Università degli studi di Genova

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Informatica

Anno Accademico 2000/2001

Tesi di Laurea

D-CaseLP: un ambiente distribuito per l'integrazione di Agenti Eterogenei.

Candidato
Riccardo Albertoni

Relatori
Prof. Maurizio Martelli

Dott. Viviana Mascardi

Correlatore
Prof. Maura Cerioli

A Sandro, che mi ha insegnato
l'ottimismo.
A nonna Mariuccia,
che mi insegnò a salire sui ciliegi.

Ringraziamenti

Iniziamo con i ringraziamenti “Tecnici” a coloro che mi hanno aiutato nella realizzazione di questo lavoro.

Ringrazio il professor Maurizio Martelli per la disponibilità che mi ha dimostrato.

La dottoressa Viviana Mascardi, per l’aiuto fornitomi e per aver fatto l’impossibile per esserci nonostante i suoi “improrogabili” impegni da mamma ...

A suo marito che suo malgrado si è più volte trovato nel ruolo di tramite fra me e Viviana.

Stefano Miglia per le preziose discussioni.

Veniamo ora alle persone che in questi anni hanno allietato la mia permanenza al DISI, senza le quali forse oggi non sarei qui a scrivere i ringraziamenti. In ordine più o meno sparso ..

Alle mie due amichette Marina e Stefania un ringraziamento particolare per aver sopportato in questi anni i miei voli pindarici, aver studiato, discusso, litigato, e cucito cappotti cappottini camicette ecc. Non scorderò facilmente (e qui ci sono gli estremi per una causa di risarcimento per danni biologici a Gerry) i pomeriggi, i sabati, e le domeniche mattina persi a cercare di capire se due frasi in uno stesso foglio di appunti fossero li per caso o perché nella testa di qualcuno erano in una qualche relazione...

A Sir, per aver condiviso insieme a noi parte dei pomeriggi di cui sopra, per la sua precisione maniacale che non gli impedisce di sembrare un personaggio uscito da un libro di Lewis Carroll ...

A Paolo (Paolino) e Emanuele (Emma) per la dedizione con la quale riescono a sopportare le mie amichette e svolgere contemporaneamente un lavoro socialmente utile ...

A Monica per essere la giovane marmotta di sempre ed una delle poche persone al quale si puó “dare la schiena” e stare più tranquilli di quando ci si volta le spalle allo specchio.

Al Bradi per essere il testone di sempre, compagno di molte discussioni, innumerevoli pause sul senso dell’esistenza ed una delle poche persone più confuse di me.

Alla Lorella fedele spalla quando le discussioni con il bradipo diventavano impossibili ...

Ad Andrea, per avere almeno provato ad essere un po’ più diplomatico con il resto del mondo e nonostante i risultati non essersi scoraggiato del tutto ... a Daniela per aver redatto il libretto di istruzioni su Andrea in 100 semplici fascicoli, anche se temo che nessuno avrà mai né il tempo né la voglia di leggerli...

A Daniela per la sua risata solare ad Emilio per la sua silenziosa timidezza, ad entrambi per essere una splendida coppia (non si offendano gli altri, è solo l’ennesimo tentativo di conquistare una sedia in prima fila davanti al megaschermo ... lo sapete che altrimenti non ci vedo !!!)

A Milena per le imitazioni dei professori e la perseveranza; a Carla perché non ho mai avuto voglia di ucciderla anzi pensavo che fosse il contrario; Ileana per essere “bastarda dentro” e nasconderlo discretamente bene; a Cristiano per la sua sensibilità, le battute irripetibili ed essersi addossato gran parte dell’esercitazione di ROGA; a Luca e Elisabetta per essere i miei “gemelli”(per loro fortuna) eterozigoti.

Un grazie ed un saluto speciale agli amici della pausa pranzo/caffé che con la loro compagnia hanno saputo rendere piacevoli questi ultimi mesi (in ordine sparso): Alberto, Ponta, Biagio, Olcese, Mister, Depra, Lollo, Luisa, Elena C., Elena B., Giada, Daniela P., Mara ed i suoi ripascimenti, Flavia ...

Ai “colleghi” dello CSITA che hanno fatto di tutto per farmi sentire a mio agio durante le mie “collaborazioni”, in particolare, Barbara per la sua amicizia, la grande spontaneità e tutto ciò che mi ha insegnato; Claudia per la sua gentilezza; Maurizia per tutte le risate e le lezioni di depilazione (un giorno di questi

metterò in pratica tutti i suoi insegnamenti ...); Luisa per la fiducia accordatami; Beppe per i discorsi sulla politica; Fabio per i CD; Domenico per avermi concesso la sua postazione.

Veniamo a chi pur restando al di fuori dell'università ha contribuito al raggiungimento di questo obiettivo incoraggiandomi e sopportando il mio non facile carattere.

I miei genitori per avermi lasciato sempre grande autonomia ed avermi finanziato in questa lunga impresa.

Mio fratello Maurizio, presenza molto silenziosa ma non per questo meno importante.

Maura, la mia "sorella adottiva", per essere la persona che in assoluto conosce meglio il me che ero e per essermi stata accanto quando le cose mi erano meno chiare ...

Alessandra, per avermi regalato a Siena una delle giornate più belle della mia vita ed avermi rassicurato sulle mie vite precedenti ...

I coniugi Binelli per la grande ospitalità e le bellissime gite in montagna.

Gian per il litro di birra a Bellaria ... a pensarci ora vedo ancora tutto girare, Cristiano e Patrizia per la loro importante amicizia. Un ringraziamento anche ad Alessio, Fabio, Leo, Tony, Quikkia, Monte, Valle, Fede e tutti quelli che per lungo tempo ho frequentato nella vecchia compagnia.

Infine ringrazio la prof.ssa Carla Ferretto per avermi sconsigliato l'iscrizione a lettere; chiunque avrà occasione di leggere questa tesi potrà infatti rendersi conto quanto io ignori l'arte della scrittura e quanto prezioso sia stato il suo consiglio...

