

ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Ingegneria Informatica Specialistica

RUOLO E UTILIZZO DI PROLOG NELLO
SVILUPPO DI NUOVI LINGUAGGI DI
PROGRAMMAZIONE AGENT-ORIENTED

Elaborato nel corso di: Sistemi Multi-Agente

Relazione di:
VALERIO MONTI

Prof.:
Prof. ANDREA OMICINI

ANNO ACCADEMICO 2010–2011

Indice

Introduzione	v
1 Prolog nei MAS: come e perché	1
1.1 Introduzione ai MAS	1
1.2 Programmazione dichiarativa e logica nei MAS	2
1.3 Perché (non solo) Prolog?	4
2 Prolog e MAS: esempi di nuovi linguaggi	7
2.1 JavaLog	7
2.1.1 Introduzione a JavaLog	7
2.1.2 Il ruolo di Prolog in JavaLog	9
2.2 ALBA	9
2.2.1 Introduzione a ALBA	9
2.2.2 Il ruolo di Prolog in ALBA	11
2.3 Qu-Prolog	11
2.3.1 Introduzione a Qu-Prolog	11
2.3.2 Il ruolo di Prolog in Qu-Prolog	12
2.4 3-APL	13
2.4.1 Introduzione a 3-APL	13
2.4.2 Il ruolo di Prolog in 3-APL	16
3 Conclusioni	17

Introduzione

Si parla di agenti e sistemi multi-agente (MAS) da più di un decennio, ma, nonostante siano stati analizzati e definiti i concetti che porterebbero allo shift di paradigma, ancora non ci sono linguaggi agent-oriented mainstream per la loro diffusione. Ciò deriva dalla complessità che inevitabilmente portano le nuove astrazioni nello sviluppo dei sistemi, ma anche dei linguaggi stessi. Uno dei dettagli emersi quasi contemporaneamente al concetto di agente, in quanto entità computazionale autonoma (con facoltà cognitive, nel caso di accezione forte di agente), è l'impossibilità pratica di mappare la nuova astrazione sui vecchi linguaggi o piattaforme object-oriented. Invece pare evidente l'utilità di un approccio dichiarativo/logico al problema, totalmente inevitabile nel caso di agenti intelligenti. Risulta dunque inevitabile l'utilizzo di linguaggi dichiarativi (logici) per lo sviluppo di nuovi linguaggi agent-oriented, anche se, come verrà spiegato nel primo capitolo, l'approccio unicamente logic-based è evidentemente insufficiente. Con questa relazione mi propongo di approfondire il discorso appena sviluppato e soprattutto il ruolo di Prolog (e delle sue derivazioni) nello sviluppo di linguaggi agent-oriented dedicati allo sviluppo dei MAS, facendo riferimento in particolare a qualche esempio pratico di linguaggi come Javalog, ALBA, Qu-prolog e go!.

Capitolo 1

Prolog nei MAS: come e perché

In questo capitolo si introduce l'accostamento dei MAS all'approccio dichiarativo e della loro inscindibilità. Si presenta anche l'utilità del Prolog come linguaggio logico per attività cognitive e non solo. Infine si spiega come i nuovi linguaggi agent-oriented non possano essere sviluppati a partire solamente da Prolog o da un linguaggio object-oriented, ma di come necessitino di entrambe le componenti, dichiarative e non.

1.1 Introduzione ai MAS

L'aumento di complessità, sempre crescente nell'ingegneria del software, ha portato nell'ultimo decennio a cercare una soluzione in nuovi sistemi dotati di certe caratteristiche (autonomia, mobilità, openness, ...)[6], e la stessa evoluzione nella programmazione si è diretta verso il paradigma degli agenti. Un MAS (Multi-Agent System) è un sistema computazionale dove più agenti cooperano o competono con altri per raggiungere goal (o task) individuali e collettivi. L'astrazione fondamentale che porta ai MAS è dunque quella dell'agente, ovvero un'entità computazionale autonoma, dotato di diverse capacità anche a seconda della definizione 'weak' o 'strong' che si vuole adottare. Nel secondo caso in particolare, e nell'accezione più comune, si fa riferimento a caratteristiche e nozioni mentali dell'agente, come belief,

desire, intention, knowledge. Gli agenti definiti ‘forti’ sono sostanzialmente gli agenti (autonomi) intelligenti.

Nei MAS gli agenti vivono, comunicano, cooperano e si coordinano immersi in un Environment, nel quale è compreso l’intero sistema e con il quale gli agenti stessi possono interagire. Nel paradigma A&A (Agents & Artefacts) è presente inoltre una terza astrazione, ovvero gli artefatti, entità passive le quali non incapsulano il controllo (come gli agenti) ma fungono da intermediari sia comunicativi che cognitivi. Tuttavia, fra queste tre astrazioni, l’unica (pro)attiva è quella dell’agente, i veri abitanti dell’Environment (spesso gli artefatti contenuti nell’Environment sono visibili come rappresentazioni dello stesso a beneficio degli agenti).

