i metodi che le implementano sono metodi con ritorno booleano, poiché serve un meccanismo di feedback per capire se l'azione è stata eseguita o meno (non ci interessa se abbia avuto l'effetto desierato, ma soltanto che sia stata eseguita). In questo caso il comportamento di default (che può anche non essere adottato) prevede che le internal actions siano rimosse dal plan body e l'intention torni nel set, pronta per l'esecuzione.

E' utile far notare che non tutti i reasoning cycle portano l'agente a copiere azioni sull'ambiente, poiché in molti casi è di per sé già molto pesante l'elaborazione del perceiving dell'ambiente. Accorgimenti minori come controllare le intention sospese così come i means avvengono durante la fase finale di un ciclo di reasoning, ma i concetti fondamentali sono stati mostrati abbondantemente. Verrà ora presentato un breve focus su quella che è la gestione dei plan failure.

2.4.2 Plan Failure

Viene posta una certa attenzione nella gestione del fallimento dei plans, dato che quasi sempre un sistema multi agente è formato da agenti immersi in un ambiente imprevedibile, dove le azioni effettuate dagli effectors possono spesso fallire. Ciò comporta quindi il fallimento di plan e il bisogno di contingency plan, ovvero piani di contingenza a seguito di tali imprevisti. Se utilizziamo un triggering event nella forma +!g per arrivare all'obiettivo

g, il contingency plan per gestire il fallimento di g ha la forma -!g. Questi fallimenti possono presentarsi prevalentemente nei seguenti casi:

- Mancanza di piani applicabili o rilevanti per achivement goals: ovvero, in maniera molto semplice, l'agente non sa (non ha il know how) come eseguire un certo plan; in presenza di un subgoals ad esempio, non sappiamo come soddisfarlo. Potremo non avere effettivamente le conoscenze per farlo, oppure i plan disponibili contengono contesti non accettabili e quindi rendono i plan inutili.

- Fallimento di un testo goal: cerchiamo delle informazioni che pensiamo siano presenti nella beliefs base, ma effettivamente così non è. L'interprete, come già visto, cerca a questo punto di lanciare un plan che possa gestire il goal per ritornare tale informazione; se anche quest'ultimo tentativo fallisce, il plan stesso viene abbandonato.

- Fallimento delle azioni: semplicemente le azioni interne e quelle usuali sull'ambiente possono fallire, ad esempio per malfunzionamenti degli effectors; se l'azione fallisce, fallisce anche il plan cui essa si trova (nel body chiaramente).

In ogni caso, l'idea è quella per cui un plan per la goal deletion è un plan di clean-up, di pulizia, poiché si vuole in generale evitare che subito si ritenti di eseguire un plan appena fallito; l'ideale è magari effettuare alcune operazioni per poi riprovare in seguito. Questo è l'esatto motivo per cui si usano plan che hanno come triggering event una goal deletion (-!g). Se si vuole tornare indietro in ogni caso e ritentare sempre, la soluzione è chiaramente quella di creare un plan per una goal deletion in cui il context è vuoto (true) e viene rilanciato nel body un achievement goal per lo stesso obiettivo (!g). Il programmatore in questo aspetto dell'interprete Jason può personalizzare praticamente ogni cosa, è lasciata molta libertà.

In breve, se si ha una lista di plan dove ognuno di essi nel proprio body (ognuno con contesto vuoto) lancia il successivo, e l'ultimo (assumiamo il quinto plan) fallisce esplicitamente, viene generato un meccanismo a cascata; viene generato un evento (-g5) al fallimento di quest'ultimo plan, ma se non vi sono plan rilevanti per -g5, esso viene abbandonato con fallimento, e non potrà richiamare il plan p4 che era triggerato da +!g4, presente nel

```
MAS <mas_name> {

    infrastructure: <Centralised|Saci|Jade|...>

    environment: <environment_simulation_class> at <host>

    executionControl: <execution_control_class> at <host>

    agents: <ag_type1_name> <source_file> <options>
                            agentArchClass <arch_class>
                            agentClass <ag_class>
                            beliefBaseClass <bb_class>
                            #<num_instances> at <host>;
            <ag_type2_name> ...;
}
```

Figura 2.8: File di configurazione

body del plan fallito. Il fallimento perciò continua a cascata fino a g3, dove però è presente un failure plan, che è eseguito con successo. Da lì in poi continua l'esecuzione dei restanti plan uno e due in maniera normale, come se il plan g3 fosse stato eseguito correttamente senza problemi. In generale è tutto personalizzabile, e un programmatore, se non vi sono plan applicabili per una goal deletion o eventi esterni, può scegliare sia di eliminare l'intera intention o ripostare l'evento per un tentativo futuro.

2.4.3 Configurazione dell'interprete

Un sistema MAS, quindi un'applicazione lanciabile in Jason, è configurata tramite un file .mas2j, e lanciata dallo stesso. Un esempio della sintassi di tale file è qua visibile.

Il file include diverse configurazioni, qua ci si concentra su quella che è la personalizzazione possibile dell'interprete Jason, approfondito appunto in questa sezione. Ciò avviene nella parte denotata dal tag <options>, dove abbiamo diversi parametri:

- Events: può essere discard (di default), requeue e retrieve. Quando

non vi sono plan applicabili per un evento, con discard abbandoniamo l'operazione, con requeue lo riposizioniamo in fondo alla nostra coda, mentre con retrieve l'utente può simulare un comportamento particolare modificando la funzione selectOption. Può essere usata ad esempio per richiede ad altri agenti dei plan poiché non si ha il know how necessario per i nostri obiettivi.

- intBeliefs: con sameFocus (di default) agiamo come definito nel ciclo dell'interprete poco sopra, mentre con newFocus possiamo creare, dopo la ricezione di un evento esterno, differenti intentions che competono per l'attenzione dell'agente.

- Nrcbp: ovvero, number of cycles before perception. Come già appuntato, in simulazione e sistemi d'uso commune percepire l'ambiente è un task oneroso, e farlo assieme all'eseguire azioni nello stesso task diventa spesso proibitivo. Questo parametro, come specificato dall'acronimo, specifica quanto l'interprete deve ciclare prima di lanciare il perceive dell'agente. I valori vanno ad 1 (di default) in poi, dove con quest'ultimo valore percepiamo l'ambiente una volta a ciclo.

- Verbose: semplicemente, quanto l'agente deve stampare a video, riguardo la frequenza. I valori vanno da 0 a 2, dove con 0 stampiamo soltanto azioni di stampa esplicite, con 1 i risultati di azioni e messaggi scambiati, mentre con 2 ben più informazioni, che spesso aiutano in fase di debug.

- User settings: possiamo creare i nostri settings personalizzati, pur essendo questo un aspetto avanzato non trattato qua.

Possiamo configurare inoltre la modalità di esecuzione della piattaforma Jason, con l'executionContol visibile nell'immagine sopra. I valori sono asincrono, sincrono e debugging. Nel primo caso, di default, ogni agente agisce in maniera asincrona, indipendente, ed è spesso la maniera più utilizzata per avere sistemi realistici ed utili per ogni casistica. Con il valore sincrono invece abbiamo agenti che in qualche modo agiscono in sincronia, muovendosi in base ad una sorta di clock virtuale fornito dal platform Jason, che quindi regola le loro tempistiche. Il debugging è una forma ancor più avanzata di sincronismo dove l'utente può far riprendere l'esecuzione, step by step, dei

suoi agenti, per controllare magari in quale specifica circostanza si verifica un bug o un comportamento imprevisto.

2.4.4 Plan Annotations

Approfondiamo qua il discorso annotations, poiché esse possono offrire meta informazioni riguardo ai plan, e nella fattispecie, la maniera cui essi vengno interpretati da Jason. Difatti, esistono quattro parametri standard già disponibili e previsti nella distribuzione:

- Atomic: se un istanza di un plan con annotation atomic è scelta dalla funzione di selection intention, l'intention viene selezionata per esser eseguita nel ciclo successivo finché il plan non è finito. Ciò può essere utile quando si vuol far sì che non vi siano altre intention (quindi che la funzione non operi) fra le esecuzioni delle formule nel body del plan scelto.

- Breakpoint: è chiaramente utile per il debugging, dato che possiamo fermare l'agente e far riprendere l'esecuzione di tale plan (con appunto quest'annotation) quando vogliamo

- Priority: termine con arity uno che identifica la priorità di un plan durante la scelta per il plan applicabile. Maggiore è il valore, più alta è la probabilità che venga scelto. Può risultare molto utile quando si vuole privilegiare certi plan e personalizzare al massimo il comportamento del nostro agente.

2.4.5 Esempio: Domestic Robot

Per concludere a dovere questa sezione dedicata all'interprete Jason e a ciò che sta dietro le quinte, nel cuore della piattaforma, viene presentato un altro esempio simile al precedente, sperando che ora con le nozioni aggiuntive sia ancora più semplice da comprendere. L'esempio è presente nel libro principale di Wiley per Jason, cui ogni riferimento è presente nella bibliografia a fine tesi.

L'idea è quella di avere un agente robot che, su richiesta del suo possessore (altro agente), recupera dal frigo e consegna delle birre, acquistandole

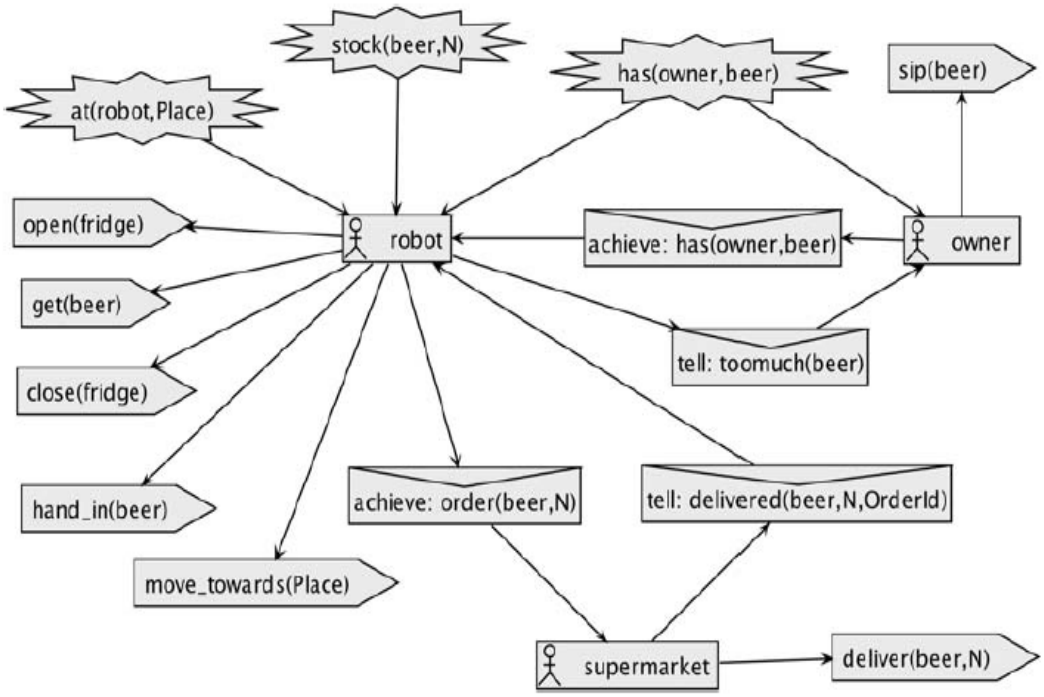

Figura 2.9: Modello Prometheus dell'esempio

in caso non ve ne siano abbastanza. Vi sono inoltre regole definite a priori per il massimo numero di birre giornaliere fornibili all'agente owner. Anche il supermarket dove vengono acquistate in caso le birre è un agente, per un totale di tre agenti nel MAS.