Indice

Introduzione **1**

1 Il contesto **5**
1.1 Agenti e sistemi multi-agente 5
1.1.1 Definizione di agente 6
1.1.2 Definizione di Sistema Multi-Agente (MAS) 6
1.2 CaseLP 7
1.2.1 La concettualizzazione degli agenti 7
1.2.2 Una gerarchia di astrazione di concetti 9
1.2.3 La metodologia di sviluppo 11
1.3 Il progetto D-CaseLP 14
1.3.1 Assunzioni 14
1.3.2 Obiettivi 18
1.3.3 Linee guida metodologiche 23
1.3.4 Un Function Refinement Tree per D-CaseLP 27
1.3.5 Un'interfaccia astratta per far comunicare gli agenti 31
1.3.6 Dalle specifiche in Ag-UML all'esecuzione in D-CaseLP 32

2 Valutazione di CORBA per realizzare MAS **36**
2.1 CORBA 36
2.1.1 Object Management Architecture (OMA) 37
2.1.2 Terminologia in CORBA 38
2.1.3 La comunicazione tra client e server in CORBA 41
2.1.4 Servizi CORBA 42
2.2 Primo design 43
2.2.1 Funzionalità da fornire nell'immediato 43
2.2.2 Come realizzare la comunicazione tra agenti via CORBA 45
2.2.3 Configurazione degli agenti prototipali per l'esecuzione e simulazione 48
2.2.4 Funzionalità aggiuntive 50
2.2.5 Considerazioni di valutazione 53

2.2.6 Conclusioni 54

3 FIPA **56**
3.1 Introduzione alla FIPA 56
3.1.1 Fattori dai quali scaturisce lo standard 57
3.1.2 Assunti e principi nella produzione FIPA 58
3.1.3 La portata delle specifiche FIPA 59
3.1.4 Organigramma della FIPA 62
3.1.5 Il processo di produzione dello standard 63
3.2 Organizzazione delle specifiche FIPA 65
3.3 Architettura astratta 66
3.3.1 Tipo opaco ed iscrizione ai servizi 67
3.3.2 Cosa specifica l'architettura astratta 68
3.3.3 Directory Services 70
3.3.4 Messaggi e Message Transport Services 71
3.3.5 Esempio di comunicazione 74
3.4 Specifica della gestione degli agenti FIPA 75
3.4.1 Un modello di riferimento 75
3.4.2 Il nome degli agenti 77
3.4.3 Il Directory Facilitator 78
3.4.4 Agent Management System 79
3.4.5 Ciclo di vita degli agenti 81
3.5 Le ontologie 83
3.6 Progetti FIPA 84
3.7 Conclusioni 85

4 Inclusione del framework FIPA in D-CaseLP **88**
4.1 Perché includere le specifiche FIPA 88
4.1.1 Aggiornamento del FRT di D-CaseLP 90
4.1.2 Le platform aderenti alle specifiche FIPA 91
4.2 JADE 92
4.2.1 JADE Run-Time 93
4.2.2 Il modello di sviluppo degli agenti JADE 97
4.3 Implementazione attraveso JADE di D-CaseLP 106
4.3.1 Come realizzare con JADE i punti del FRT 106
4.3.2 Ipotesi per l'utilizzo di JADE in D-CaseLP 110
4.3.3 Gli agenti Ag-Java 112
4.3.4 Gli agenti Ag-Prolog 112
4.3.5 Iter dagli agenti prototipali all'esecuzione usando JADE . 113
4.4 Confronto tra JADE e CORBA 113
4.5 Conclusione 116

5 Agenti logici **117**
5.1 Agenti logici in CaseLP. 117
5.1.1 ProlAg 118
5.1.2 AgentRules 119
5.1.3 CaseLP-MASadl 122
5.1.4 $\mathcal{E}_{hhf}$ 122
5.1.5 HEMASL 123
5.1.6 Il CaseLP Visualizer: un'interfaccia grafica per la simulazione 123
5.1.7 Comandi di shell per l'esecuzione della simulazione . . . 123
5.2 Implementazioni Prolog 127
5.2.1 Tipi di interfacce Prolog-Java 128
5.2.2 tuProlog. 129
5.3 Utilizzo degli agenti logici nel modello di sviluppo D-CaseLP . . 134
5.4 Analisi 138
5.4.1 Obiettivi dell'integrazione 138
5.4.2 $_N$ProlAg: linguaggio d'implementazione 139
5.4.3 Use Case Diagram per l'estensione di JADE 142
5.5 Design dell'estensione 146
5.5.1 La concorrenza degli agenti in JADE 146
5.5.2 Design di agenti logici con tuProlog e JADE 147
5.6 Considerazioni conclusive 152

Conclusione e sviluppi futuri **154**

Introduzione

Negli ultimi anni abbiamo assistito ad una significativa crescita dell'interesse verso le tecnologie informatiche. Fattore determinante di questo fenomeno è l'affermarsi dell'utilizzo della "rete", tanto nei contesti della produzione quanto nella vita di tutti i giorni. Si pensi ad esempio a come l'avvento della rete, in particolare del World Wide Web e della posta elettronica, abbia significativamente cambiato il modo di lavorare e le abitudini di una parte della popolazione che vive nei paesi industrializzati.

Allo stato attuale la "rete" si pone come un'entità passiva: un enorme contenitore di "dati" eterogenei in cui spesso la ricerca delle informazioni desiderate comporta dedicare molto tempo a visionare informazioni che poco hanno a che fare con quelle cercate. Per evitare questo spreco di tempo c'è un crescente interesse a superare la passività della rete fornendo quelli che potremmo definire funzionalità di nuova generazione. Tali funzionalità permetteranno agli utenti di accedere ad informazioni già collezionate secondo i loro interessi. Oggi è possibile usare la rete per fruire di un numero sempre maggiore di servizi, che vanno dall'acquisto di libri ad una prenotazione di un posto a teatro o di un viaggio, ma è ancora abbastanza difficile unire servizi differenti o attuare dei confronti al di fuori di quelli che sono stati già esplicitati dai fornitori di informazione.

Verso una nuova generazione di funzionalità per la rete.

Vi è l'esigenza di superare i limiti attuali e completare quella che è stata la rivoluzione degli ultimi anni che ha portato a ciò che viene definita in termini altisonanti "*new economy*". Per far ciò si stanno seguendo almeno due approcci. Il primo si basa su una strutturazione dei "dati" disponibili in rete attraverso l'utilizzo di metalinguaggi come XML [64]. Il secondo approccio [51] sottointende l'idea di una società virtuale di processi: una rete flessibile di processi software eterogenei ognuno con una propria conoscenza, una propria capacità di prendere decisioni e di cooperare con altri processi per il raggiungimento di obiettivi intermedi comuni.

Le due approcci non sono ortogonali fra di loro: il successo della primo può essere utile agli scopi della secondo.

La tendenza nello sviluppo del software.

Allo stesso tempo ed in parte anche in seguito a questa "esigenza" di dovere accedere a tutto ovunque ci si trovi, la produzione del software coinvolge problematiche sempre più vaste. La tendenza è di fornire un numero sempre maggiore di funzionalità ad un insieme di utenti con conoscenze ed esigenze differenti. Per gestire la crescente complessità dei sistemi software i progettisti si trovano sempre più ad integrare componenti eterogenee già realizzate piuttosto che progettarne di nuove da capo. Inoltre, la varietà di utenti ai quali i software sono destinati costringono a dotare i prodotti dell'intelligenza necessaria ad apparire *user friendly*.