Analizzando i MAS, sorgono dunque dubbi e problemi da risolvere, i quali accrescono (anziché diminuire) la complessità, ma questa è praticamente una condizione costante per ogni nuova teoria o tecnologia ed uno dei motivi per cui ancora non si può parlare di linguaggi agent-oriented mainstream. Oltre alla cosiddetta intelligenza, gli agenti hanno altre caratteristiche derivanti dall’autonomia all’interno dell’Environment del MAS, quali la reattività e (pro)attività, l’abilità sociale, la situabilità, la possibilità di comunicare con altri agenti e con l’Environment stesso (tramite artefatti o meno).

Tutte queste caratteristiche hanno rese evidenti la limitatezza e povertà dei linguaggi object-oriented per descrivere questa nuova astrazione, anche se in particolare è l’aspetto intelligente degli agenti (ed il modello BDI) a far muovere nella direzione dei linguaggi dichiarativi (ed in particolare logici). Tuttavia ciò non vuol dire che non si farà uso di linguaggi object-oriented mainstream (come Java) per la costruzione di nuovi framework per i nuovi linguaggi agent-oriented.

1.2 Programmazione dichiarativa e logica nei MAS

Ormai, distinguere i MAS dai sistemi di agenti dichiarativi è un’impresa ardua[4] e questi ultimi, anziché essere trattati come un sottoproblema, dovrebbero confluire e confondersi con la più generale

definizione di Multi-Agent System. Questo poichè l'approccio dichiarativo, linguistico e tecnologico, ai MAS è essenziale e non accantonabile. Adottando una nozione di agenti 'debole' o basata unicamente sull'autonomia, le tecniche dichiarative non appaiono più come strettamente necessarie, eppure anche in questi casi sono estremamente utili (se non indispensabili) ogni qualvolta che si progetta un MAS non banale o triviale, data l'astrazione di alto livello che promuove il sistema ad agenti.

La ragione del successo della programmazione dichiarativa applicata agli agenti è data dal fatto che questo tipo di approccio è particolarmente adeguato per gestire la complessità dei MAS ed i suoi paradigmi, ai quali è intuitivamente legata. Le caratteristiche derivanti dall'autonomia degli agenti vanno ad influire i sistemi multi-agenti dando loro caratteristiche essenziali, ma non così semplici da governare: i MAS sono dinamici (gli agenti possono entrare ed uscire dal sistema ed essere modificati a runtime), le interazioni (fra agenti) sono eterogenee e gli agenti potrebbero necessitare di meccanismi di cooperazione. I linguaggi dichiarativi astraggono dal meccanismo di esecuzione e permettono lo studio di una soluzione e delle sue proprietà a livello concettuale. Inoltre alcuni concetti introdotti dai linguaggi dichiarativi sono sempre più diffusi in nuovi linguaggi e sistemi, alcuni dei quali essenziali anche per la costruzione di applicazioni agent-oriented (assertions, pattern-matching, etc.).

Assumendo i MAS come sistemi complessi, è inoltre possibile suddividerli in tre livelli (mechanism, agent, social, in crescendo di complessità) nei quali modellare i nuovi sistemi multi-agente[10], ma la cosa che interessa in questo contesto è la visione dello shift paradigmatico evidenziato da questi tre livelli e soprattutto dell'agente come entità trasversale che attraversa tutti e tre i livelli. Si vuole rimarcare anche come questa visione dei MAS evidenzia l'impossibilità di un approccio puramente dichiarativo come allo stesso tempo puramente object-oriented. Al livello-agente, dove vengono incarnate le logiche BDI, ovviamente funziona meglio un approccio logic-based che, come evidenziato anche da altri lavori [3], non solo aiuta a collegare i livelli di meccanismo e dell'agente, ma abilita anche quest'ultimo a incapsulare astrazioni per il livello successivo, quello sociale.

Quindi pare evidente come l'approccio dichiarativo e l'utilizzo di lin-

guaggi di questo tipo sia pressoché inevitabile nello studio e nella progettazione di nuovi MAS e linguaggi agent-oriented. Da notare inoltre come questo nuovo tipo di progettazione idealmente, definiti i goals, lasci che il meccanismo di controllo (l'agente) capisca cosa fare per raggiungerli, sviluppando un proprio modello di comportamento in base a beliefs e desires, in modo autonomo, senza che il programmatore debba stabilire come l'agente debba agire per realizzare il proprio obiettivo. Per questo motivo in teoria la programmazione agent-oriented potrebbe addirittura essere considerata come 'post-dichiarativa'.