Tra le percezioni presenti per il robot, abbiamo !at, che ci informa se ci troviamo nel frigo, vicino all'owner o fra di essi; inoltre possiamo percepire con !stock e !has rispettivamente quante birre abbiamo in frigo e se l'owner in un preciso istante è in possesso di una birra. Il movimento del robot è semplificato, viene simbolicamente mosso di un passo per volta e il percorso possibile è soltanto dall'owner al frigo o verso il supermarket; sono previste azioni per far comunicare tra loro gli agenti, di tipo informativo (tell) o per l'achievement di determinati Goal. Viene mostrato il loro codice e poi dettagliatamente esplicato.

Owner Agent

```
!get(beer).  // initial goal

/* Plans */

@g
+!get(beer) : true
   <- .send(robot, achieve, has(owner,beer)).

@h1
+has(owner,beer) : true
   <- !drink(beer).
@h2
-has(owner,beer) : true
   <- !get(beer).

// while I have beer, sip
@d1
+!drink(beer) : has(owner,beer)
   <- sip(beer);
      !drink(beer).
@d2
+!drink(beer) : not has(owner,beer)
   <- true.

+msg(M)[source(Ag)] : true
   <- .print("Message from ",Ag,": ",M);
      -msg(M).
```

Figura 2.10: Codice agente Owner

Supermarket Agent

```
last_order_id(1). // initial belief

// plan to achieve the the goal "order" from agent Ag
+!order(Product,Qtd)[source(Ag)] : true
  <- ?last_order_id(N);
     OrderId = N + 1;
     -+last_order_id(OrderId);
     deliver(Product,Qtd);
     .send(Ag, tell, delivered(Product,Qtd,OrderId)).
```

Figura 2.11: Codice agente Supermarket

innanzitutto l'agente owner ha soltanto l'obiettivo iniziale di prendere birra, quindi verrà attivato il plan @g (con contesto vuoto) che manderà all'agente robot la richiesta di eseguire il goal has(owner,beer). Quando ciò accadrà, ad esempio con l'aggiunta nei beliefs tramite source percezione, verrà attivato il plan successivo che lancia il subgoals !drink. Quest'ultimo ricorsivamente farà sorseggiare la birra finché l'agente crederà fermamente che non ve ne sia più (secondo plan), attivando un plan con body nullo, che non fa nulla. L'ultimo plan è utile per stampare un particolare messaggio, rimuovendo il beliefs per far si che venga creato un evento soltanto se non era già presente nella base.

Il codice dell'agente supermarket è anch'esso piuttosto semplice, e prevede un belief iniziale per tenere traccia dell'id dell'ultimo ordine, e un plan per gestire il goal order dall'agente robot (qui denominato Ag). Nel plan, dove notiamo la source che deve aver lanciato l'evento per l'achievement goal è proprio Ag, il contesto è vuoto e il body prevede diverse formule; viene recuperato dalla belief base l'id dell'ultimo ordine, viene incrementato di uno e tale numero viene poi aggiornato nella belief base (con la sintassi -+ già vista). Tramite l'external action deliver viene mandata la birra richiesta e infine viene inviato un messaggio con .send, di tipo tell (informativo) all'agente Ag, contenente anche l'id aggiornato precedentemente.

Il codice del robot domestico è dei tre chiaramente il più complesso ed elaborato, poiché racchiude tutte le funzionalità del sistema. innanzitutto il belief iniziale prevede che il robot sappia che ci sia un certo numero di birre

Robot Agent

```
/* Initial beliefs */

// initially, I believe that there are some beers in the fridge
available(beer,fridge).

// my owner should not consume more than 10 beers a day :-)


/* Rules */

too_much(B) :-
   .date(YY,MM,DD) &
   .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
   limit(B,Limit) &
   QtdB > Limit.

/* Plans */

@h1
+!has(owner,beer)
   :  available(beer,fridge) & not too_much(beer)
   <- !at(robot,fridge);
      open(fridge);
      get(beer);
      close(fridge);
      !at(robot,owner);
      hand_in(beer);
      // remember that another beer will be consumed
      .date(YY,MM,DD); .time(HH,NN,SS);
      +consumed(YY,MM,DD,HH,NN,SS,beer).

@h2
+!has(owner,beer)
   :  not available(beer,fridge)
   <- .send(supermarket, achieve, order(beer,5));
      !at(robot,fridge). // go to fridge and wait there.

@h3
+!has(owner,beer)
   :  too_much(beer) & limit(beer,L)
   <- .concat("The Department of Health does not allow me ",
              "to give you more than ", L,
              " beers a day! I am very sorry about that!",M);
      .send(owner,tell,msg(M)).

@m1
+!at(robot,P) : at(robot,P) <- true.

@m2
+!at(robot,P) : not at(robot,P)
  <- move_towards(P);
     !at(robot,P).

// when the supermarket finishes the order, try the 'has'
// goal again
@a1
+delivered(beer,Qtd,OrderId)[source(supermarket)] : true
  <- +available(beer,fridge);
     !has(owner,beer).

// when the fridge is openned, the beer stock is perceived
// and thus the available belief is updated
@a2
+stock(beer,0)
   :  available(beer,fridge)
   <- -available(beer,fridge).

@a3
+stock(beer,N)
   :  N > 0 & not available(beer,fridge)
   <- +available(beer,fridge).
```

Figura 2.12: Codice agente robot domestico

presenti già nel frigorifero, viene impostato un belief per il limite di birra fruibile giornalieramente (10 bottiglie) e una rules toomuch(B) per gestire il limite, spiegata in seguito. Il robot ha tre plan principali per gestire l'ordine di birra dall'owner, tutti con lo stesso triggering event ma ovviamente con contesto differente. In caso la birra sia presente e l'agente creda non sia stato superato il limite giornaliero, entrerà in gioco il plan @h1 che tramite diversi subgoal porterà la birra all'owner. Tra i subgoals, !at utilizza l'action movetowards simile a quella presente nell'esempio precedente, e sfrutta la ricorsione finché l'agente non si trova dove vorrebbe. Prima di finire, il plan prevede l'aggiunta di !consumed alla base belief per ricordarsi in un determinato giorno e orario (recuperati tramite opportune actions) del fatto che abbiamo consegnato un atom di tipo beer (e l'owner l'ha quindi bevuta). Il secondo plan gestisce l'ordine al supermarket in mancanza di birre nel freezer, per poi far aspettare lì il robot in attesa della consegna. Per l'ordine viene usata chiaramente un action .send di tipo achieve. Nel plan @a1 notiamo la gestione dell'avvenuta aggiunta nella belief base di delivered, dopo un eventuale invio di birra da parte del supermarket. Il terzo plan principale entra in gioco (come si vede dal contesto di @h3) quando si controlla che vi sia un limite per l'item beer (ovvero 10, appunto presente nella belief base) e che la rule toomuch sia verificata. A proposito di quest'ultima, vediamo analizzandola che si recupera la data effettiva e si conta (tramite action count) il numero di istanze del belief consumed presenti nella base, filtrando per data e per item beer (non ci interessa l'orario). Il risultato, unito nella variabile QtdB, viene comparato con Limit, dopo che quest'ultima variabile è unificata con il limite già citato. Se la rule riporta true chiaramente, il limite è stato oltrepassato, e il robot giustamente manda un messaggio all'owner con i dettagli dell'impossibilità di eseguir tale richiesta. Gli ultimi due plan servono per aggiornare la belief base quando il robot si trova di fronte al freezer; in particolare @a2, lanciato se percepisco 0 birre mentre credevo ci fossero (dalla belief base), rimuove tale belief ormai obsoleto; @a3 invece, percependo che vi sono N birre e credendo che non ve ne fossero, con N maggiore di 0, aggiunge un belief di birra disponibile nella base. Il sistema è lanciato come già detto dal file di configurazione, che oltre alle opzioni già descritte comprende la specifica degli agenti presenti e del file che simula l'ambiente, analizzato nella sezione seguente. L'immagine seguente mostra l'ambiente nella sua implementazione con GUI minimale, con i due robot principali (non il supermarket) e i loro movimenti duran-

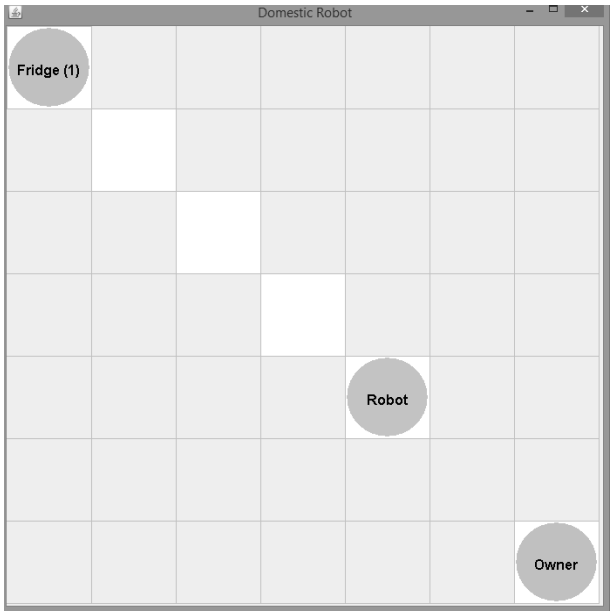

Figura 2.13: Applicazione Robot Domestico in esecuzione con GUI

te l'esecuzione. Verrà ora trattata la realizzazione dell' ambiente in Jason, in linguaggio Java, e per questo multipiattaforma e di grande impatto divulgativo e sociale, dato che tale piattaforma è ormai diffusa ovunque e conosciuta da praticamente ogni studente e/o professionista informatico.

2.5 Environments in Jason

Un aspetto cruciale di un MAS sviluppato in Jason e in generale, è la realizzazione dell'ambiente cui saranno immersi gli agenti. Essi dovranno percepirvi informazioni, eseguirvi azioni modificandolo, dovranno quindi trattare con l'ambiente e generare eventi relativi ad esso. In Jason l'ambiente è realizzato con una o più classi Java; tale scelta è stata fatta poiché viene più naturale e semplice gestire il tutto con un linguaggio ad oggetti piuttosto che vedere magari l'ambiente come un particolare tipo di agente. Sono in lavorazione linguaggi ad alto livelli specifici per questo obiettivo, ma per ora Java offre tutto ciò di cui abbiamo bisogno, garantendoci l'espressività del simulare un ambiente con oggetti dotati di uno stato interno e metodi

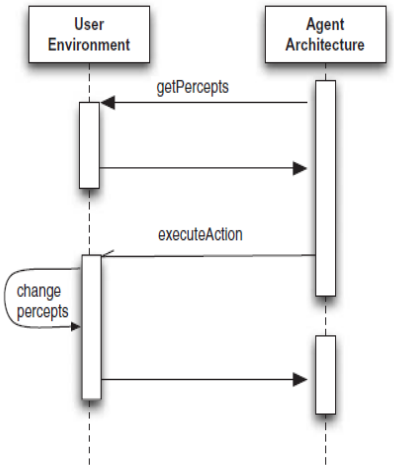

Figura 2.14: UML per la gestione degli Environments e Agenti

(quindi azioni) eseguibili su di essi. Un sistema multi agente spesso entra in gioco nel mondo reale, ovvero può pilotare sistemi ed agenti come droni reali che si interfacciano col mondo. Ma spesso un ambiente simulato è tutto ciò di cui si ha bisogno, durante simulazioni di società e/o più semplicemente testing preventivo per calcolare rischi e scelte in base alla parte algoritmica e software, prima di un effettiva implementazione hardware. In entrambi i casi, ciò che va cambiato è semplicemente il modo in cui si interfacciano i sensori ed i dispositivi hardware con il nostro modello, mentre quest'ultimo rimane valido a prescindere dalla tecnologia utilizzata. In generale poi, per sistemi più o meno complessi si usa il pattern Model View Controller, molto famoso nell'object oriented programming, e qui tornato in auge per questo scopo; offre una netta separazione fra modello dell'ambiente, un'entità che farà da tramite per gli agenti e l'ambiente e l'effettiva componente visuale, che può essere una GUI o meno. A tal proposito, si ricordi che ciò è esprimibile con un parametro inseribile nel file di configurazione più volte citato.