La *tecnologia ad agenti* [35], grazie all'alto livello di astrazione a cui si colloca (quello a livello di cooperazione della *conoscenza* tra sistemi autonomi), risulta particolarmente adatta a progettare applicazioni con le caratteristiche descritte sopra ed comunità virtuali; essa consente infatti di modellare entità che comunicano (dotate quindi di *abilità sociali*), che monitorano l'ambiente in cui si trovano e sono in grado di reagire agli eventi che si verificano (*capacità di reazione*), sono in grado di prendere l'iniziativa se la situazione lo richiede (*proattività*) senza l'intervento umano o di altri agenti (*autonomia*). Società composte da tali entità vengono definite *sistemi multi-agente (MAS)*. Esse contemplano la *distribuzione* degli agenti coinvolti e dei dati su cui operano e *l'integrazione* di software e dati eterogenei. Queste due caratteristiche sono fondamentali per il successo dei sistemi software e sono una delle ragioni per il consenso ricevuto dai MAS sia in campo accademico che industriale.

Nonostante l'interesse e le prospettive che il paradigma ad agenti offre vi sono ancora molti problemi da risolvere prima che tale tecnologia si affermi stabilmente. Innanzi tutto vi è la necessità di individuare metodologie efficaci e strumenti di sviluppo che aiutino la progettazione e lo sviluppo dei sistemi multi-agente in modo che tale tecnologia sia utilizzabile in ambito industriale. In secondo luogo, affinché siano sfruttate appieno le potenzialità del paradigma ad agenti e sia possibile costituire le infrastrutture per la creazione di comunità virtuali è necessario che l'applicazione delle metodologie e dei tool di sviluppo portino alla realizzazione di MAS in grado di interagire tra di loro. Al fine di promuovere in ambiti industriali il paradigma ad agenti e l'interoperabilità dei MAS, la FIPA (Foundation for Intelligent Physical Agents) [23, 52] si è fatta carico di una meritevole iniziativa di standardizzazione. Tale iniziativa ha come obiettivo fornire delle specifiche liberamente fruibili che costituiscano un punto di partenza ai differenti approcci e realizzazioni di sistemi multi-agente.

Strumenti e metodologie per lo sviluppo di sistemi MAS.

Già da tempo si lavora per offrire strumenti e metodologie di supporto allo sviluppo di sistemi multi-agente; in questo contesto il gruppo di programmazione logica aveva sviluppato presso il DISI di Genova uno strumento per la prototipazione rapida di sistemi ad agenti denominato CaseLP [40]. In tale lavoro si forniva un ambiente di prototipazione basato sulla programmazione logica e si evidenziavano i vantaggi che tale paradigma offre nello studio di soluzioni basate ad agenti. Il presente lavoro di tesi nasce dall'esperienza di CaseLP e dall'esigenza di estendere le funzionalità verso un ambiente di prototipazione distribuito che supporti insieme al paradigma logico, strumenti che godano di una certa popolarità in ambiti industriali.

Struttura della tesi.

Descriviamo in breve il contenuto dei capitoli che compongono questo documento.

Il capitolo 1 può essere suddiviso in due parti. Nella prima diamo una descrizione del contesto dal quale è iniziato il lavoro, forniamo una introduzione minimale ai concetti di agente e sistema multi-agente (MAS), facendo seguire una introduzione alla metodologia ed al tool di prototipazione CaseLP. Nella seconda, seguiamo le linee guida metodologiche proposte da Wieringa [62] e sulla base di considerazioni relative al ciclo di vita del software ed alle peculiarità dei sistemi multi-agente individuiamo l'insieme di requisiti di un nuovo tool per la prototipazione rapida di sistemi multi-agente. L'insieme di requisiti individuati è abbastanza esteso, contenente funzionalità che potrebbero considerarsi a livelli di astrazione differenti, per questo l'abbiamo suddiviso in due sottoinsiemi rispettivamente denominati M-CaseLP e D-CaseLP. I due sottoinsiemi possono essere visti come due progetti distinti fortemente correlati o come due parti di un unico ambizioso progetto. In questa tesi abbiamo focalizzato l'attenzione sulle problematiche relative a D-CaseLP.

Nel capitolo 2, presentiamo un sottoinsieme dei requisiti fra quelli previsti in D-CaseLP che costituisce ciò che intendiamo includere nei primi prototipi del tool. Le funzionalità sono tutte inerenti alle problematiche della distribuzione e della comunicazione degli agenti, quindi, illustriamo un primo studio della distribuzione che vede l'impiego del middleware CORBA [59] . In conclusione del capitolo mettiamo in luce le difficoltà che si incontrano nel fornire la distribuzione degli agenti con tale tecnologia.

Nel capitolo 3, decriviamo le modalità con cui la FIPA sta portando avanti il lavoro di standardizzazione, illustrando le caratteristiche e la portata dell'iniziativa. Presentiamo alcune specifiche che offrono interessanti spunti per il nostro lavoro e concludiamo con alcune osservazioni sull'iniziativa.

Nel capitolo 4, dopo aver sintetizzato le ragioni che ci spingono ad allinearci con il lavoro FIPA, forniamo la descrizione di JADE [7]: un tool aderente alla specifiche di interoperabilità FIPA che offre un run-time per l'esecuzione distribuita degli agenti. Tale framework si pone come un vero e proprio middleware per agenti e parte da ipotesi vicine a quelle effettuate in CaseLP. In conseguenza di ciò, si è studiata la possibilità di fornire le funzionalità previste in D-CaseLP utilizzando questo nuovo strumento. Concludiamo il capitolo con un confronto tra il supporto ai nostri obiettivi offerto da CORBA e quello offerto da JADE e illustrando le motivazioni di progetto che hanno portato a scegliere definitivamente JADE.

Nel capitolo 5, descriviamo le difficoltà nell'integrazione di agenti logici[1] con il run-time JADE. Presentiamo tuProlog [20] una particolare implementazione Prolog che fornisce un'integrazione soddisfacente con Java, grazie alla quale è possibile eseguire agenti logici nella platform offerta da JADE.

In conclusione della tesi, si metteranno in luce quelli che sono i contributi di questo lavoro al progetto D-CaseLP e presenteremo gli sviluppi futuri.

[1] Si legga agenti logici come sinonimo di agenti scritti in un linguaggio logico.

Capitolo 1

Il contesto

In questo capitolo dopo una breve introduzione dei concetti che stanno alla base del paradigma ad agenti, descriviamo il lavoro effettuato dal gruppo di programmazione logica di Genova all'interno del progetto CaseLP, illustrando le motivazioni che hanno portato ad una riformulazione dei requisiti nei termini di D-CaseLP.