1.3 Perchè (non solo) Prolog?

Oltre alla visione generale del paragrafo precedente, dove appare evidente la necessità di un approccio dichiarativo ai MAS, l'utilizzo di linguaggi logici è una conseguenza della richiesta di atteggiamenti mentali degli agenti, per questa ragione Prolog può essere considerato una scelta ovvia per rappresentare ed inferire le relazioni fra entità mentali come intentions, goals e beliefs, essendo probabilmente il linguaggio di programmazione logica più diffuso, nelle sue varie versioni[5]. La computazione in Prolog è basata sulla logica e sulla conoscenza, sviluppata e testata da predicati tramite il pattern-matching; queste caratteristiche lo rendono il primo candidato parlando di abilità cognitive in sistemi software (infatti è possibile usarlo anche per artefatti cognitivi, oltre che per le caratteristiche mentali degli agenti). Discutendo di agenti intelligenti è quindi quasi impossibile non nominarlo. Tuttavia, se ormai pare ovvio l'utilizzo di un linguaggio che così bene si adatta ad alcune esigenze degli agenti, è altrettanto evidente che parlare solo di Prolog non avrebbe molto senso. L'evoluzione dei MAS in strutture sempre più elaborate ha dato sempre più spazio per approcci logic-based, ma allo stesso tempo i modelli dichiarativi o logici sviluppati negli ultimi dieci anni sono così vasti da non poter essere coperti da un unico linguaggio. Inoltre, linguaggi logici e dichiarativi presentano molte limitazioni per la definizione e l'utilizzo di abilità che linguaggi object-oriented supportano naturalmente, e che solitamente i linguaggi per agenti (anche logic-based) devono trattare, come anche

caratteristiche non dichiarative ed inerenti ad altri tipi di linguaggio (oggetti, thread, etc.). Esistono alcuni linguaggi a multi-paradigmi, integranti linguaggi logici e object-oriented, tuttavia la sfida è sempre trovare il giusto bilanciamento fra potere espressivo, supporto tecnologico e funzionalità.

Riassumendo, gli attuali linguaggi ad oggetti mainstream definiscono implicitamente un livello troppo povero per i MAS, lo stesso Prolog fornisce un supporto molto più espressivo per astrazioni e modelli agent-based. Tuttavia Prolog da solo non può incarnare un sistema multi-agente, non può neanche incarnare un singolo agente. Un'integrazione di linguaggi object-oriented e dichiarativi (logic-based) è la soluzione che ad ora sembra la migliore, e gli esempi portati nel capitolo successivo sono qui a ribadire questa tesi.

Capitolo 2

Prolog e MAS: esempi di nuovi linguaggi

In questo capitolo si esaminano alcuni esempi tecnici di linguaggi agent-oriented sviluppati nell'ultimo decennio, nei quali l'utilizzo di Prolog è fondante e necessario, analizzandone il ruolo all'interno del sistema in esame.

2.1 JavaLog

2.1.1 Introduzione a JavaLog

JavaLog si presenta come un framework object-oriented atto all'integrazione di paradimi logici e object-oriented[1] in modo che le nuove proprietà del linguaggio siano aggiungibili in maniera incrementale, senza toccare il cuore del linguaggio. Quest'ultimo è basato su moduli logici integrati ad astrazioni legate al paradigma ad oggetti. In sostanza il framework di JavaLog si occupa di integrare Java e Prolog, in una sua istanza basilare, per poi offrire la possibilità di essere esteso con il supporto a multi-threading, mobilità e operatori temporali-logici per Prolog. Queste sono caratteristiche opzionali per alcuni MAS e sono fornite da estensioni del framework successive a quella di base qui presentata.

Avendo dato una certa importanza all'estensibilità della piattaforma di partenza è evidente come il ruolo di Java sia fondamentale per le

proprie caratteristiche date dal paradigma ad oggetti e difficilmente sfruttabili dal solo Prolog.

Facendo considerazioni simili a quelle del capitolo precedente (1.3), gli sviluppatori di JavaLog hanno dunque progettato questo linguaggio come integrazione di una parte ad oggetti (rappresentata da Java), per la parte più strutturale e per rendere il framework estendibile in maniera incrementale, creando quindi una famiglia di linguaggi, ed una logica (rappresentata da Prolog), essenziale per la definizione delle caratteristiche mentali degli agenti, incapsulate in moduli logici. Questi ultimi sono i componenti basilari per il linguaggio e per l'integrazione dei due paradigmi, fra cui Java fornisce la modularizzazione e l'incapsulamento come concetti fondamentali per la struttura del linguaggio, mentre Prolog fornisce il proprio meccanismo logico per la definizione e l'interrogazione delle clausole definite all'interno dei moduli logici.