Dal modello UML qui mostrato vediamo come l'agente interagisce con quello che è l'implementazione dell'environment, che appunto estende la classe Environment fornita da Jason. Non è altro che il nostro Controller,

dove l'agente chiama getPercepts (che andrà implementato da noi assime all'altro metodo poi citato) per percepire informazioni, che saranno poi effettivamente fornite al ritorno di tale metodo. L'executeAction esegue un azione, che come abbiam visto, blocca l'intention (ma non l'interprete e il suo ciclo) finché essa non ritorna un risultato di fallimento o successo. ExecuteAction è eseguita in maniera asincrona, per garantire determinate performance anche a fronte del lancio di agenti e ambiente in macchine diverse, quindi di un MAS distribuito. Possiamo modificare anche il metodo init, per gestire informazioni d'inizializzazione, e abbiam la possibilità di personalizzare i percept a seconda dell'agente, oltre a gestire i percept globali. Sono disponibili metodi Java come addPercept(L), dove L è un literal gestibile tramite la suddetta classe offerta da Jason, oppure addPercept(a,L) con a singolo agente, per percezioni personalizzate. Possiamo inoltre rimuovere literal alla lista degli agenti e con clearPercept cancellare tutti i percept di una lista globale o di un agente specifico. Quindi in base alla struttra che riceviamo, all'agente che percepisce, ed altri parametri in ingresso, gestiamo le varie casistiche, in genere modificando i percept degli agenti dopo appunto un'azione; da ricordare che tali azioni possono fallire o meno, per questo il metodo è un booleano che ritorna true o false.

Vi sono due errori che spesso commettono coloro che da poco programmano sistemi MAS in Jason, parlando di environments: innanzitutto essi tendono a pensare che gli agenti si ricordino di tutti i percept, anche quelli che durano un solo reasoning cycle. In realtà gli agenti perdono tali informazioni, che sono memorizzabili tramite le mental notes, che perdurano finché non sono esplicitamente cancellate. Inoltre spesso avvengono errori di digitazione o di mismatch tra le nomenclature per nomi d'actions e literals utilizzate nel codice Java e in quello di modellazione d'agenti, overo l'AgentSpeak. Ecco due errori comuni da evitare poiché portano a problemi non facilmente debuggabili nel breve termine.

2.5.1 Model View Controller

La realizzazione in Jason dell'environment per l'esempio precedente del Domestic Robot consta di tre file Java, appunto l'environment, il modello e la view. Il primo, detto anche controller, fa da tramite fra agenti e ambiente, estende la classe Environment ed ha i metodi citati sopra. Ad esempio, se si recupera dall'istanza del modello (realizzato da un'altra classe) la posi-

zione del robot e la si controlla con quelle note dell'owner e del frigorifero, si può aggiornare il belief dell'agente facendogli sapere dove si trova, con l'aggiunta di un literal alla belief base. Dettagli di questo tipo verrano mostrati nel caso di studio presentato nel terzo capitolo della tesi, riferendosi al particolare progetto, ma sono del tutto simili a quelli presenti in questo esempio.

Nel modello (altra classe), in questo caso si estende la classe GridWorldModel, molto utile in tantissime casistiche, poiché simula un ambiente in cui vi possono essere oggetti ed agenti in forma di griglia, dove sono posizionabili oggetti in base a coordinate x,y a due dimensioni. Ciò facilita di molto il tutto, anche in fase di painting e rendering grafico. A proposito dell'aspetto visuale, esso è gestito dal componente view, l'ultima classe che assieme alle due precedenti realizza un MAS completo in Jason. Qui non si fa altro che disegnare un eventuale GUI in diverse maniere a discrezione del progettista. Insomma, si ha una netta seperation of concerns per ogni aspetto, com'é giusto che sia per un buon approccio ingegneristico, di modo che si possa sviluppare indipendentemente ognuno di essi.

2.6 Comunicazione e Interazione

Abbiamo già visto in maniera generica la comunicazione in Jason, ora approfondiremo tale aspetto concentrandoci sulla semantica e sintassi dei messaggi scambiati. In generale, un progettista deve prevedere specifici plan per la comunicazione, anche se in futuro sarà sicuramente automatizzato tale processo, tramite protocolli ben definiti. Ad ora, gli elementi chiave in un messaggio scambiato fra agenti, sono il sender, ovvero la denominazione dell'agente che invia, l'illocforce, ovvero la performative (che analizzeremo a breve) e il content, quindi il contenuto vero e proprio. Ad esempio, i parametri di una internal action classica per inviare messaggi, quale .send, sono esattamente questi tre valori, tra i quali il più interessante è sicuramente la performative, ovvero il tipo di messaggio inviato; possiamo così catalogare in maniera ben definita i messaggi tramite meta-informazioni. Le performatives disponibili sono qua spiegate, dove r denota il ricevente ed s colui che invia il messaggio:

- Tell: s comunica a r che crede che un certo literal sia vero

- Untell: s comunica a r che crede fermamente che un certo literal non sia vero
- Achieve: s richiede r di provare a raggiungere uno stato in cui il literal specificato è vero
- Unachieve: al contrario, s richiede r di rinunciare all'obiettivo sopracitato
- askOne: s vuol sapere da r se (crede che) il contenuto del literal sia vero
- askAll: simile a quello sopra, ma richiesto in broadcast a tutti
- tellHow: s informa r di un plan, quindi fornisce know-how
- untellHow: s richiede r di eliminare un plan dalla sua libreria
- askHow: s vuole tutti i plan di r rilevanti per un triggering event specificato.

Oltre .send possiamo usare anche .broadcast, di modo che tale messaggio sia inviato a tutti gli agenti della società, quindi presumibilmente a tutti quelli presenti nel nostro MAS. Ci sono diversi tipi di messaggio inviabili da s a r, vengono qui elencati in maniera non troppo approfondita poiché ciò esulerebbe dai nostri obiettivi primari:

- Scambio di informazioni: sono del tipo tell e untell, per il semplice scambio di informazioni fra agenti. E' utile quando si vuole comunicare un belief fra agenti, quindi informazioni relative all'ambiente. Ad esempio, una send con tell come performative e come contenuto il fatto che la porta sia aperta, crea nel ricevente un evento per l'aggiunta di tale belief nella sua base. Lo stesso messaggio con performative di tipo untell crea un evento per la rimozione di tale belief (-open(door)[source(s)]), la quale avverrà magari al reasoning cycle successivo.

- Goal delegation: la performative qua è di tipo achieve o unachieve, e si delegano degli obiettivi ad altri agenti; questi ultimi possono poi decidere se prendere azioni o meno, e in base ai loro plan se e come conseguire la richiesta. Ciò crea un aggiunta di un evento che causa l'aggiunta o la rimozione di una intention nella coda di intentions dell'agente che riceve. Da notare inoltre che riguardo l'unachieve, viene abbandonato di default ogni istanza dell'obiettivo specificato; a volte potrebbe essere utile rimuovere soltanto le istanze delegate dall'agente s, e questo può esser realizzato personalizzando il sistema, come (é già stato visto) è possibile fare in molti altri aspetti di Jason. Per maggiori informazioni, visitare la documentazione ufficiale.

- Ricerca di informazioni: del tipo askOne o askAll, sono richieste che bloccano l'intention finché non è ricevuta una reply (di default).

- Scambio di Know-How: la performative chiave è di tipo tellHow, untellHow o askHow. Chiaramente, il contenuto è una stringa, il cui parsing pruduce un plan perfettamente inseribile nella libreria dell'agente che riceve/invia. Ulteriori informazioni sono disponibili nella bibliografia a fine tesi e online sul sito ufficiale di Jason; ci fermiamo qui per quanto riguarda il livello di dettaglio in questo argomento, poiché diverrà sicuramente chiaro (se già non lo è) durante l'analisi del progetto presentato nel terzo capitolo.

2.7 Componenti user defined

Jason prevede un set di funzionalità vasto e che copre quasi tutto ciò che occore ad un progettista di un sistema MAS; non sarebbe però un ottimo linguaggio/framework se non offrisse libertà al progettista stesso. Difatti la visione di Jason è quella di lasciar libera la personalizzazione di molti componenti, dalle internal actions, alla classe Agent fino all'architettura globale del framework.

2.7.1 Nuove internal actions

Esistono già parecchie internal actions di default offerte da Jason, alcune delle quali già viste, come .send, .at e .print. Ricordiamoci che accedere alle internal actions in Jason è possibile tramite l'operatore punto, preceduto dal package se l'action non è tra quelle di default di Jason. Ogni classe che definisce una nuova actions dovrebbe estendere la classe DefaultInternalAction, che a sua volta implementa l'interfaccia InternalAction. Il metodo base da implementare, cui andrà la logica dell'operazione, è il metodo execute. I suoi parametri prevedono un sistema di transition, un unificatore (una funzione importante se il valore delle variabili in AgentSpeak va usato nel nostro codice Java) e infine la lista di parametri che l'azione stessa ha bisogno in ingresso. Un altro metodo della classe è suspendedIntention, che ritorna vero quando l'azione va sospesa, ad esempio con la performative askOne in .send. Di default ritorna false.

Ad esempio una funzione per calcolare una distanza generica, potrebbe prevedere in execute una prima fase di recupero parametri dal terzo parametro sopracitato, un calcolo ed infine una creazione di un literal d'uscita e il successivo bounding ad una variabile, che in caso può essere rappresentata dall'ultimo argomento della lista sopracitata(ad esempio, il terzo parametro di execute può prevedere un array args[5], dove i primi quattro valori sono parametri d'ingresso e l'ultimo d'uscita). Da ricordare che un internal action può ritornare un valore booleano (come già visto), oppure un iteratore di unifiers, in casi particolari. In generale poi è da ricordare che ogni agente crea un'istanza di un internal action inizialmente, per poi riusare quest'ultima ogni volta. Quindi le internal actions hanno uno stato, internalmente parlando per gli agenti.

2.7.2 La classe Agent

La classe Agent è di vitale importanza nel framework Jason, possiede diversi metodi, alcuni dei quali personalizzabili che sono mostrati nell'immagine seguente, in un diagramma esplicativo di tale classe.

Ci sono determinati metodi che vengono spesso sovrascritti (viene fatto il cosidetto overriding) dai progettisti, eccoli di seguito:

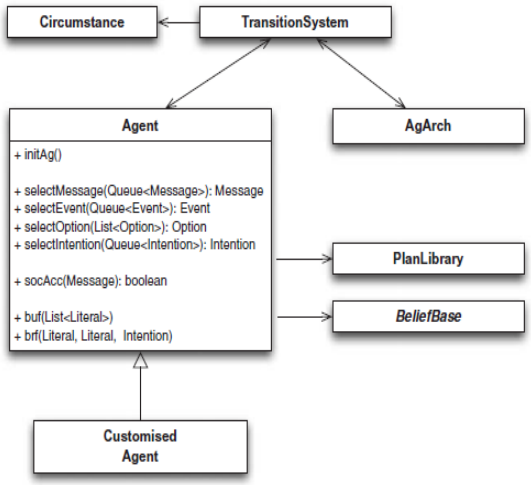

Figura 2.15: Modello della classe Agente

- SelectMessage: seleziona il messaggio per il reasoning cycle corrente. Di default sceglie e rimuove dalla lista il primo messaggio.

- SelectEvent: metodo duale a quello sopracitato, relativo però alla gestione degli eventi.

- selectOption: in questo caso si seleziona una tra le opzioni possibili per gestire un evento, quindi un plan applicabile e un unification. Recupera i primi elementi della lista di entrambi i parametri richiesti, di default.