1.1 Agenti e sistemi multi-agente

A lungo la comunità scientifica che lavora nel campo degli agenti si è posta il problema di dare una definizione precisa e sufficientemente condivisa di agente. Purtroppo, a rendere difficoltoso tale compito vi è l'inerente multidisciplinarità degli studi sugli agenti. I campi nei quali il paradigma ad agenti trova fondamento sono

- Intelligenza artificiale;
- Programmazione ad oggetti e metodologie associate;
- Programmazione concorrente;
- Simulazione di sistemi.

Le differenti definizioni sono classificate in **strong** o **weak** a seconda che caratterizzino gli agenti in termini "mentali" o meno.

Da qualche tempo a questa parte, il dibattito su cosa e quali caratteristiche debba avere un agente per essere tale, è scemato spostando l'attenzione su problematiche di ordine più pratico. La discussione pur non riuscendo del tutto ad appianare i differenti punti di vista ha prodotto un insieme di definizioni sufficientemente conosciute nella comunità dei ricercatori. In questo lavoro di tesi si

è scelto di adottare le definizioni di *agente* e *sistema ad agenti multipli* proposte in [35]. Il lettore interessato può trovare altre definizioni e caratterizzazioni del concetto di agente in [31, 12, 32].

1.1.1 Definizione di agente

Definizione 1.1 *Un agente è un sistema su computer, situato in un qualche ambiente, capace di agire in modo flessibile, autonomo, allo scopo di raggiungere gli obiettivi per il quale è stato progettato.*

In tale definizione troviamo tre concetti chiave:

Situato: l'agente riceve stimoli dall'ambiente e può agire su di esso per modificarlo effettuando azioni.

Autonomo: l'agente è in grado di agire prendendo iniziativa senza l'intervento umano o di altri agenti avendo quindi il controllo del suo stato e delle sue azioni.

Flessibile: per risultare flessibile l'agente deve essere

reattivo: ricevere gli stimoli dell'ambiente ed essere in grado di rispondere ad essi in maniera tempestiva[1].

proattivo: non deve semplicemente rispondere agli stimoli dell'ambiente, ma dove richiesto, esibire comportamenti guidati da obiettivi.

sociale: deve essere in grado di interagire con altri agenti artificiali e umani quando ciò sia necessario a completare il proprio compito.

1.1.2 Definizione di Sistema Multi-Agente (MAS)

Nello stesso articolo da cui abbiamo preso la definizione di agente, troviamo la definizione di *sistema ad agenti multipli*:

Definizione 1.2 *Un MAS può essere definito come una rete di risolutori (agenti) non strettamente accoppiati che lavorano insieme per risolvere problemi oltre le proprie capacità individuali o la conoscenza dei singoli.*

Il termine MAS è utilizzato per riferirsi genericamente a tutti quei sistemi che sono composti da componenti semi-autonome multiple (agenti) che interagiscono tra loro su principi di cooperazione o competizione e presentano le seguenti caratteristiche:

[1] Prima che la situazione nell'ambiente cambi in maniera significativa per l'agente stesso.

- assenza di controllo centralizzato nel MAS;
- ogni agente ha un punto di vista limitato; possiede infatti informazioni o capacità per risolvere parte del problema alla cui risoluzione è preposto il MAS;
- i dati sono decentrati, suddivisi fra i differenti agenti;
- la computazione degli agenti è asincrona, gli agenti sono concorrenti.

1.2 CaseLP

D-CaseLP è il proseguimento di un progetto abbastanza ambizioso condotto dal gruppo di programmazione logica del DISI di Genova che va sotto il nome di CaseLP[42, 41], il quale oltre alla progettazione e realizzazione di un tool per la simulazione di sistemi munti-agente ha visto lo sviluppo di vari linguaggi per specificare agenti ed una metodologia di supporto allo sviluppatore nelle fasi di analisi e design. Il tool di sviluppo CaseLP pur fornendo una distribuzione logica era fisicamente centralizzato. Gli obiettivi di D-CaseLP sono in gran parte un'estensione del progetto precedente, nella direzione di una distribuzione fisica degli agenti. Al fine di far comprendere l'ottica nella quale D-CaseLP trova fondamento, riportiamo qui alcune parti della tesi [18], che illustrano i concetti base in CaseLP. La sezione 1.3 illustrerà le estensioni a tali concetti.

1.2.1 La concettualizzazione degli agenti

Un agente in CaseLP può essere rappresentato da due diversi punti di vista.
Da un punto di vista descrittivo, un agente è caratterizzato da un'*architettura* e dai *ruoli* che esso ricopre e, di conseguenza, da un insieme di *servizi*.
L'*architettura* si riferisce alla struttura interna dell'agente. Un'architettura specifica sia la struttura dati che ne forma le componenti interne che il flusso di controllo che guida l'attività delle componenti stesse.
I *ruoli* possono essere considerati da due punti di vista: come un insieme coerente di servizi o come le "impersonificazioni" che un agente attua nel corso della sua esistenza. Ad esempio un agente atto alla compravendita ricoprirà alternativamente il ruolo di acquirente e di venditore.
I *servizi* definiscono funzionalità che l'agente, o meglio un ruolo, fornisce o necessita per svolgere il proprio compito. I servizi dipendono tipicamente dal dominio dell'applicazione del prototipo.

Nel seguito considereremo un agente come una entità caratterizzata da

- uno stato;
- un programma;
- un engine.

Lo stato contiene le informazioni che, durante l'esecuzione dell'agente, possono subire modifiche. Ad esempio: un agente che percepisce e gestisce eventi inserendoli e prelevandoli in/da una event queue, e le cui informazioni sul mondo sono rappresentate da credenze, avrà uno stato che conterrà la event queue e le credenze.

Il programma contiene le informazioni che non cambiano durante l'esecuzione. Per il semplice agente descritto sopra, il programma potrebbe essere costituito da un insieme di regole che descrivono quali azioni eseguire in funzione di un evento e delle credenze correnti. Le azioni che un agente può eseguire sono di svariata natura, e potrebbero comprendere ad esempio aggiornamenti dello stato, azioni sull'ambiente e comunicazioni con altri agenti.

L'engine, infine, governa l'esecuzione dell'agente. Sempre riferendosi all'esempio precedente, un tipico engine potrebbe estrarre un evento da una coda, operare la scelta di una regola adatta a gestire l'evento e che sia applicabile nello stato corrente, ed infine attuare l'esecuzione di tutte le azioni indicate nella regola. Va notato come l'engine ed il programma si collochino a due livelli di astrazione diversi: l'engine rappresenta infatti un meta-interprete per il programma e per i dati (stato) su cui il programma opera.