Un agente JavaLog può essere formato da due semplici oggetti Java, uno dei quali chiamato 'brain', dentro il quale si occupa del ragionamento dell'agente stesso e dei suoi processi mentali. L'approccio di JavaLog permette alle classi di definire moduli logici tramite metodi, il che abilita le classi a registrare fatti e regole in moduli logici rappresentanti la conoscenza dell'agente incarnato da quella classe. Questi moduli logici definiscono gli atteggiamenti mentali di ogni agente. L'integrazione fra i due linguaggi avviene dunque in tre modi: 'per riferimento' ovvero i moduli logici possono essere localizzati in instance variables o come parti di un metodo, la definizione di un modulo logico tramite metodi Java permette agli sviluppatori di agenti di definire atteggiamenti mentali comuni a più agenti della stessa classe; 'per valore' ovvero gli oggetti possono diventare fatti Prolog come parte del modulo logico, le clausole stesse possono utilizzare gli oggetti come termini; 'per composizione' ovvero come i moduli logici possono ricombinarsi, esprimendo la composizione di una base di conoscenza.

I moduli logici sono dunque il concetto fondamentale su cui è basato JavaLog e sono rappresentati da una classe Java *LogicModule*. Ogni modulo logico è in realtà una sequenza di clausole Prolog. Questo costruito è esteso per supportare caratteristiche extra rispetto alla versione base del framework, come moduli con operatori temporali (*TemporalLogicModule*), clausole contenenti oggetti Java (*PlJavaObj*), ed altre. In modo simile, il cuore dell'interprete JavaLog può

essere esteso per supportare nuove abilità del linguaggio come la concorrenza, la mobilità forte e l'integrazione con i Web servers.

2.1.2 Il ruolo di Prolog in JavaLog

Il paradigma ad oggetti è dunque la base di questo framework per linguaggi agent-oriented, sfruttando le caratteristiche di Java come linguaggio object-oriented, quali la modularizzazione e l'incapsulamento, per la struttura della piattaforma di base, oltre che per renderlo estensibile in maniera incrementale.

Tuttavia gli oggetti non hanno la possibilità di gestire gli atteggiamenti mentali rappresentati da clausole naturalmente, quindi utilizzando un interprete di un linguaggio logico integrato negli oggetti Java si trova una soluzione per programmare gli agenti. Un'istanza dell'interprete è contenuta nell'oggetto già chiamato 'brain' nella precedente sezione (2.1.1) ed è responsabile della gestione dei moduli logici; in questo caso è chiaramente un'istanza del motore Prolog, od eventualmente di una sua estensione. Infine, questi oggetti tramutati in agenti grazie alla potenzialità di un linguaggio logico come Prolog, utilizzato per incarnare le caratteristiche mentali degli agenti stessi. In questo modo gli agenti possono ottenere atteggiamenti mentali espressi in forma logica da regole e fatti, manipolati però attraverso metodi della principale classe dell'agente.

2.2 ALBA

2.2.1 Introduzione a ALBA

ALBA è una libreria dedicata per agenti scritti in Prolog [2], che cerca di rendere possibile lo sviluppo di una piattaforma robusta e funzionale per agenti distribuiti e mobili. Spesso (se non sempre) la migrazione degli agenti nelle piattaforme viste e proposte avviene tramite entità dedicate che sono a conoscenza degli agenti locali e remoti, ALBA si prefigge di ottenere una piattaforma decentralizzata che si adatti dinamicamente ed in maniera piuttosto libera rispetto al contesto, in un

ambiente distribuito.

ALBA si presenta come una libreria per Prolog ed è più facile da descrivere sotto questa luce, più che con il nome di piattaforma; le tre componenti principali (decentralizzazione, genericità, flessibilità) ovviamente generano diversi problemi, specie a livello coordinativo e di comunicazione, eppure in questo caso vogliamo porre l'accento sull'ancora una volta fondamentale utilizzo di Prolog per la progettazione e lo sviluppo di agenti per MAS.