- selectIntention: l'implementazone di default seleziona un intention per il reasoning cycle corrente e la esegue, posizionandola poi in fondo alla coda per i successivi cicli. Implementa di fatto un algoritmo round robin, che garantisce fairness. Se il set di intention già partito contiene una intention atomica, tale funzione non viene chiamata e viene selezionata la suddetta intention.

- socAcc: prende in ingresso un messaggio m, e ritorna true se esso è socialmente accettabile. Di default viene ritornato true in ogni ca-

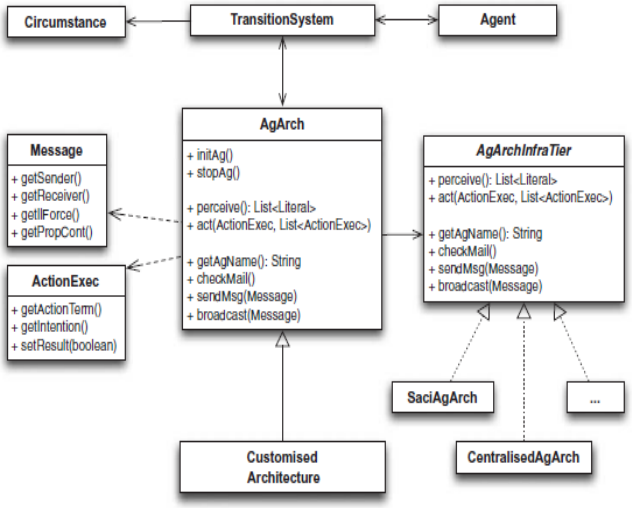

Figura 2.16: Modello dell'architettura globale

so. Per questioni di sicurezza è sempre consigliato l'overriden di tale metodo dato che di default il suo comportamento non è affatto sicuro.

- Buf e brf sono utilizzati per l'update della belief base tramite determinati percepts (nel primo caso) e literal da aggiungere e rimuovere (nel secondo). E' importante ricordarsi che è compito del progettista far sì che la belief base sia consistente, prevedendo meccanismi ed aggiornamenti frequenti per gestire tale problema.

2.7.3 L'architettura globale

Quella che intendiamo qua come architettura globale è l'insieme di proprietà che definiscono il comportamento che un agente Jason, di default, ha con l'ambiente. La BDI è sì un'architettura su cui si basano tali agenti, ma soltanto per la parte cognitiva e di reasoning.

L'implementazione per tale architettura è presente nella classe AgArch, da estendere se si vuole personalizzare tali aspetti; è sostanzialmente un ponte fra l'agente è l'infrastruttura sottostante, per cui in un sistema con

effectors e sensori hardware, qua il progettista dovrebbe gestire tali interfacciamenti con essi. I metodi più spesso sovrascritti non sono molti, abbiamo chiaramente perceive, per costumizzare ad esempio il modo cui gli agenti percepiscono l'ambiente in condizioni particolari, come di deficienza fisica o mentale (se si trovano dietro ad un muro ad esempio, percepiscono l'ambiente di fronte in maniera diversa). Act e sendMsg sono autoesplicativi, e di default richiamano le rispettive funzioni dell'infrastruttura sottostante, se non diversamente specificato. Altri metodi sono previsti per gestire l'invio di messaggi broadcast e per il checking dei messaggi stessi (checkMail).

2.7.4 Belief Base

E' possibile personalizzare anche la belief base, ciò è utile ad esempio per applicazioni su larga scala. I metodi principalmente sovrascrivibili sono add(literal) e remove(literal), per rispettivamente aggiungere e rimuovere il literal avuto come parametro in ingresso. Di default ciò viene eseguito ogni volta. Abbiamo poi contains, che ricerca il literal in ingresso nella belief base, ed infinte getRevelant, che tramite dei matching col literal in ingresso dà in uscita un iterator per una lista di literal rilevanti. Ad esempio, nella belief base con a(10),a(20),a(1,2) e b(f), se in ingresso al metodo citato si ha a(5), il risultato è dato dall'iterator per la lista composta da a(10) e a(20). E' possibile definire dall'utente la belief base sia salvando i belief in un file di testo che in un database relazione, molto utile per applicazioni su larga scala; ciò non viene comunque approfondito in questo documento. Vengono ora mostrati alcuni aspetti avanzati per la programmazione in Jason, assieme a consigli generici dallo stesso team di Jason per progettare i MAS in maniera adeguata.

2.8 Aspetti avanzati

2.8.1 BDI Programming

Ci concentriamo ora sugli aspetti avanzati e sui patterns, ovvero linee guide per risolvere problemi standard, riferendoci a quella che è la programmazione goal-based, quindi orientata ai goals. Non parliamo altro che di aspetti della BDI programming, poiché un desires della BDI può esser visto, senza ledere troppo la generalità, come un goals dell'agente.

In AgentSpeak in genere un context di un plan viene valutato osservando la belief base dell'agente, ma se in genere si vuole rifarsi sulle intentions o sui derires, o si vuole piuttosto rimuovere un goal, si deve ricorrere a pratiche eccezionali, che in Jason sono gestite tramite delle specifiche internal actions. Esse sono standard, quindi lanciate col prefisso dell'operatore punto, e sono:

- .desire: l'action ha successo se il literal ricevuto come parametro si unifica con un literal nel triggering event che ha la forma di una goal addition (+!g) negli eventi, oppure in un triggering event presente in una intention presente nell'insieme delle intentions.
- .intend: simile a .desire ma per una intention specifica
- .dropdesire: praticamente tutte le intentions e eventi che farebbero aver successo l'action desire verrebbero abbandonato
- Dropintention: tutte le intention che renderebbero intend true verrebbero abbandonate
- Dropevent: simile alle due sopra, da usare per eventi dove appaiono achievement goals
- Dropalldesires: non richiede parametri ed abbandona tutti gli eventi e le intention, inclusi gli eventi esterni
- Dropallintentions: non richiede parametri, elimina tutte le intention tranne quella corrente
- Dropallevents: non richiede parametri ed elimina tutti gli eventi, inclusi quelli esterni.

Analizziamo ora due importanti internal actions, usate solitamente in combinazione con la gestione del Plan failure, ovvero .succedgoal(g) e .failgoal(g). La prima viene usata quando l'agente crede che il goal g sia già stato conseguito e non servono altri plan per l'obiettivo. La seconda è usata quando l'agente capisce che è impossibilitato, per qualsiasi motivo, al conseguimento del goal, ed ogni plan che verrà comunque preso in carico per ciò fallirà. Nell'immagine osserviamo come, a partire da sinistra in cui sia un

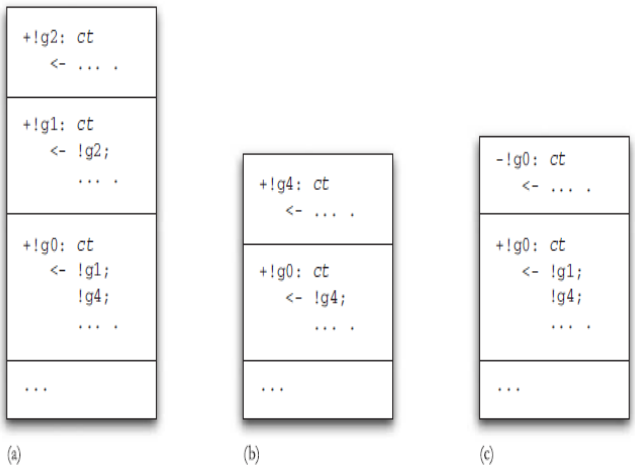

Figura 2.17: Esempio dell'utilizzo delle due internal actions succedgoal e failgoal

insieme di plan, nel secondo caso abbiamo il risultato dell'internal actions succedgoal(g1) mentre nel terzo caso quello dell'esecuzione di failgoal(g1).

2.8.2 Pattern utili

I pattern, ispirandosi alla programmazione orientata agli oggetti, dove hanno tutt'oggi grande successo, sono linee guide per risolvere problemi standard ben noti e spesso ricorrenti. In Jason, e nella programmazione BDI o goal-based, ci indicano come gestire i plan, e in pratica ci forniscono vincoli per la creazione di agenti ben programmati. Un plan standard per muovere un agente, utile in moltissime applicazioni, potrebbe essere visto cosi:

+!l(x,y) : bc(B) and B>0.2 <- go(x,y)

Bc indica il valore della batteria B e go permette al robot di muoversi, supponiamo. Il problema sta nel fatto che tale plan dovrebbe (e vorrebbe) essere usato per gestire il goal in maniera dichiaravita. Ma così non è. Gli autori di Jason danno molta enfasi al fatto che se per qualche motivo il

nostro plan non esprime un goal dichiaravito in maniera appropriata, abbiamo sbagliato la nostra programmazione o l'applicazione non è adatta a questo paradigma (degli agenti). In sostanza, un declarative goal è un goal che lancia un plan, dopo il quale l'agente teoricamente crede che ciò che è sentenziato nel goal (ovvero lo stato dell'ambiente) sia ora vero. Per completare correttamente il plan sopra nella giusta maniera in ottica dell'avere declarative goal, basta aggiungere un test goal, ovvero:

+!l(x,y) : bc(B) and B>0.2 <- go(x,y); ?l(x,y)

Verrà testato tale belief, e se andrà a buon fine l'agente crederà fermamente che il goal sia stato, effettivamente, conseguito. Abbiamo trasformato con poco un approccio procedurale in uno dichiarativo, ma abbiamo cambiato molto in quella che è la visione che abbiamo del funzionamento dei nostri agenti, e del sistema in generale.

Alcuni pattern vengono definiti con del meta-codice, utilizzando metavariabili e rules, ovvero regole che con un insieme di plan dati (in questo caso) gestiscono l'insieme con determinati obiettivi. La rule nota chiamata DG, ovvero declarative goal, non prevede altro che il passaggio da procedurale a dichiarativo sopra visto, aggiungendo ad un insieme di plan un plan iniziale che non fa nulla se il goal è stato conseguito, ed uno iniziale (sempre lanciato) che ha nel body l'action .succedgoal, per controllare che il goal sia raggiunto mentre sta eseguendo un plan, bloccandolo in caso. Un'altra rule EBDG, ovvero exclusive backtracking declarative goal, assicura che dato un insieme di plan nessuno di essi sia eseguito due volte, prima che il goal sia conseguito. Questi due plan erano del tipo declarative goal, chiaramente. Un altra serie di pattern, detti Commitment Strategy, hanno una visione simile a quella precedente ma con differenze. Ovvero, fanno si che l'agente cerchi in ogni caso di raggiungere l'obiettivo, il goal. Nell'EBDG precedente poteva accadere che l'agente abbandonasse il goal anche se esso non fosse stato conseguito; ciò viene evitato con il fanatical commitment, una rule che simula quello che è anche detto un comportamento fanatico verso un persistent goal. Il cercare ad ogni costo, fino alla fine di raggiungere un obiettivo è spesso un'asserzione troppo forte, a volte un robot o drone che sia non può semplicemente (magari per problemi di batteria) raggiungere certi obiettivi.

Altri pattern utili hanno luogo in situazioni non troppo particolari, che in certi ambiti possono risultare anzi fondamentali. Vi sono sistemi dove i maintenance goals sono di vitale importanza; ovvero, sistemi in cui l'agente deve controllare che un goal g sia sempre conseguito. Nella realtà gli agenti spesso falliscono in questo, ma ciò che possono fare è cercare di ri-conseguire g dopo che esso non è più valido. Un pattern che gestisce tutto ciò è definito dalla rule maintenance goal MG, che realizza appunto il fallimento in un goal maintenance. Un altro esempio di pattern utile è l'SGA, usato per gestire più goal in maniera sequenziale, poiché non si vuole che certe azioni siano eseguite concorrentemente. Si gestisce il tutto utilizzando la prima occorrenza del goal e poi memorizzando le altre come goal pendenti, sottoforma di beliefs speciali, che andranno poi riesumati uno ad uno non appena il goal corrente è stato conseguito del tutto.