La applicazione delle istruzioni dell'engine allo stato ed al programma dell'agente ne determinano il comportamento. L'architettura di un agente deve consentirne la rappresentazione nei termini sopra descritti. Essa sarà quindi caratterizzata da componenti che contengono lo stato, componenti che contengono il programma, e da un engine che opera su queste componenti. L'assunzione che facciamo sulle componenti che contengono lo stato è che tra esse ci sia una *mailbox* che consenta la comunicazione asincrona tra gli agenti. Ogni agente che considereremo avrà quindi una architettura che specializzerà quella descritta in Figura 1.1.

Due agenti possono avere le stesse componenti, contenenti gli stessi dati, ma engine diversi, oppure possono avere lo stesso engine e le stesse componenti, contenenti però dati diversi. In entrambi i casi il comportamento degli agenti risulterà in generale differente.

Il contenuto delle componenti, caratterizzante lo stato ed il programma degli agenti, sarà espresso in un linguaggio oggetto dipendente dalla architettura; l'engine fornirà un interprete per tale linguaggio.

Agenti che interagiscono tra di loro con le caratteristiche dette sopra possono essere descritti in CaseLP a differenti livelli di astrazione, usando i differenti linguaggi e strumenti forniti dall'ambiente. Un Metodo guida l'utente nei vari passi

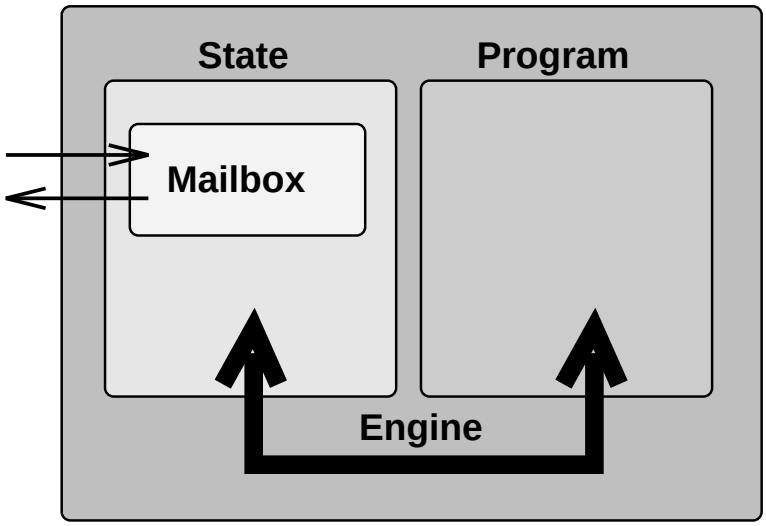

Figura 1.1: Architettura generale.

della prototipazione, dalla descrizione informale dell'applicazione allo sviluppo di un prototipo funzionale. In ogni fase l'utente può adottare il linguaggio e lo strumento più adatto tra quelli forniti.

1.2.2 Una gerarchia di astrazione di concetti

Al fine di permettere una maggiore modularità e flessibilità nella definizione di agenti eterogenei introduciamo una gerarchia di livelli di astrazione, illustrata in figura 1.2.

- Architettura astratta. La architettura astratta definisce le componenti e la struttura di base dell'engine. Per ogni macro-istruzione presente nell'engine è possibile fornire indicazioni sui possibili modi per realizzarla, senza necessariamente arrivare ad un livello di dettaglio paragonabile ad una implementazione. È possibile stabilire una gerarchia di architetture astratte in cui dettagliare in modo incrementale le architetture che verranno utilizzate all'interno del MAS, senza però arrivare a dettagli implementativi. Riprendendo ancora il semplice esempio descritto sopra, una architettura astratta per l'agente dovrebbe prevedere

  - una componente per contenere le credenze;
  - una componente per la event queue;
  - una componente per contenere le regole che definiscono il programma.

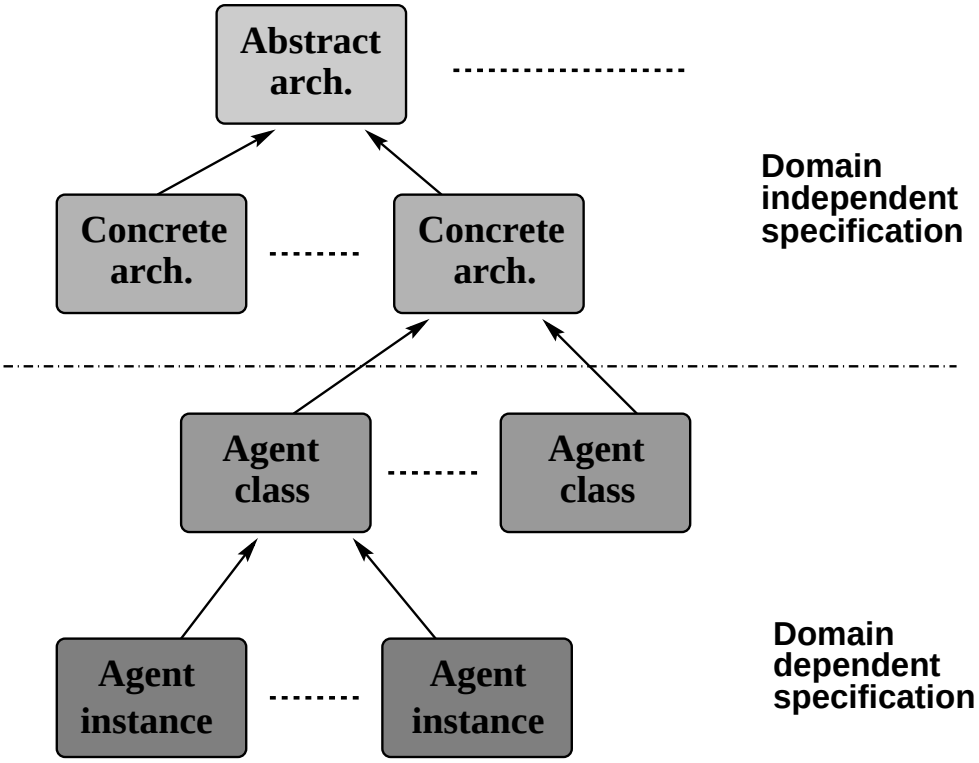

Figura 1.2: La gerarchia architetturale in CaseLP.

L'engine potrebbe essere quello già descritto, composto dalle seguenti macro-istruzioni

1. estrai un evento dalla struttura event queue;
   (a) scegli una regola per gestire l'evento, in base anche alle credenze correnti;
   (b) esegui tutte le azioni indicate dalla regola.