Un agente ALBA è costituito di due parti, la libreria embedded che si occupa di tutte le componenti basilari (messaging, contacts, etc.) ed il comportamento dell'agente stesso codificato in Prolog. Ogni agente è un processo Prolog indipendente ed ha un nome univoco che lo identifica all'interno del sistema. L'autonomia ha un ruolo fondamentale nel concetto di agente, quindi si cerca di mostrare come ALBA possa funzionare ed essere utilizzata con un modello specifico di agente chiamato Reasoning Thread (RT), il quale agisce e decide in totale autonomia. Un agente non può predire l'esatto comportamento degli altri agenti, ma può delimitare l'insieme di comportamenti che si aspetta. Di conseguenza, gli agenti pianificano i propri movimenti in corrispondenza alle possibili azioni/reazioni del sistema e delle sue componenti. Ogni RT può essere descritto come una macchina a stati finiti estesa, rappresentante una conoscenza procedurale associata al contesto, un RT consiste quindi in un set di stati. Gli RTs sono istanziati e distrutti dinamicamente, quando un RT è istanziato una memoria locale è creata per conservare i dati rilevanti del contesto. Segnali interni permettono la comunicazione fra vari RTs, il che può servire anche per risolvere conflitti fra vari contesti; ogni regola (rule) specifica un'azione (action) che sarà eseguita, se selezionata, allo stato successivo dell'RT. Un'azione consiste in codice Prolog includente chiamate per incapsulare routine di ALBA (invio di messaggi, modifiche o invocazioni di RT, aggiornamenti della memoria,...). Ogni regola è scritta come una regola Prolog. I piani (plans) possono essere incompleti o inaccurati o la conoscenza (knowledge) incompleta e quindi da estendere o correggere a runtime, per questo gli agenti devono essere capaci di estendere e modificare le proprie caratteristiche mentali dinamicamente.

2.2.2 Il ruolo di Prolog in ALBA

Come chiariscono gli stessi creatori [2], Prolog (in quanto linguaggio logico) ha diverse caratteristiche utili, se non vitali, per la progettazione di agenti intelligenti e più in generale di intelligenze artificiali: come i meccanismi di unificazione e risoluzione, l'efficienza nella gestione di strutture ad albero. Come linguaggio dichiarativo si avvantaggia dell'espressività della logica di prim'ordine, ed ancora una volta è il primo candidato per gestire gli agenti.

Oltre a ciò, anche se non si vede molto in ALBA, la quale propone solo un modello di agenti, sviluppando in realtà solo la libreria per renderli poi utilizzabili in maniera mobile e flessibile, l'utilizzo di Prolog si fa sempre più necessario nelle parti cognitive e mentali dell'agente definito intelligente.

Da chiarire che, come già detto altrove anche in questa relazione, Prolog non è adatto ad ogni compito nell'impersonificare gli agenti, ma ALBA non tratta di questa parte, dando per scontata l'integrazione con linguaggi object-oriented, ad oggi sicuramente fattibile e non messa in discussione.

2.3 Qu-Prolog

2.3.1 Introduzione a Qu-Prolog

Qu-Prolog è una versione di Prolog multi-thread progettato specificamente per lo sviluppo di applicazioni agent-oriented distribuite [7], dove gli agenti stessi possono essere multi-threaded. Inizia come variante di Prolog per risolutori di teoremi, per divenire un linguaggio agent-oriented con l'aggiunta della comunicazione inter e intra-thread. L'ultimo passaggio è stato collegare Qu-Prolog con il context-based message router Elvin per ottenere la comunicazione broadcast, il che fornisce anche un'interfaccia per i messaggi per le applicazioni scritte con linguaggi imperativi come Java, grazie alle numerose APIs di Elvin.

Una delle discriminanti principali fra oggetti ed agenti è che i secondi hanno e gestiscono il loro flusso di controllo, ma con Qu-Prolog (e non

solo) si suppone che ciascun agente sia multi-threaded; ogni thread è quindi utilizzato per incapsulare uno dei componenti comportamentali più importanti dell'agente. Per esempio si avrà un thread per ogni altro agente con cui si sta comunicando, quindi ogni thread di conversazione avrà accesso e potrà aggiornare un base di belief condivisa. Ovviamente, la cosa va gestita per evitare inconsistenze ed ogni thread potrà essere sospeso in attesa di un aggiornamento della base da parte di altri thread. Si avrà quindi un altro thread interno dell'agente per coordinare il database di clausole dinamiche.

I linguaggi di comunicazione per agenti come KQML e FIPA ACL assumono che la comunicazione fra agenti sia asincrona, in Qu-Prolog si ottiene ciò tramite il modello di comunicazione thread to thread. Ogni thread ha un buffer di messaggi ed un nome univoco, i messaggi mandati ad un thread interno sono copiati nel buffer del thread di destinazione.

Per controllare i thread sono presenti appositi predicati come *thread_fork(Name, Goal)* o *thread_exit(Name)*. Nel primo caso il goal è creato assieme al thread che cerca di raggiungerlo, il quale terminerà da solo al compimento del proprio scopo appunto. Per gestire inoltre delle computazioni con precedenza si può utilizzare il predicato *thread_atomic_goal(Goal)* che si assicura che nessun altro thread ottenga una time slice mentre viene eseguita quell'azione, considerata quindi come atomica.