La bella notizia è che una volta compresi questi pattern per gestire i plan, sono facilmente implementabili in Jason grazie alle direttive pre-processing. La sintassi è semplice, ed ha la seguente forma:

*begin nomepattern(goal) *elenco di plan da gestire* end*

Possiamo così utilizzare i vari pattern per tutti i plan che vogliamo in maniera molto comoda grazie a questa peculiarità del framework Jason, che come altre, ci facilita di molto il lavoro.

2.8.3 Consigli utili dal team di Jason

Gli sviluppatori di Jason hanno fornito una breve serie di consigli utili quando si programma per la prima volta un sistema MAS in Jason. Viene fatta molta enfasi sul fatto che si fa gran uso di Java per aspetti come l'environment e le internal actions, ma proprio per la grande diffusione del linguaggio e del numero di sviluppatori che lo conoscono, si corre il rischio che essi ne abusino a sfavore della programmazione logica in AgentSpeak. Andiamo per gradi:

- Ambiente: innanzitutto è necessario realizzare in Java una buona simulazione dell'ambiente, di modo che esso sia stabile e coerente, offrendo percezioni con un senso logico e pratico. Va realizzato prima

l'ambiente poi i vari agenti, e il primo va testato in maniera adeguata dato che un successivo debugging dello stesso con agenti che vi lavorano sopra diverrebbe fin troppo oneroso. Il problema con gli sviluppatori Java arriva quando essi abusano del codice ad oggetti per non utilizzare troppo l'AgentSpeak, producendo percezioni fasulle ed azioni fake. Tutto ciò è da evitare assolutamente.

- Azioni interne: in breve, l'aspetto di reasoning andrebbe lasciato completamanete all'AgentSpeak. Ci posson esser casi però, come l'utilizzo di codice legacy Java o calcoli pesanti, in cui si può utilizzare un codice imperativo del genere per l'aspetto reasoning. Non va comunque rimosso l'aspetto dichiarativo del reasoning dell'agente. Non è insomma da abusare troppo l'uso delle azioni interne in questa maniera, anche se effettivamente può portare minor danno di quello che ne porterebbe modificare l'ambiente Java per facilitare la nostra programmazione logica.

- Infine, viene fatta enfasi sullo sviluppo dell'interfaccia e dell'architettura dell'agente e la sua possibile estensione in Java, che permette, se ben progettata, di realizzare agenti immergibili in ambienti diversi, senza cambiare la loro architettura e quindi codice. Concludiamo con un ultimo aspetto integrativo di Jason con altre tecnologie prima di approcciarci al terzo capitolo della tesi.

2.9 Jason e JADE

JADE è un progetto italiano, almeno inizialmente, che ha portato alla creazione dell'omonimo middleware e insieme di tecnologie che permette la creazione e lo sviluppo di MAS. JADE si basa completamente su Java, quindi anche gli agenti, oltre all'environment, sono scritti in codice ad oggetti, ma con un approccio ad agenti chiaramente. Offre inoltre librerie e un ambiente runtime per gestire gli agenti e si basa sulle direttive FIPA per la comunicazione inter-agente.

JADE è uno standard de facto al giorno d'oggi come framework ed environment generale per far coesistere fra loro agenti. Per questo, Jason permette l'integrazione con JADE potendolo selezionare come infrastructu-

Figura 2.18: JADE, un'infrastruttura per il development di MAS

re (lo si può fare modificando la suddetta voce nel file di configurazione di ogni applicazione Jason). Personalmente ho analizzato in fase di scelta del caso di studio per questa tesi sia Jason che JADE, fra gli altri, e quest'ultimo utilizza un approccio al quale ho preferito l'architettura BDI, a mio parere più espressiva per il reasoning degli agenti. Ma JADE oltre ad un'infrastruttura solida e largamente utilizzata, offre degli utilissimi tool grafici per monitorare le azioni degli agenti, che risultano disponibili anche ai programmatori Jason quando lo si utilizza come infrastruttura per il progetto.

Come specificato nel sito di Jason sulle faq (Frequently Asked Questions), l'infrastruttura di JADE è da usare quando si vuole avere un'infrastruttura distribuita, con agenti che eseguono in diversi host collegati in rete, oppure quando si vogliono far convivere nello stesso sistema MAS agenti non necessariamente sviluppati in Jason. Se si vogliono questi due features, è consigliato l'uso di JADE. Inoltre in generale, lo si può fare comunque ogni qualvolta si voglia usufruire dei comodissimi tools grafici per la diagnostica del sistema che offre JADE, come lo sniffer, che intercetta tutti i messaggi scambiati fra agenti e fra agenti e ambiente. Jason offre sì nella sua versione base un tool di diagnostica ma semplicemente quelli di

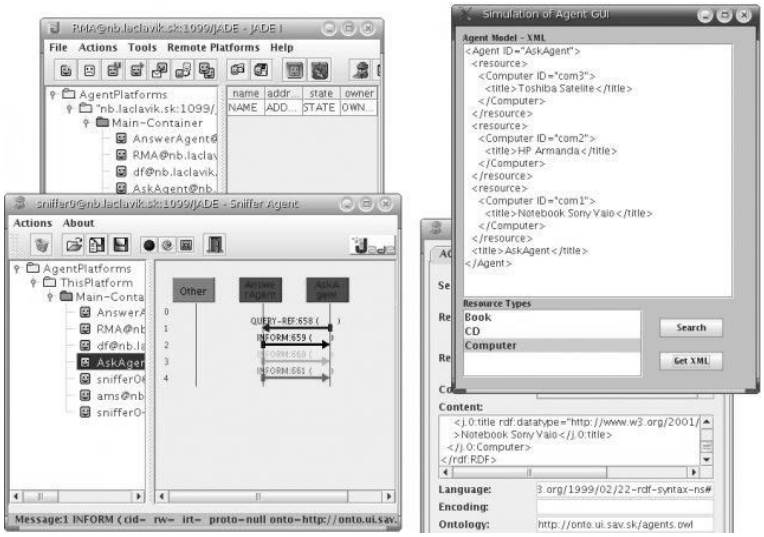

Figura 2.19: I tools grafici di JADE

JADE sono più completi e semplici da usare. Maggiori informazioni sono disponibili online sul sito del progetto.

2.10 Conclusione

Si conclude questa seconda parte che rappresenta un pò il cuore della tesi, senza la comprensione della quale perde di senso il capitolo tre ed ogni possibile sviluppo con questo linguaggio. Si è cercato di esprimere in maniera il più possibile chiara e diretta tutti i concetti, gli aspetti, i costrutti e le metodologie per programmare ogni tipo di sistema MAS con Jason, anche per coloro che non hanno conoscenze di programmazione logica e agenti. Jason è stato scelto, lo ribadiamo, per essere un linguaggio orientato agli agenti tra i più (anzi forse il più) diffusi ed apprezzati dalle community del settore. Inoltre la simbiosi con JADE e il largo utilizzo di Java per definire molti aspetti lo rende interoperabile e multi-piattaforma. Gli sviluppatori hanno previsto futuri cambiamenti, tra cui l'uso di conjuctive goals o parallels goals, inizialmente evitati per non caricare troppo il livello di difficoltà nella progettazione. Ma cosa porterà Jason in futuro al settore di riferimento? I suoi sviluppatori hanno individuato alcuni punti su cui stanno focalizzando

la propria ricerca e sviluppi futuri, eccoli elencati brevemente:

- Organizzazione: l'organizzazione e l'aspetto sociale sono, come abbiam visto, cruciali in un sistema MAS, ma Jason ora come ora offre soltanto supporto per la semplice comunicazione. Pattern vari sono sì possibili, ma implementabili dal progettista. In futuro sarà possibile avere funzionalità direttamente offerte da Jason come la nozione di ruolo, gruppi, e norme sociali, che favoriranno e faciliteranno il lavoro.
- Scambio di plan: verrà fatta ancora più enfasi e fornito più supporto per lo scambio di plan fra agenti, quindi la fruizione di know-how verso quegli agenti che richiedono aiuto. Sarà basato sul meccanismo di scambio plan dell'archiettura Coo-BDI.
- Verrà revisionata la belief base introducendo il supporto alle ontologie.
- Riformulazione della visione degli ambienti, con oggetti che possono venire osservati o meno sulla base di norme sociali e organizzazioni fra agenti. Anche l'uso di uno specifico linguaggio ad alto livello per definire tali tipi di ambienti potrebbe sopperire l'uso di Java in futuro.

Insomma, c' è tanto lavoro previsto e di certo molto altro di cui non possiamo ancora essere a conoscenza. Per ora concentriamoci su quello che offre Jason, ed andiamo a toccare con mano nel dettaglio la progettazione di un sistema MAS nel terzo capitolo, che concluderà il percorso con larghi esempi su casi d'uso reali. Prima di proseguire, viene mostrato un punto della situazione degli argomenti principali trattati finora, per accompagnare al meglio il lettore verso l'aspetto pratico e non rimanere spiazzato ad un primo impatto con codice e modelli.

- Un sistema multi-agente (MAS) è composto da più agenti che convivono nello stesso ambiente. Un agente è un'entità autonoma con capacità sociali, attive, proattive e reattive rispetto all'ambiente cui è immersa. Si differenziano principalmente dagli oggetti poiché possono decidere, sulla base di politiche di reasoning, se eseguire certe azioni richieste o meno.

- Esistono diversi tipi di agenti, i quali vengono utilizzati in diversi ambiti dell'informatica, e in futuro saranno il cardine della programmazione a fianco a quella ad oggetti, poiché ci si sposta verso un mondo dove l'interazione è cruciale, come lo era la computazione fino ad oggi.
- E' stato introdotto un linguaggio di programmazione logica, AgentSpeak, la cui estensione può essere interpretata dal framework di Jason. Tale linguaggio si basa sull'architettura BDI.
- BDI significa beliefs, intentions, desires. Un agente è quindi descritto sulla base di ciò che crede di sé stesso e dell'ambiente (beliefs), su quello che vorrebbe che fosse l'ambiente (desires o goals) e sui piani d'azione che può eseguire per far sì che accadano tali cambiamenti (intentions o plan). Un plan nello specifico è composto da triggering event, context e body, ovvero ciò che fa scattare un plan, ciò che lo rende applicabile o meno, e le azioni che andranno eseguite, rispettivamente.
- Il ciclo di reasoning dell'interprete di Jason gestisce il reasoning degli agenti, dalla percezione di cambiamenti o messaggi ricevuti, al processo che fa sì che tali percezioni portino a determinate azioni o cambiamenti mentali. Viene analizzato con cura e in ogni casistica possibile poiché aiuta a capire nel dettaglio come eseguirà ogni nostra applicazione in Jason.
- Jason è altamente personalizzabile, tra internal actions (quelle azioni che non modificano l'ambiente), classe Agent (ovvero l'architettura singola dell'agente) e architettura globale. Tutto ciò è scritto in Java. Ciò aiuta molto coloro che sono già familiari con tale linguaggio, ma li mette anche in guarda nel non abusare troppo di codice legacy ma dare invece importanza anche all'AgentSpeak per l'aspetto di reasoning.
- Jason è utilizzabile con infrastruttura centralizzata, in locale, oppure per MAS distribuiti è possibile utilizzare JADE, che offre la simbiosi con agenti realizzati con tale framework e la possibilità di utilizzare tutti i tools grafici che nativamente offre.
- Vengono mostrati durante il percorso due esempi pratici di applicazioni in Jason. Si consiglia caldamente la loro comprensione per continuare nella lettura.