La macro-istruzione dell'engine "scegli una regola per gestire l'evento" potrebbe essere realizzata in modi diversi:

1. scegli a caso una regola;
   (a) verifica che la testa della regola sia uguale all'evento;
   (b) verifica che le condizioni sulle credenze espresse nella regola siano soddisfatte;

(c) se entrambe le verifiche hanno successo restituisci la regola scelta, altrimenti ritorna al punto 1.

oppure, un altra scelta potrebbe essere

1. costruisci l'insieme di tutte le regole applicabili;

   (a) restituisci la regola che, tra queste, prevede il minor numero di azioni da eseguire.

A sua volta costruisci l'insieme di tutte le regole applicabili potrebbe espandersi in altre istruzioni ad alto livello: non si pongono limiti sul livello di annidamento delle definizioni delle macro-istruzioni.

- Architettura concreta. Una architettura concreta si definisce a partire da una architettura astratta nel seguente modo:

  – ad ogni componente si associa un tipo tra quelli previsti dal linguaggio;
  – per ogni istruzione ad alto livello contenuta nell'engine si sceglie una delle possibili indicazioni su come realizzarla (se tali indicazioni sono fornite) e se ne dà una implementazione concreta.

  Ad esempio,si potrebbe stabilire che le credenze di un agente sono contenute in una struttura dati di tipo insieme, mentre le regole sono contenute in una lista. La event queue sarà ragionevolmente implementata con una coda. Inoltre si dovrebbe scegliere che interpretazione dare, tra quelle proposte, alla istruzione scegli una regola, e si dovrebbe infine fornirne una implementazione coerente con le indicazioni fornite.

- Classe di agente. Una classe si definisce a partire da una architettura concreta istanziando le componenti contenenti il programma dell'agente. Tali componenti saranno definite a livello astratto.

- Istanza di agente. A partire da una classe, si completa la istanziazione delle componenti della architettura definendo il contenuto iniziale delle componenti che contengono lo stato.

1.2.3 La metodologia di sviluppo

1. **Descrizione architetturale statica del prototipo.**
   Lo sviluppatore determina la struttura statica del MAS. Questo passo viene ulteriormente suddiviso in:

(a) Determinare quali ruoli sono necessari all'interno del sistema studiato. Tali ruoli sono insiemi logici di servizi, che suggeriscono funzioni complesse che gli agenti presenti nel sistema si troveranno a ricoprire.

(b) Stabilire i legami di soddisfacimento dei servizi tra i ruoli, questo fornisce un primo legame di comunicazione tra gli agenti che, in seguito, ricopriranno tali ruoli ;

(c) Stabilire e definire le necessarie ontologie, ossia le specifiche dei concetti utilizzati nel domino applicativo, necessario per stabilire in maniera univoca il significato dei messaggi scambiati all'interno del MAS.

(d) Raggruppare i ruoli in classi di agenti stabilendo così quali classi di agenti sono necessarie all'interno dell'applicazione;

(e) Per ciascuna classe, determinare il tipo d'agente, la sua architettura e gli interpreti eventualmente necessari; l'architettura può fare riferimento ad una già disponibile in una libreria o essere una nuova architettura che debba ancora essere definita;

(f) Determinare il numero di istanze di ogni classe che sono necessarie alla realizzazione del sistema (ove è possibile stabilirlo a priori);

(g) Definire, dove necessario, le interconnessioni tra le varie istanze di agente e stabilire i legami tra servizi richiesti ed offerti quando è necessario differenziare o specificare a livello di istanza. Questa fase definisce gli agenti che formano il prototipo ed i canali di comunicazione tra essi.

2. **Individuazione dei protocolli di comunicazione tra agenti**
   Il secondo passo della metodologia consiste nello stabilire e descrivere i protocolli di comunicazione (o conversazioni) per ciascun canale, ossia un legame tra l'agente che richiede un servizio e quello che lo fornisce (sia a livello di ruolo, classe o di istanza) stabilito in precedenza. Un protocollo può essere un istanziazione di un protocollo generico o essere un nuovo protocollo definito ad hoc. Un protocollo stabilisce l'ordine e la forma dei messaggi che verranno scambiati. All'interno di ciascun protocollo potranno essere definiti sub-protocolli o comportamenti complessi (scelte, messaggi in parallelo...). Il modello di conversazione del MAS include l'insieme completo di tutte conversazioni stabilite e le eventuali relazioni di sub-conversazione.

3. **Specifica ad alto livello del sistema e degli agenti.**
   A questo punto un linguaggio di definizione ad alto livello è usato per modellare il prototipo. Tre diversi livelli di modellazione possono essere individuati:

(a) Specifica delle interazioni tra gli agenti all'interno del MAS, astraendo dalla loro architettura e tenendo in considerazione il modello delle interazioni specificato al passo 2;

(b) Specifica della nuova architettura scelta durante il passo 1, cioé modellare le interazioni tra le componenti interne dell'agente;

(c) Specifica del comportamento dell'agente.

4. **Verifica e testing del sistema.**
   In questo passo viene effettuata la verifica e/o la validazione preliminare della specifica astratta effettuata al punto 3. La validazione è possibile se il linguaggio di specifica è eseguibile. La verifica è invece possibile se il linguaggio di specifica usato mette a disposizione un meccanismo di prova. Se la specifica è stata fatta utilizzando un linguaggio logico, un'esecuzione goal-directed è un meccanismo di prova valido. In questo caso, data una specifica del prototipo ed una configurazione iniziale del sistema, è possibile seguire l'evoluzione di un MAS in dettaglio. È altresì possibile verificare se una particolare computazione del sistema può essere eseguita o, cosa ancor più importante, che ogni computazione che inizia in particolare stato porti ad uno stato finale che soddisfi le proprietà ricercate.

5. **Implementazione del prototipo.**
   Questo passo trasforma la specifica astratta del sistema nel prototipo finale. Un linguaggio di programmazione logica esteso viene usato per l'implementazione del prototipo. Le interfacce verso moduli software esterni o dati sono fornite utilizzando interpreti appropriati. Inoltre lo scambio di messaggi per le comunicazioni ed altri dettagli implementativi che precedentemente erano stati descritti astrattamente o non erano stati del affatto considerati sono ora realizzati.

6. **Esecuzione del prototipo ottenuto.**
   L'esecuzione del prototipo ottenuto è usata per validare il prototipo secondo le richieste dei clienti. L'esecuzione fornisce agli utenti ed agli sviluppatori informazioni sugli eventi che accadono nel MAS, e permette ad essi di controllare se il prototipo si comporta nella maniera attesa e se possiede tutte le caratteristiche richieste. Ogni errore o malfuzionamento scoperto in questo passo può implicare una revisione di una qualunque delle fasi precedenti della metodologia adottata da CaseLP.