2.3.2 Il ruolo di Prolog in Qu-Prolog

Si pone l'accento in questo caso sulla comunicazione e soprattutto sull'interazione fra i thread inter-agent ed intra-agent, dato che il linguaggio stesso è interamente logic-based, essendo un'evoluzione del Prolog stesso. Ogni parte dell'agente è governata da procedure e fatti tipicamente logic-based, in maniera precisa e sintetica. Tuttavia, come citato nella sezione (1.3.1) vi è modo di comunicare con gli agenti Qu-Prolog tramite linguaggi object-oriented grazie ad interfacce fornite dal server Elvin.

2.4 3-APL

2.4.1 Introduzione a 3-APL

3-APL (Triple APL) è un linguaggio di programmazione (e piattaforma) agent-oriented[8] [9], creato per lo sviluppo di agenti intelligenti. L'intelligenza negli agenti è fornita da complessi stati mentali, consistenti in atteggiamenti e caratteristiche mentali e processi che forniscono un meccanismo di transizione fra uno stato mentale e l'altro. Gli agenti sono inoltre capaci di interagire fra di loro, direttamente comunicando, o indirettamente attraverso un ambiente condiviso. La piattaforma di 3-APL fornisce la possibilità di gestire più agenti allo stesso tempo nelle loro attività oltre che nella loro comunicazione.

Il componente principale di un agente in 3-APL sono le sue caratteristiche mentali, incarnate da strutture dati rappresentanti beliefs, goals, plans, actions e rules. Si può quindi dire che un agente ha uno scopo (goals) che lo stimola ad agire od interagire con l'ambiente (environment); per raggiungere il proprio scopo pianifica (plans) le azioni (actions) richieste; ha regole di ragionamento pratico (rules) che gli permettono di modificare i propri piani in caso di necessità; ed ha anche conoscenze, o meglio convinzioni, (beliefs) a proposito di sé stesso e dell'ambiente che gli consentono di pianificare le azioni e di valutare se il proprio scopo sarà raggiunto. La relazione fra gli atteggiamenti mentali e la logica BDI è definita come segue:

Belief (BDI) -*i* Beliefs (3APL)

Desires (BDI) -*i* Goal (3APL, dichiarativi)

Intentions (BDI) -*i* Plans (3APL, procedurali)

Il deliberation process utilizza i beliefs ed i goals per decidere che piano d'azione seguire, o decide quando il piano corrente necessita di essere rivisto o rimosso.

Un agente necessita di beliefs al riguardo di sé stesso o dell'ambiente circostante; un belief è rappresentato da una formula Prolog. Il set dei beliefs di un agente è chiamato Belief Base, la quale è basata su un modello di mondo chiuso, ovvero se un belief non è presente nella base allora è falso.

Le azioni dell'agente rappresentano la propria capacità di modificare i propri stati mentali o quelli dell'ambiente, queste azioni sono l'unico

costruito in 3-APL in grado di modificare la Belief Base. Ci sono tre tipi basilari di actions: mentali, per la comunicazioni ed esterne. Le prime sostanzialmente si occupano di aggiornare la belief base, come già detto, e sono specificate come queries per la base, e riguardano la parte logica del linguaggio di 3-APL. Le action per la comunicazione invece si occupano di mandare messaggi ad altri agenti o alla piattaforma stessa. Un messaggio contiene il nome del ricevente, il tipo di messaggio (inform, request, etc.) ed il contenuto, sotto forma di predicato. Le actions esterne invece si occupano di ciò che sta al di fuori dell'agente stesso, ovvero l'ambiente. Da notare che il reale effetto delle proprie azioni potrebbe essere sconosciuto all'agente e potrebbe essere necessaria un'altra action esterna per percepire le modifiche avvenute, in ogni caso si può assumere che in generale l'agente sa il risultato esatto delle proprie actions esterne ed aggiorna la Belief Base di conseguenza. Le actions esterne in 3-APL sono metodi delle classi Java che rappresentano l'ambiente, i quali possono essere implementati per ritornare il risultato dell'azione tramite una lista (nel caso questa fosse vuota, sarebbe necessaria un'azione esplicita per percepire le modifiche). I goals rappresentano lo stato delle cose desiderato dall'agente; in 3-APL sono rappresentati come una congiunzione di fatti, rappresentanti le beliefs che si vorrebbero raggiungere. Il set di goals di un agente è chiamato Goal Base. Un agente goal-driven continua la propria esecuzione finché le belief desiderate (quindi il proprio Goal Base) non fanno parte della propria Belief Base.