Capitolo 3

Realizzazione di un MAS in Jason

3.1 Un caso di studio reale

In questo capitolo ha luogo l'analisi della progettazione e realizzazione di un'applicazione completa di un MAS scritto in Jason, basato su un caso d'uso reale. Più nello specifico, si è partiti da un progetto commerciale in fase di sviluppo e lo si è usato come prototipo per risolvere un problema simile. Chiaramente, la soluzione da me proposta non è assolutamente una soluzione che risolverebbe nella realtà il caso reale, bensì risolve quello che è una simulazione molto semplificata, dove non vengono trattati vari aspetti implementativi che avrebbero complicato il tutto nonché sviato la soluzione dai nostri interessi. D'altra parte il caso reale, seppure semplificato e simulato in scala molto ridotta, ha fornito un ottimo problema la cui soluzione si sposa alla perfeziona con la filosofia della programmazione ad agenti, e quindi con il nostro linguaggio Jason.

3.1.1 Amazon Prime Air

Il colosso dell'e-commerce Amazon ha divertito un pò tutti qualche tempo fa (fine 2013) quando ha rilasciato un video ufficiale dove venivano mostrati piccoli droni, dalle dimensioni ridotte e dal design accattivante, che prelevavano da un deposito Amazon piccoli pacchi per poi consegnarli via aria al domicilio di un cliente. Amazon ha poi stupito tutti rilasciando una di-

Figura 3.1: I futuri droni postino di Amazon

chiarazione in cui ammise che il video non mostrava un gioco o uno scherzo, bensì era da vedersi come la prospettiva di quello che è un progetto a cui stanno attivamente lavorando.

Dopo diverse indiscrezioni, tra cui il fatto che Amazon abbia ufficialmente chiesto al governo Americano norme per governare la mobilità di questi droni civili, è stato chiarito ufficiosamente che il progetto diverrà realtà nel giro di 3-4 anni, almeno nel territorio americano. Questi droni potranno trasportare pacchetti sotto il kg, quindi l'86 percento del traffico totale di Amazon, in soli trenta minuti, visti i vari centri di smistamento disposti in maniera strategica nel territorio. E' tutto incredibilmente affascinante e futuristico, ma quantomai attuale.

Citando Amazon dalla sezione apposita sul suo sito ufficiale:

We're excited to share Prime Air, something the team has been working on in our next generation RD lab. The goal of this new delivery system is to get packages into customers' hands in 30 minutes or less using unmanned aerial vehicles. Putting Prime Air into service will take some time, but we

will be ready as soon as the FAA grants permission. It looks like science fiction, but it's real. From a technology point of view, we'll be ready to put Prime Air into service as soon as the necessary regulations are in place. The Federal Aviation Ad-ministration (FAA) is actively working on rules for unmanned aerial vehicles. The FAA is actively working on rules and an approach for unmanned aerial vehicles that will prioritize public safety. Safety will be our top priority, and our vehicles will be built with multiple redundancies and designed to commercial aviation standards.

Inoltre, mentre sto scrivendo questa tesi, è trapelata la notizia che pure il colosso Google sta sperimentando e lavorando su un progetto simile, denominato Google Project Wind. Sarà utile per aspetti commerciali e per migliorare quello che era il servizio delle Google Cars che hanno aiutato a definire l'immenso servizio che Google Maps offre oggi. In generale, aiuterà Google a raccogliere ancor più informazioni in maniera diversa e meno invasiva per rendere ancora più vasto il data-set dal quale può attingere per i propri scopi commerciali. Anche Facebook ha previsto l'utilizzo futuro di droni volanti per altri scopi, ovvero rendere possibile un accesso internet nei paesi in via di sviluppo, per permettere loro di accedere a servizi che ora non possono permettersi. Come già detto, non vi sono soltanto problemi tecnici e di progetto ad impedire sviluppi nel breve termine ma rigide regole e leggi, soprattutto nel territorio americano (dal quale partiranno tutte le prime fasi di test) per regolamentare il volo di questi piccoli ma potenti robot. Sono già previsti però sviluppi legislativi già dalla fine di quest'anno (2014).

3.2 Amazon Prime Air Simulation

3.2.1 Modelliamo il caso Amazon

Inanzitutto riformuliamo il problema che si ha di fronte dal momento che vogliamo basarci sul caso Amazon, ma allo stesso tempo modellarlo secondo determinate caratteristiche che ci interessano nello specifico, tralasciandone altre.

Un possibile problema da risolvere con la suddetta applicazione potrebbe essere qualcosa del tipo:

Si vuole produrre un'applicazione software ad agenti, che preveda la simulazione dell'esecuzione di droni postino in un ambiente a griglia. I droni una volta ricevute le informazioni sul proprio obiettivo dovranno autonomamente e proattivamente consegnare il pacco, reagendo al cambiamento della propria posizione nonché allo scontro con altri droni e a situazioni estreme come problemi di autonomia (batteria ad esempio). Viene data libertà allo sviluppatore riguardo possibili scenari, numero e peculiarità degli agenti per mostrare al meglio gli aspetti di autonomia e abilità sociali degli stessi, tralasciando aspetti computazionalmente onerosi.

Da tali richieste, vengono estrapolati i seguenti requisiti che saranno poi implementati come vedremo nella soluzione finale:

- Due scenari, ognuno con caratteristiche e numero di agenti diverso
- Possibilità di interagire real-time con l'ambiente producendo cambiamenti percepibili dagli agenti
- Possibilità di seguire tramite GUI e pannello di Controllo GUI l'esecuzione, oltre che dall'ambiente di debug stesso di Jason
- Ambiente descritto a griglia
- Vengono tralasciati aspetti computazionalmente onerosi come il calcolo del percorso migliore per gli agenti.

E' necessario fare un appunto su quest'ultima questione: è stato scelto di precaricare nell'applicazione due percorsi che collegano quattro destinazioni ben precise, e tutto ciò non è modificabile ne è possibile fornire agli agenti una destinazione diversa. Se ad un primo sguardo ciò può sembrare limitante, si ricordi che questa è una simulazione ad hoc con l'intento di apprendere l'approccio ad agenti e soprattutto mettere in risalto gli aspetti di reasoning degli agenti, quindi dando molta importanza al codice AgentSpeak rispetto al codice Java, utilizzato come semplice tramite fra le due architetture e come linguaggio per descrivere l'ambiente.

Inoltre, la scelta è ancor più sensata se si pensa al verosimile comportamento che avranno questi droni nella realtà; essi non calcoleranno ogni

volta internamente il percorso migliore (benché non avrebbero alcun problema nel farlo) ma lo recupereranno da sistemi cloud o distribuiti basati su geolocalizzazione e sistemi di mappe. Principalmente quindi è stata scelta questa semplificazione poiché non è di nostro interesse e abbiamo voluto concentrarci sull'AgentSpeak, ma ci ha comunque fornito una simulazione della realtà più che discreta e sicuramente migliore del vedere eseguire agenti che cercano, con movimenti più o meno efficaci, di arrivare prima o poi a trovare il percorso per la destinazione.

Viene ribadito quindi che la simulazione è pensata per scopi didattici di apprendimento, e per questo si concentra su certi aspetti; si presta inoltre ottimamente per una breve presentazione molto diretta riguardo determinati concetti prima esposti oralmente.

3.2.2 Analisi dei requisiti

Verranno ora analizzati nel dettaglio gli scenari previsti nell'applicazione, nonché i modelli che descrivono in una prima fase gli aspetti di Reasoning degli agenti. Tutto ciò si basa su una prima versione già ben revisionata del software, eventuali cambiamenti all'ultimo minuto non verranno mostrati qua. Chiaramente si tratterà al massimo di scelte stilistiche o che migliorano leggermente le performance, quindi nulla che tolga validità a ciò che viene mostrato qua poiché i concetti chiave rimangono gli stessi.

Innanzitutto introduciamo i due scenari: la scelta è ricaduta sui seguenti:

- Scenario 1 : sono presenti due droni, uno leader ed un secondo detto robot1, ovvero il drone postino. Le caratteristiche di questo scenario sono la possibilità tramite il pannello GUI detto ControlPanel di creare runtime una percezione fittizia per gli agenti, per simulare un'improvvisa carenza di batteria e vedere la loro reazione. Ciò come vedremo comporterà un arresto del drone che segnalerà la sua posizione al leader per il prelievo d'emergenza. In questo scenario è stato introdotto non-determinismo (pseudo) per il picking del pacco da parte dell'agente. Sarà possibile velocizzare o rallentare l'esecuzione a piacere.

- Scenario 2 : sono presenti tre droni, uno leader e due postino detti robot1 e robot2. Il codice di questi ultimi si differenzia per pochissimo. Difatti è stato previsto uno scontro in un determinato punto dei due percorsi, dove gli agenti dovranno reagirvi (il percorso è sì prestabilito, ma gli agenti lo percepiranno come 'improvviso' ogni volta simulando appunto il comportamento dovuto a percezione e reazione). Non sarà possibile segnalare una percezione di carenza di batteria. Si potrà velocizzare o rallentare l'esecuzione a piacere.

3.2.3 Analisi del problema

Analizziamo ora, seguendo quello che potrebbe essere uno pseudo-documento per una produzione di software documentata e non eroica, l'analisi del nostro problema. Nel primo modello qui mostrato osserviamo in maniera grafica, ispirandoci ad una versione semplificata e rivisitata graficamente della notazione Prometheus quelli che sono i principali aspetti degli agenti.

Come vediamo robot1 e robot2 sono molto simili fra loro, e prevedono operazioni basilari come muoversi verso una posizione, prendere e lasciare il pacco, ecc. Nel paragrafo inferiore abbiamo quelle che sono le percezioni possibili, e come vediamo i due robot postino possono percepire dove si trovano rispetto alle posizioni d'interesse già note, quindi partenza e destinazione, dove si trovano in assoluto (rispetto a coordinate x,y), se possiedono il pacco da trasportare ed altro ancora.

Si noterà che l'agente leader ad una prima occhiata non possiede azioni disponibili ma ciò è erroneo: difatti quelle mostrate sono azioni implementate ad hoc dallo sviluppatore per questa applicazione, e non vengono quindi mostrate le azioni interne previste dal framework Jason, le quali sono invece frequentemente usate specialmente dall'agente Leader.

Non vengono inoltre mostrati plan e goal poiché le percezioni sono intese come aggiunta di beliefs nella beliefs base e non come l'attivazione di determinati goal e plan.

Nel modello seguente invece ci rifacciamo al modello di interazione dell'UML 2.0 riproponendolo in una veste grafica semplificata poiché più facilmente fruibile per i non informatici (per la seguente visione, si legga

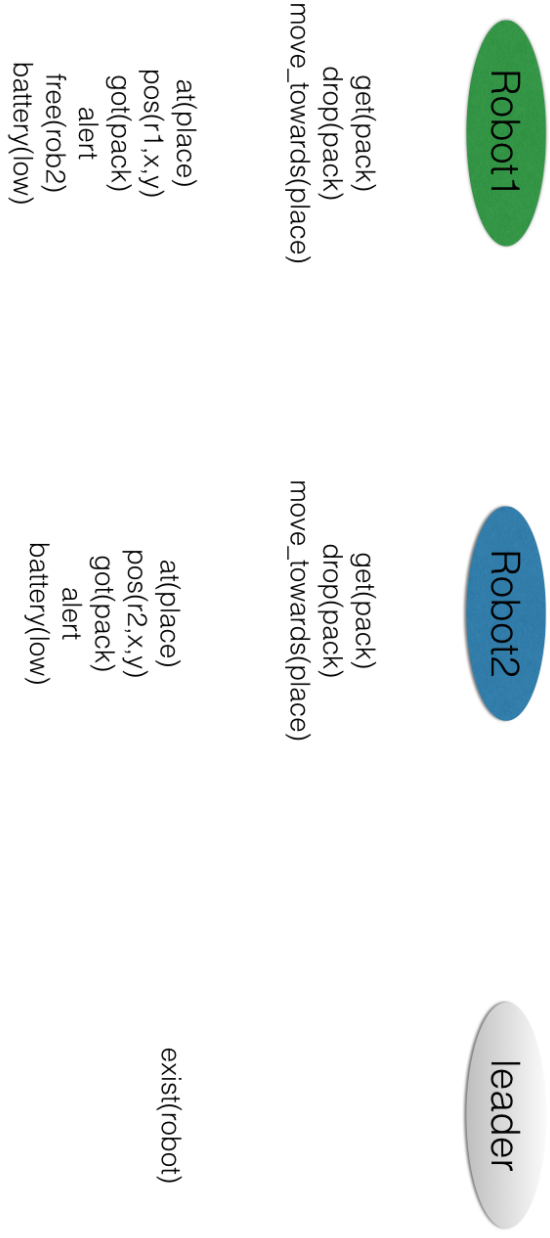

Figura 3.2: Beliefs e azioni non interne degli agenti

l'introduzione di questo documento).