Possiamo quindi distinguere tre fasi principali all'interno del metodo adottato da CaseLP. I passi dall'1 al 4 formano la fase di modellazione. Il passo 5 rappresenta la fase dell'implementazione e il passo 6 quella dell'esecuzione.

1.3 Il progetto D-CaseLP

Si è già detto che D-CaseLP nasce dall'esigenza di estendere il progetto CaseLP, buona parte dei contenuti teorici presenti in D-CaseLP sono frutto di tale esperienza. Le persone attualmente coinvolte nel progetto D-CaseLP sono, oltre al candidato, Stefano Miglia ed Alessandro Rolando rispettivamente con una tesi di laurea ed una di diploma e Viviana Mascardi, con la tesi di dottorato "Logic-Based Specification Environments for Multi-Agent Systems". Il professor Maurizio Martelli coordina il gruppo di lavoro. La finalità del progetto è creare uno strumento che aiuti la progettazione, prototipazione e realizzione di sistemi multi-agente. Purtroppo, per quel che riguarda il tool, sin dal principio si è inteso che, considerata l'esigenza di sostituire il linguaggio Prolog con linguaggi e strumenti (Java, CORBA) più vicini alla realtà industriali, difficilmente si sarebbe riusciti nell'intento di recuperare il codice già esistente. La complessità legata allo sviluppo di un nuovo software, e la suddivisione del lavoro in più tesi ha posto l'esigenza di adottare una metodologia di progettazione. Vista la scarsa forza lavoro di cui disponevamo, si è scelto di seguire le linee metodologiche illustrate in [62] piuttosto che una vera e propria metodologia. Ci siamo avvalsi di tali linee metodologiche essenzialmente nella fase di analisi dei requisiti, per fissare le caratteristiche potenziali e poterci riferire ad esse in corso di discussione.[2] L'analisi è stata fatta tentando di pensare "in grande", senza preoccuparci di cosa potesse effettivamente essere portato a termine nelle nostre tesi, e che cosa no. L'idea era di tentare d'individuare cosa un *tool case* per agenti dovesse offrire.

1.3.1 Assunzioni

Illustriamo quali sono le principali assunzioni che hanno guidato la progettazione del nuovo framework. Il leitmotiv che caratterizza le scelte progettuali operate in D-CaseLP è, per quanto possibile, evitare quelle ipotesi che in qualche modo possano limitare la genericità nell'impiego del framework prodotto.

Quale ciclo di vita per lo sviluppo di sistemi MAS?

Per realizzare uno strumento che sia realmente di aiuto nella progettazione di sistemi multi-agente, è importante aver presente quali siano le problematiche legate alla specifica e sviluppo di sistemi basati su agenti. Altrettanto importante fare attenzione al contesto nel quale gli utilizzatori del tool realizzano il software. A questo scopo, è bene porsi le seguenti domande:

[2] Avendo già fatto notare che si tratta di linee metodologiche nel seguito ci riferiremo ad esse semplicemente con la metodologia Wieringa o la metodologia.

- Chi sono gli utenti del framework che stiamo progettando?
- Quale modello di sviluppo è ragionevole per il software che il framework dovrebbe aiutare a sviluppare?

Nonostante un crescente interesse, mostrato in particolare da aziende di telecomunicazioni, e la realizzazione di un certo numero di casi studio, il paradigma ad agenti è relativamente giovane. Non vi è ancora un'applicazione delle tecnologie ad esso ispirate tale da permettere di delineare come e con quali figure professionali si lavori ad un MAS; né vi sono domini in cui l'impiego di tale tecnologia sia sufficientemente affermato da permettere d'individuare comprovati pattern risolutivi. Tutto ciò rende difficile fare delle assunzioni circa chi siano o come possano lavorare i potenziali utenti di D-CaseLP. La mancanza di standard affermati limita l'utilizzo del paradigma ad agenti negli ambiti industriali, spesso confinandolo a progetti di ricerca. Per questo sembra ragionevole pensare che i contesti nei quali tale tecnologia venga applicata siano caratterizzati da problematiche nuove, le cui soluzioni richiedano un alto contenuto innovativo. Si presume infatti che si impieghi una tecnologia giovane la dove le più comprovate tecniche risultino insoddisfacenti.

La risoluzione di problematiche innovative, presuppone requisiti nuovi; in tali frangenti spesso non è facile individuare e mantenere stabili i requisiti durante la progettazione. Necessitiamo quindi di un modello di sviluppo nei quale i requisiti iniziali possano essere parzialmente delineati, per cui sia necessaria la produzione di alcuni prototipi a fine esplorativo. Sulla base di tali considerazioni abbiamo deciso di ipotizzare che i progetti in cui vengono utilizzati gli strumenti che stiamo progettando seguano un modello di sviluppo simile a quello "a spirale" introdotto da Bohem negli anni '80. Tale modello prevede più iterazioni[3] in cui si susseguono fasi di progetto, analisi, prototipazione e valutazione, come illustrato in figura 1.3.

Analizzando la schematizzazione del modello a spirale partendo dal centro e seguendo la linea fino alla parte più esterna, si può notare che le prime iterazioni portano alla definizione di prototipi, mentre le iterazioni più esterne consistono nel raffinamento dei prototipi fino alla realizzazione del prodotto finale. In figura vi sono quattro iterazioni, ma nulla impedisce di prevederne un numero maggiore. La definizione di prototipi permette di effettuare un'analisi "incremetale", particolarmente utile in progetti in cui per la natura stessa del problema o del dominio applicativo in cui si lavora, sia difficile individuare i requisiti. CaseLP era un ambiente di prototipazione rapida, e quindi forniva già gli strumenti per effettuare una analisi incrementale. Ciò di cui non si disponeva in CaseLPerano gli strumenti per raffinare il prototipo nel sistema definitivo. In altre parole, non si

[3]Con iterazioni su intende un giro completo nei quattro quadranti.