I plans sono uno strumento per raggiungere i propri goals e possono essere di tre tipi: basici, astratti, composti. I primi possono avere la forma di un'azione base (mentale, comunicativa od esterna) e consistono in una query nella Belief Base per determinare se un dato belief è vero o falso. Un piano astratto è una rappresentazione del plan che può essere istanziata con un piano più concreto durante l'esecuzione. Un piano composto invece è formato da altri plans (di uno dei tre tipi già citati), può essere sequenziale, condizionale, ovvero formato di una serie di scelte dove la condizione è ancora una volta una belief query che sceglie fra due o più piani, oppure iterativo, ovvero dove un piano viene eseguito iterativamente finché il belief desiderato non è vero.

Le regole di ragionamento forniscono all'agente la capacità di costruire o rivedere i plans; una volta che l'agente crede che un piano sia

sufficiente per raggiungere un goal, lo adotterà. La struttura basilare delle regole di ragionamento consiste in tre parti: una ‘head’ che può essere un goal od un plan, un ‘body’ che corrisponde ad un plan, ed una ‘guard’ che consiste in una belief query. Le regole di ragionamento in 3-APL sono di due tipi: goal-plannig-rules, il cui scopo è creare un piano per raggiungere il dato goal, la sua struttura è quella delle regole basilari con l’unico dettaglio che la testa della regola è sempre un goal; e plan-revision-rules, utilizzate per adottare, rivedere o rimuovere un piano, la struttura è la solita solo che la testa della regola è sempre un plan, che sarà rimpiazzato da quello del body.

Il deliberation process è formato da una serie di operazioni di delibera, come regole di esecuzione, selezione di piani, etc. Lo scopo di questo processo è praticamente quello di modificare le caratteristiche mentali dell’agente fino a che non ha raggiunto il proprio goal o completato il proprio piano. Questo processo può anche essere visto come un interprete che determina l’ordine in cui le operazioni devono essere eseguite/performate (potrebbe essere programmato per esempio per rimuovere goals irraggiungibili o per controllare se un goal persiste durante l’esecuzione di un piano). I creatori di 3-APL hanno proposto l’implementazione dell’interprete come un programma di meta-livello che può essere personalizzato dall’utente a seconda delle sue necessità. Sono presenti un set di deliberation operations per le operazioni basilari (SelectPGrule, SelectPlan,...) che possono essere ricombinate con composizioni sequenziali, scelte, test e loop iterativi per creare programmi più complessi.

La piattaforma di 3-APL permette di programmare, sviluppare ed eseguire gli agenti in questo linguaggio; si incarica inoltre del trasporto dei messaggi di comunicazione, fornisce anche informazioni a proposito di altri agenti presenti nel MAS. Infine fornisce un tool di monitoring per lo scambio di messaggio ed una finestra speciale per controllare gli stati mentali degli agenti. L’ambiente è parte formante della piattaforma 3-APL e si presenta sottoforma di una classe Java programmabile, il che fornisce le actions esterne agli agenti. In particolare, l’ambiente è modellato come un plugin per la piattaforma.

2.4.2 Il ruolo di Prolog in 3-APL

Come intuito nella sezione precedente (2.3.1), Prolog è direttamente e largamente usato per le caratteristiche mentali degli agenti in 3-APL. L'intero sistema di Belief-Plan-Goal è supportato dal linguaggio logico grazie alle proprie caratteristiche intrinseche. Ogni belief è rappresentato da una formula Prolog, le quali formano la già citata Belief Base, mentre i goal sono congiunzioni di fatti, quindi anch'essi espressi tramite il linguaggio logico. I plans invece sono composti in una forma di linguaggio imperativo (come evidenziato anche nella corrispondenza con il BDI, nella sezione precedente (2.3.1)), con tanto di cicli nel caso di piani iterativi. La parte logica ritorna nelle inevitabili interrogazioni della base di goal o di belief, da parte dei piani stessi, legata quindi in maniera inestricabile all'intera 'intelligenza' dell'agente.

La struttura anche in questo caso è legata a quella di un linguaggio object-oriented (Java nello specifico), per la difficoltà di un linguaggio logico come Prolog a gestire gli aspetti più strutturali del MAS, per quanto essenziale nella gestione di quelli mentali degli agenti. Sono legate a Java le azioni esterne, come visto nella sezione precedente (2.3.1) e l'intero ambiente, il quale (come si è spiegato nel capitolo precedente (1.1)) ha un ruolo passivo nei MAS e per quanto fondamentale è probabilmente l'unico a non necessitare in nessun caso un ruolo nella programmazione logica (gli artefatti cognitivi per quanto passivi possono sfruttare positivamente un approccio dichiarativo/logico (1.3)). 3-APL probabilmente gestisce in maniera più approfondita l'aspetto mentale degli agenti rispetto a JavaLog, il quale però si propone più come framework, mentre quest'ultimo è prima di tutto un linguaggio agent-oriented che pone l'accento sulle caratteristiche mentali degli agenti.