Qua con un semplice diagramma notiamo in maniera istantanea i flussi d'interazione message based che regolano l'intera applicazione, in entrambi gli scenari. Sopra ad ogni linea abbiamo la definizione del messagio KQML (con la tipologia e il contenuto), e si noti che linee senza definizione vanno associate a linee dallo stesso colore con il suddetto dettaglio in più, evitato per motivi di spazio.

Inizialmente l'agente leader non comunica né prende alcuna iniziativa, ma i suoi plan vengono attivati dai messaggi di 'exist' di robot 1 e robot 2 (nello scenario 1 soltanto robot1). Difatti tale messaggio di tipo TELL aggiunge alla belief base dell'agente leader tali belief, che renderanno applicabili certi plan e daranno il via alla consegna dei pacchi. La consegna ha inizio con l'ordine da parte dell'agente leader inviato con modalità ACHIEVE, quindi viene aggiunto un vero e proprio goal da conseguire per gli agenti postino. La comunicazione è quindi sia informativa sia volta a perseguire determinati obiettivi generici dell'intera applicazione. Non scordiamoci che un'altra peculiarità della comunicazione fra agenti non mostrata qui è il passaggio di know-how inter-agente.

La linea denominata TELL: free(rob2) è utilizzata nel secondo scenario e viene inviata dal robot2 al robot1 secondo questo protocollo: per evitare lo scontro fra i due agenti, il primo robot attende che venga aggiunta alla sua belief base un certo belief, il quale verrà aggiunto solo e soltanto se inviato dal robot 2. Il robot 2 difatti, una volta avvertito il pericolo, invierà tale messaggio e continuerà sul suo percorso, mentre il robot1 sarà in attesa finché non lo riceverà, evitando così che essi si scontrino.

Questo è appnto un esempio di comunicazione per coordinare gli agenti. Gli ultimi due messaggi possibili sono il messaggio di sos da parte dei due robot postini verso l'agente leader, che vedendoseli arrivare reagirà stampando posizione ed altre info utili per un eventuale recupero del drone.

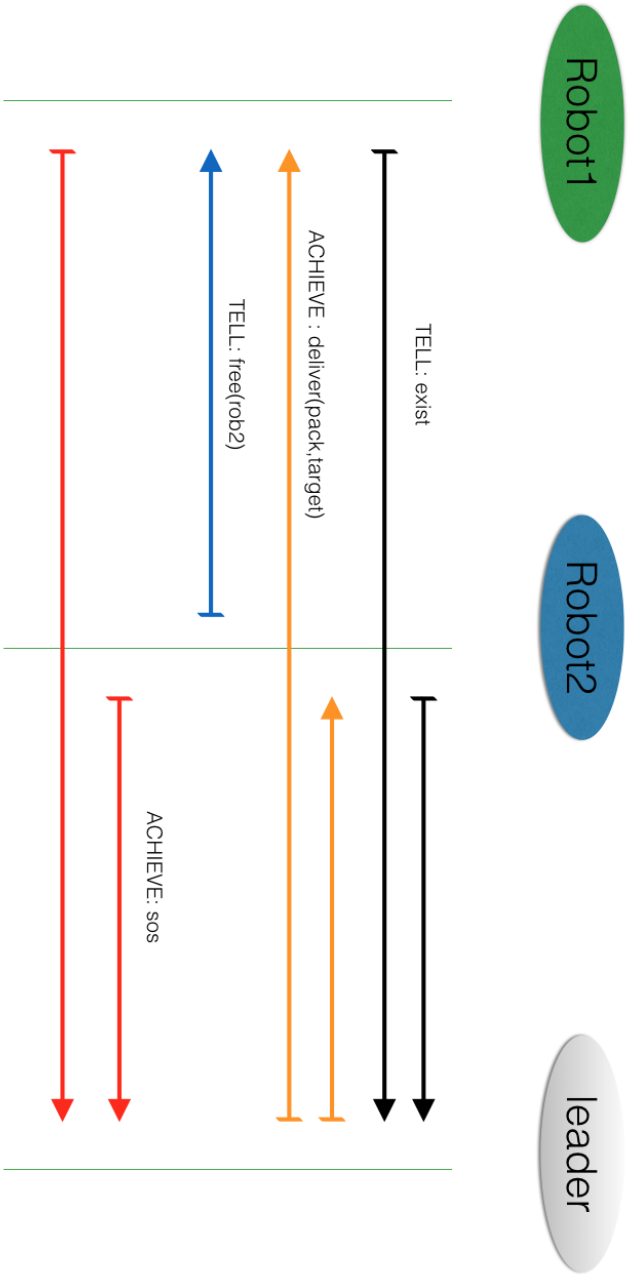

Figura 3.3: Modello d'interazione dell'applicazione

3.2.4 Progetto

Entrando ora nello specifico dell'implementazione, si tratterà più nel dettaglio il codice AgentSpeak nonché velocemente i metodi principali delle classi Java che compongono l'ambiente del MAS.

Viene qui ripetuto che ci si basa su una prima versione dell'applicazione e che l'applicativo finale potrebbe differire, non nei concetti chiaramente.

Prima di concentrarci sull'AgentSpeak diamo un'occhiata, senza entrare troppo nel dettaglio, al codice che implementa l'ambiente. E' stato utilizzato il pattern Model View Controller, quindi abbiamo tre classi Java ognuna con le proprie responsabilità e caratteristiche.

- amazonModel è il modello, ciò che gestisce la logica del programma e delle azioni eseguibili dai vari agenti. Viene modellato quindi il behaviour degli agenti e l'ambiente astraendo dalla sua realizzazione grafica. Estende il modello a griglia fornito da Jason.

- amazonEnv estende invece la classe Environment di Jason, e funge da ponte fra AgentSpeak e codice Java del modello. Gestisce poi quello che è il ControlPanel, altra classe che permette tramite bottoni GUI di impartire ordini e gestire meglio la simulazione. Non verrà trattata nel dettaglio poiché esula dai nostri interesse, ma è mostrata nelle immagini relative all'esecuzione. In amazonEnv si traducono i literal degli AgentSpeak in metodi Java forniti dalla classe modello, e viceversa.

- amazonView estende la classe view a griglia di Jason e disegna in un JFrame il nostro ambiente nonché i vari agenti. E' stato scelto uno stile sobrio e minimale definendo gli ostacoli con sezione di colore scuro, mentre gli agenti avranno la stessa sagoma ma colori diversi, e inoltre si potrà notare visivamente quando essi porteranno con sé il pacco o meno.

Il lancio dell'applicazione può avvenire da Eclipse tramite plugin o dall'IDE Jason, che non è nient'altro che il noto editor JEdit. Ad ogni modo,

```
public class amazonView extends GridWorldView {

public amazonModel model;
public Font defaultFont;
public JPanel panel;
public Image img;

public amazonView(amazonModel modello) {
super(modello, "AmazonPAir simulation", 800);
model = modello;
defaultFont = new Font("Arial", Font.BOLD, 9);
setVisible(true);
repaint();
}

public void draw(Graphics g, int x, int y, int object) {
        switch(object){
         case amazonModel.START1:
         g.setColor(Color.black);
         drawString(g,x,y,defaultFont,"Start1");
         break;
         case amazonModel.START2:
         g.setColor(Color.black);
         drawString(g,x,y,defaultFont,"Start2");
         break;
         case amazonModel.TARGET1:
```

Figura 3.4: Snippet dalla classe amazonView

```
MAS amazonProva {

infrastructure: Centralised

environment: amazonProva.amazonEnv(2)



agents
robot1;
robot2;
leader;


aslSourcePath:
"src/asl";
}
```

Figura 3.5: Il file .mas2j che lancia l'applicazione

```
@pa +exist(robot1) : true
<-   !envoy(pack,target);
     -exist(robot1)[source(robot1)].

@pb +exist(robot2) : true.

@p1 +!envoy(pack,target): exist(robot2)
<-  .print("Rilevati n. 2 robots disponibili per la consegna");
    .broadcast(achieve,deliver(pack,target));
    -exist(robot2)[source(robot2)].

@p1b +!envoy(pack,target): not exist(robot2)
<-  .wait({+exist(robot2)}, 1000);
```

Figura 3.6: Snippet del codice AgentSpeak dell'agente leader

per lanciare il sistema si utilizza il file .mas2j che va opportunamente modificato: per lo scenario uno vanno settati i giusti agenti (due e non tre) e dato in ingresso il valore intero 1 alla classe amazonView. Mentre per lo scenario due il discorso è chiaramente duale, come si può vedere dall'immagine.

Andando ora a trattare il vero aspetto che ci interessa, analizziamo innanzitutto il codice AgentSpeak dell'agente leader.
L'agente inizialmente non ha beliefs né goals né rules, poiché attende che vengano inviati belief riguardo l'esistenza di un robot postino. Vediamo quindi che a seconda dello scenario (quindi a seconda del fatto che siano arrivati entrambi gli exist o soltanto quello del robot1) diventa applicabile uno dei due relativi. Con il primo notiamo che viene inviato un messaggio con l'internal action .broadcast a entrambi gli agenti, passando loro il goal deliver(pack,target). Da notare che è stato gestito il caso in cui il messaggio di exist di robot2 non arrivi in tempo assieme a quello del robot1 nello scenario due, aspettando tale messaggio per massimo un secondo prima di reputare il robot2 definitivamente non presente. Infine notiamo due plan identici se non per la differenza che ognuno di essi si rirerisce soltanto ad uno dei due agenti, e sono i plan per la stampa (con l'azione interna print) del messaggio di sos per una situazione di fault della batteria degli agenti.

L'immagine, assieme alle contestuali spiegazioni di ogni classe, dovrebbe essere sufficiente per comprendere a pieno l'implementazione. Confidando in ciò, continuiamo con l'analizzare il file .asl del robot1.

Il codice è mostrato in due immagini. Vediamo come l'agente abbia un goal iniziale, ovvero !checkin, che prevede il recupero del proprio nome con l'azione interna myName e l'invio del messaggio di 'exist' già citato al leader. Quando si percepisce un nuovo goal per il deliver, si aggiorna (cosa che verrà fatta ogni volta che si cambierà destinazione) il belief target con start1 e ci si sposta verso start, o meglio, si lancia il subgoal che prevede di trovarci in quella posizione. Una volta che siamo a start1, fatto banale dato che ci si trova già lì inizialmente, si procede con il picking del pacco da trasportare, lanciando il subgoal ensGet, dato che nello scenario uno il picking è legato allo pseudo non-determinismo. Verrà controllato tramite il relativo goal e belief got(pack), in maniera ricorsiva, se si possiede il pacco dopo ogni tentativo di prelievo. Quando la percezione sarà positiva, si proseguirà e ci si sposterà con moveTowards verso il prossimo obiettivo, la destinazione target1.

Una volta trovatosi lì, l'agente lascerà il pacco (drop non può fallire, lo assumiamo in questo modo) e lancerà goBack per tornare a start. Come vediamo, gira tutto attorno a goal, subgoal, e ricorsione per assicurarci che il robot percepisca ciò che voleva. Un altro modo poteva essere utilizzare i test goals alla fine di ogni plan per verificarne il giusto esito.

Ad ogni modo le cose si fanno interessanti nella parte finale del codice del robot. Se viene percepito un belief del tipo battery(low) (ovvero quello lanciato in maniera artificiale, ma runtime, quindi per l'agente è del tutto non-deterministico) si reagisce con l'internal action dropIntention, dove fermiamo ogni nostro corso d'azione; recuperiamo poi nome e posizione effettiva (poiché probabilmente ci troveremo in una posizione diversa sia da start1 che target1), concateniamo il risultato per aggregarlo in una variabile che verrà poi passata come parametro per un goal da conseguire per l'agente leader. Si simula quindi un vero e proprio atterraggio d'emergenza.

Infine abbiamo la gestione dello scontro: se viene percepito nelle vicinanze un drone, per la precisione se si percepisce un drone nell'area di collisione del robot (possiamo pensarlo come una sorta di percezione rice-

```
/* Initial goals */

!checkin.

/* Plans */

+!checkin : true
<- .my_name(N);
   .send(leader,tell,exist(N)).

+!deliver(pack,target) : true
<-  +target(start1);
    !at(start1).

+!at(start1) : not got(pack)
<-  !ens_get(pack);
    !at(start1).

+!at(start1) : got(pack)
<-  -+target(target1);
    !at(target1).

+!at(target1) : not at(target1)
<-  move_towards(target1);
    !at(target1).
```

Figura 3.7: Snippet del codice AgentSpeak dell'agente robot1

vuta da dei visori speciali, o da segnali emessi dai droni in volo) si genera un percept di tipo alert; come vediamo viene sospesa l'intention corrente e la si riprende soltanto quando non percepiamo più l'allarme e contestualmente abbiamo il belief free(rob2), belief che viene aggiunto solo e soltanto da un messaggio inviato dal robot2 quando esso uscirà dall'area di collisione.

Il robot2 ha lo stesso codice e gli stessi comportamenti del primo, e il fatto che non sia previsto il non-determinismo nel picking del pacco nello scenario 2 non incide a livello di reasoning ma soltanto a livello di modello (in codice Java quindi). Ciò che cambia è chiaramente la gestione del protocollo per evitare lo scontro, quindi la coordinazione col robot1. Difatti, quando non percepisce più alert, il robot 2 manda un messaggio di tipo TELL con il belief free(rob2), che come abbiamo visto è il segnale che di fatto 'sblocca' il robot 1. Riprenderà poi il suo task verosimilmente prima dell'altro robot evitando quindi lo scontro.

3.2.5 Deployment

Come già ribadito, l'esecuzione avviene tramite Eclipse o IDE Jason (JEdit) selezionando, con le dovute accortezze, lo scenario voluto. Si può controllare la simulazione tramite il control panel mostrato nell'immagine 3.10, affiancato alla GUI che mostra la simulazione vera e propria durante lo scenario due. Passiamo alla conclusione di questo capitolo che conclude il percorso e l'analisi di Jason.

3.2.6 Conclusione

E' già stato detto tutto su questa simulazione, sul fatto che ci si è concentrati su certi aspetti a discapito d'altro, poiché funge da apprendimento e da strumento per mostrare le potenzialità degli agenti in certi campi.

L'apprendimento di certi concetti non è stato certo indolore ma con un approccio graduale, e con molta pratica e studio degli esempi presenti nella distribuzione di Jason, si riesce in poco tempo a produrre codice AgentSpeak funzionante. Con altra pratica si raggiunge un livello dove il codice diverrà ancora più performante ed elegante. Come in tutti i campi, una

```
+!go_back(start2): not at(start2)
<- move_towards(start2);
   -+target(start2);
   !go_back(start2).

+!ens_get(S) : not got(pack)
<- get(S);
   !ens_get(S).

+!ens_get(S) : got(pack) <- true.

+battery(low)[source(percept)]: true
<- .drop_intention(at(_));
   .my_name(N);
   ?pos(r2,X,Y);
   .concat(X," - ",Y,C);
   .send(leader,achieve,sos(N,C)).

-alert:  true
<- .send(robot1,tell,free(rob2)).
```

Figura 3.8: Snippet del codice AgentSpeak dell'agente robot2

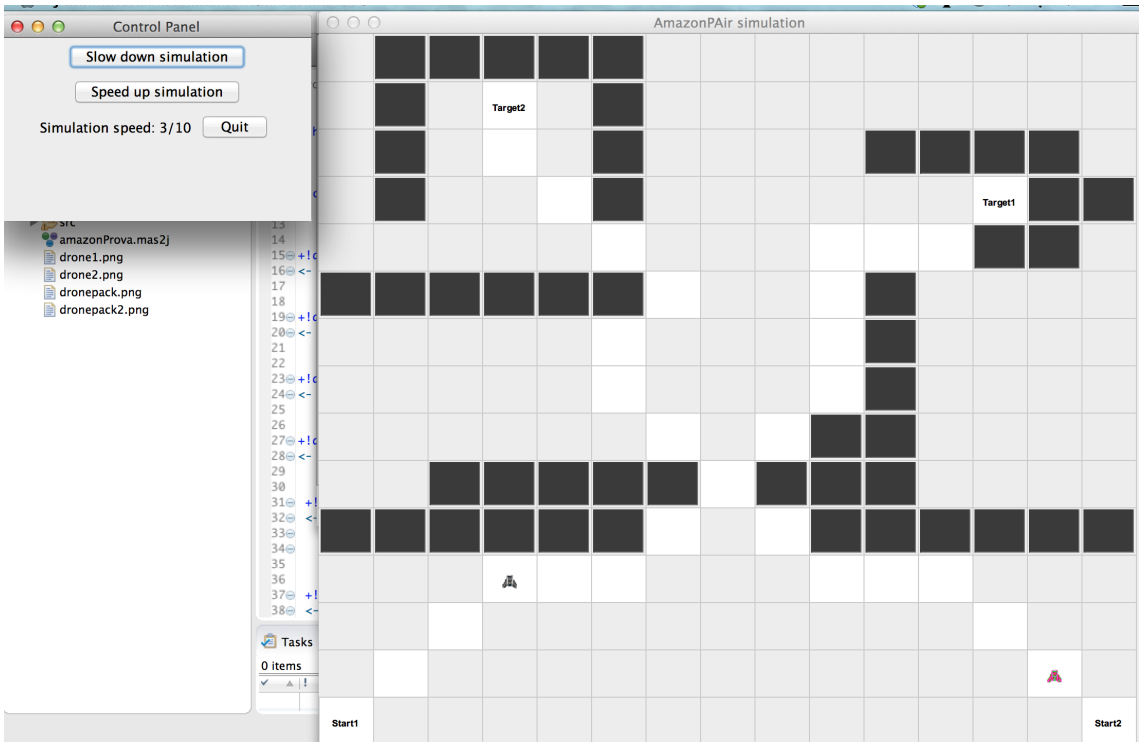

Figura 3.9: L'applicazione durante l'esecuzione

volta compresa a fondo, almeno nei concetti, la teoria, rimane tutta una questione di idee e capacità nel realizzarle. Capacità che miglioreranno, appunto, soltanto con la pratica e l'applicazione allo studio.

Capitolo 4

Conclusione

4.1 Considerazioni finali

Giunti al termine di questo percorso vorrei ricollegarmi alla premessa iniziale presentata nell'Introduzione, per me di notevole importanza; auspico che chiunque abbia letto e cercato di capire, con questo documento, la programmazione ad agenti, sia almeno riuscito a comprenderne i concetti cardine e le visioni, nonchè la sua importanza nell'informatica del domani. Non è importante l'esser già capaci o meno di produrre codice AgentSpeak ben scritto o saper eseguire un MAS in Jason, bensì l'aver compreso le differenze principali tra programmazione ad agenti e ad oggetti, e le peculiarità della programmazione logica.

Mi riterrò soddisfatto anche soltanto nel sapere che il lettore, dopo una lettura più o meno approfondita, sia diventato più interessato all'argomento ed abbia acquisito un pò di quella passione che si è cercato di trasmettere nella scrittura. Il mondo della programmazione ad agenti difatti, si è visto, oltre ad essere un settore trainante e d'importanza sempre crescente, è inevitabilmente affascinante.

Costruire un sistema dove i partecipanti, dopo semplici istruzioni a design-time ed assieme ad un environment sapientemente descritto, sapranno poi muoversi autonomamente, in maniera reattiva e pro-attiva, è appagante. E come si è visto, non è nulla di trascendentale una volta acquisiti i concetti base.

Se si è in generale appassionati di programmazione, ad oggetti o che sia, e si ha un approccio all'avanguardia per cui non si vuole mai rimanere

indietro con le nuove tecnologie, gli agenti sono ciò sui cui puntare. Non sarà una scommessa, ma una certezza. Non va però mai fatta confusione con ciò che rappresentano gli agenti: un paradigma di programmazione, niente più. Non potremo mai programmare sistemi intelligenti all'avanguardia, o perlomeno non oltre i limiti che l'intelligenza artificiale ci ha posto in questo determinato momento. In fondo, per quanto potente ed espressivo, stiamo soltanto modellando e producendo software, ad alto livello.

Spero quindi di aver trasmesso, oltre all'importanza degli agenti, anche la bellezza della loro espressività e flessibilità, per far sì che sempre più appassionati e studenti del settore si interessino all'argomento e arricchiscano la propria cultura informatica.

4.2 Ringraziamenti

Realizzare questa tesi è stato, in tutta sincerità, un impegno che è diventato un piacere e che ha sicuramente ampliato le mie vedute nel campo trattato, nonchè in generale nell'importante approccio ai documenti scientifici a livello professionale. Ma dopo tutto, è soltanto l'ultimo capitolo di quella che è stata un'esperienza durata tre anni, durante la quale sono cresciuto sotto ogni aspetto culturale e umano.

Non sarei mai riuscito a concludere gli studi senza il supporto e la presenza dei miei compagni di corso, ma è riduttivo definirli in questo modo dato che sono amici con cui si è vissuto tre anni ogni tipo di esperienza, d'ateneo e non. Ringrazio tutti, senza fare nomi, coloro che hanno vissuto con me anni di viaggi da pendolare, lezioni, esami, ansie, coloro che mi hanno ospitato senza mai rifiutarmi un aiuto e che in generale ci sono sempre stati.

Ringrazio, senza entrare nei dettagli (per non sembrare magari ruffiano), quei professori che tramite i loro insegnamenti e la loro passione hanno trasmesso in me la stessa voglia di imparare che essi hanno, e nel crescere sempre verso quella bellissima scienza che è l'informatica. Ringrazio quindi tutte le persone che in qualche modo hanno contribuito alla conclusione di questo percorso (triennale perlomeno) a Cesena.

Infine ringrazio di cuore la mia famiglia, che mi ha sempre sostenuto senza alcun ripensamento nè dubbio; che mi ha permesso di completare gli

studi con tutto il supporto possibile, economico e soprattutto motivazionale, lasciandomi tutto il tempo e lo spazio per studiare senza dover lavorare o dover portar pensiero ad altro. Non sarei mai lontanamente riuscito a farcela senza di loro.

Spero che anch'io abbia dato il mio contributo, almeno in piccola parte, a tutti coloro che mi hanno aiutato, e che sia riuscito a far capire la mia gratitudine, poichè ciò avrebbe molto più valore per me di questi semplici ringraziamenti su carta stampata.

Nicolò

Capitolo 5

Bibliografia

- Rafael H.Bordini, Jomi Fred Hubner, Michael Wooldridge - programming multi-agent systems in AgentSpeak using Jason - Wiley, 2007.
- Antonio Natali, Ambra Molesini - Costruire sistemi software: dai modelli al codice. II edizione - Esculapio-Progetto Leonardo, 2009.
- Michael Wooldridge - An Introduction to MultiAgent-Systems - Second Edition - John Wiley and Sons, 2009
- Jason's website - http://jason.sourceforge.net, 2014
- JADE's website - http://jade.tilab.com/, 2014