Capitolo 3

Conclusioni

Ad oggi, l'assenza di una soluzione (ed anche di un modello) chiara ed evidente alle esigenze dei MAS fa sì che non vi sia un linguaggio agent-oriented mainstream od utilizzato su larga scala industriale. Tuttavia è da ormai più di un decennio che in ambito di ricerca si lavora in questa direzione ed i progetti proposti sono numerosi, che siano essi nuovi linguaggi, piattaforme o librerie. Nel secondo capitolo di questa relazione si è cercato di esplorarne qualcuno, cercando di porre l'attenzione su come, anche in diversi casi e rappresentazioni, vi sia in comune l'utilizzo di Prolog come elemento fondante, in particolare per gli agenti intelligenti.

L'aspetto mentale degli agenti è fondamentale per l'utilizzo dei linguaggi dichiarativi, ed in particolare logici, tuttavia non l'unico, in quanto spesso Prolog (in quanto linguaggio logico) è esteremamente utile anche al di fuori dei processi cognitivi, come è stato evidenziato anche nelle varie sottosezioni riguardanti il Prolog all'interno dei vari casi esaminati. Un altro motivo essenziale per l'utilizzo di Prolog per lo studio e lo sviluppo di linguaggi per MAS è l'effecienza del linguaggio nello sviluppo e testing di nuovi modelli e prototipi; in ambienti di ricerca non è da sottovalutare anche questo aspetto. In ogni caso, la gestione del comportamento degli agenti è sempre più in mano a linguaggi logici (di cui Prolog è il primo rappresentante) anche per la chiarezza e la semplicità di espressione, oltre che, ovviamente, per la sua potenza espressiva che rende necessario l'utilizzo di linguaggi simili in quasi ogni ambiente riguardante entità cosiddette agenti.

Da rimarcare però, ancora una volta, come Prolog da solo, anche con le sue estensioni, non possa risolvere tutti gli aspetti presentati dai sempre più complessi MAS, anzi alcune esigenze sono tipicamente non-dichiarative e quindi di difficile gestione tramite un linguaggio logico. Per questo si è visto come spesso i linguaggi più completi siano a multi-paradagmi, ovvero gestiscano l'integrazione di un linguaggio logico ed uno object-oriented per la parte più strutturale (nei due casi esaminati il linguaggio ad oggetti era Java). In quest'ambito vale la pena di citare tuProlog quindi, per una più semplice integrazione Java-Prolog.

Bibliografia

- [1] A. Amandi, M. Campo, and A. Zunino. In *JavaLog: a framework-based integration of Java and Prolog for agent-oriented programming*, Tandil (7000), Buenos Aires, Argentina, 2004. ISISTAN Research Institute.
- [2] A. Amandi, M. Campo, and A. Zunino. In *ALBA: a generic library for programming mobile agents with Prolog*, 2 avenue Gay-Lussac, 78851 Elancourt - France, 2007. Thales Airborne Systems.
- [3] D. Ancona and V. Mascaridi. In *Coo-BDI: Extending the BDI model with cooperativity.*, 2004.
- [4] M. Baldoni, C. Baroglio, V. Mascaridi, A. Omicini, and P. Torroni. In *Agents, Multi-Agent Systems and Declarative Programming: What, When, Where, Why, Who, How?*, 2010.
- [5] I. Bratko. In *Prolog Programming for Artificial Intelligence*. Pearson Education, 2001.
- [6] K. L. Clark, P. J. Robinson, and S. Z. Ambolti. In *Multi-threaded communicating agents in Qu-Prolog*, Dept. of Computing, Imperial College, London, 2006.
- [7] M. M. Dastani, M. B. Riemdsdijk, and J.-J. C. Meyer. In *Programmig multi-agent systems in 3APL*. Springer Science, 2005.
- [8] M. M. Huntbach and G. A. Ringwood. In Springer-Verlang, editor, *Agent-Oriented Programming: from Prolog to guarded definite clauses*, 1999.

- [9] S. A. Napagao, B. Auffarth, and N. S. Ramirez. In *Agent language analysis: 3APL*, 2006.
- [10] A. Omicini and F. Zambonelli. In *MAS as Complex Systems: A View on the Role of Declarative Approaches*, 2004.