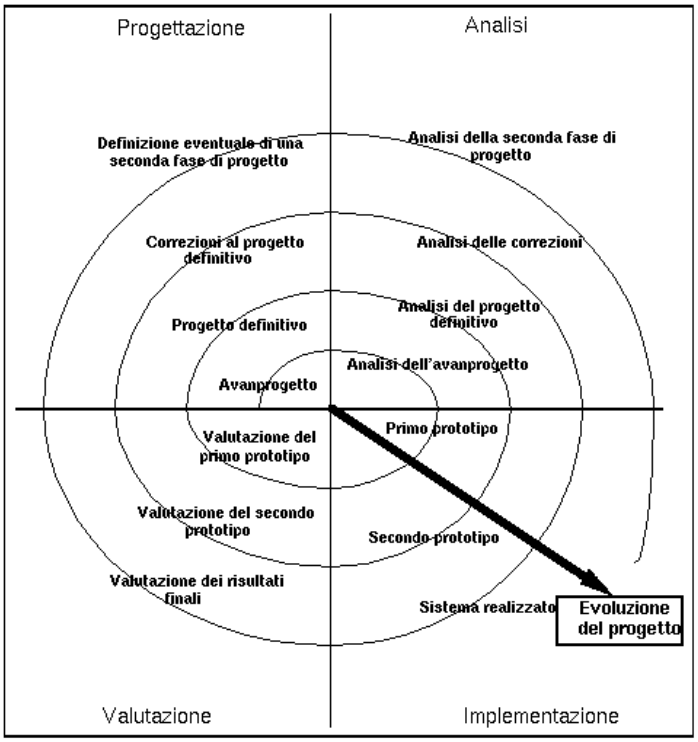

Figura 1.3: Modello a spirale proposto da Bohem.

andava oltre la realizzazione dei prototipi. In D-CaseLP si è cercato di estendere il supporto alla progettazione anche nelle fasi successive alla realizzazione del prototipo, accompagnando l'utente fino alla realizzazione finale del progetto.

Ipotesi minimali sull'architettura degli agenti.

In letteratura sono state proposte diverse architetture per agenti. Se ci riferiamo alla concettualizzazione descritta in 1.2.1, un'architettura per agente definisce il formato e le strutture dati astratte per stato, programma ed engine. Le svariate architetture forniscono differenti specifiche di tali componenti, dando luogo ad agenti con diverse peculiarità. In generale si possono suddividere gli agenti in deliberativi, reattivi,e ibridi. Consideriamo le definizioni illustrate in [63].

Definizione 1.3 *Un agente viene detto deliberativo qualora mantenga una rappresentazione logica dell'ambiente in cui è situato; della conoscenza procedurale, degli obiettivi, e sulla base di essi esegua dei ragionamenti al fine di determinare le sue azioni.*

Definizione 1.4 *Un agente viene detto reattivo qualora non mantenga una rappresentazione simbolica di ambiente, credenze ed obiettivi. Non avendo tale rappresentazione è impossibilitato a ragiornarci sopra. Il comportamento orientato agli obiettivi è una proprietà emergente, determinato da regole che dicono cosa fare quando arriva un certo input.*

Esistono poi architetture di tipo ibrido che uniscono le caratteristiche delle diverse tipologie. Per ottenere ciò, in generale, si struttura l'agente in una gerarchia di livelli. La gerarchia presenta nei livelli più bassi caratteristiche di tipo reattive, mentre ai livelli più alti fornisce le capacità deliberative e proattive.

Esiste una relazione tra tipologia di agente e architettura scelta. Ad esempio, non tutte le architetture per agente forniscono la struttura adatta a ottenere agenti deliberativi. Considerato che la tipologia dell'agente dipende dai compiti dell'agente stesso, e i compiti dell'agente dipendono dall'architettura del MAS[4] che si sceglie per risolvere un problema, è difficile operare una scelta a priori su quale architettura debbano avere gli agenti di CaseLP o D-CaseLP. Per sviluppare una soluzione, può essere necessaria una certa concettualizzazione degli agenti, la cui resa presupponga di fatto una architettura per agenti piuttosto che un'altra. Si pensi ad esempio ad una soluzione in cui gli agenti siano pensati in termini mentali, quindi tramite desideri, credenze, etc; è chiaro che modellando gli agenti in questi termini sia più naturale utilizzare una architettura BDI (Belief, Desire, Intention)[53] in cui sono supportate eplicitamente le credenze, i desideri e le intenzioni dell'agente piuttosto che una architettura come la subsumption[11, 10], il cui comportamento è codificato con regole del tipo situazione-reazione. Non è neanche scontato che all'interno dello stesso MAS gli agenti abbiano la stessa architettura. A fronte di ciò, ed al contrario di ciò che viene fatto in altri tool (es. Zeus[65], AgentBuilder[1]) CaseLP e D-CaseLP non adottano un'architettura particolare fra quelle presentate in letteratura ma si limitano ad assumere che le diverse tipologie di agente condividano un'architettura minimale, che fornisca capacità di comunicazione, utilizzando una mailbox per ogni agente e primitive di ricezione asincrona, invio sincrono e asincrono di messaggi. La tendenza in altri tool è stata quella di fornire agenti basati su un'architettura ibrida, in modo da poter modellare le differenti tipologie di agenti, con la stessa architettura. Questo approccio permette di supportare semplicemente strumenti per scrivere il program dell'agente, fornendo interfacce grafiche, parser, e simili. Riteniamo però inefficiente costringere l'utente ad usare un'architettura ibrida dove sia sufficiente una

[4]Si noti la differenza che intercorre tra architettura di agente e architettura di MAS: la prima si riferisce alla strutturazione del micro-livello e permette all'agente di avere caratteristiche quali la proattività, rattività e autonomia, ne è ad esempio istanze la BDI. La seconda si riferisce a come è strutturato il sistema MAS ad un macro livello, quindi quanti e quali classi di agente sono impiegate nella risoluzione di un problema.

reattiva. Per questo si è tentato di lasciare all'utente degli strumenti da noi progettati, la libertà di scegliere in che termini modellare il proprio MAS decidendo quale architettura sia più adatta per ogni agente.

Assunzioni sui linguaggi di sviluppo.

La tendenza che sembra affermarsi nella scelta del linguaggio con cui implementare platform e gli stessi agenti, va sempre più nella direzione di Java; non è difficile capire le motivazioni che stanno consolidando questa tendenza se si considerano le "facilitazioni" fornite da Java rispetto alla distribuzione e sincronizzazione di processi e il crescente interesse verso il paradigma ad agenti da parte di aziende che operano in domini applicativi come quello delle telecomunicazioni. Se è naturale pensare a Java in tali contesti, la scelta forse potrebbe apparire meno scontata in domini come quello manifatturiero o di controllo dei processi. Ai fini del nostro progetto, si è pensato che sviluppare il tool in Java potesse essere un modo semplice per fornire uno strumento che garantisse la portabilià su più sistemi operativi.

Linguaggi d'implementazione per gli agenti.

Parlare di linguaggio d'implementazione per agenti è in qualche modo improprio: si dovrebbe parlare di linguaggio con cui esprimere le architetture per agenti[5]. In CaseLP si utilizzava la programmazione logica, attraverso una particolare estensione di Prolog denominata ProlAg. Il tool era di fatto un case per la prototipazione rapida di agenti scritti e specificati tramite linguaggi logici. Il paradigma logico offre la possibilità di programmare ad un alto livello di astrazione, ma non gode di una grande popolarità al di fuori di ambienti di ricerca o dalle università. Essenzialmente per questo in D-CaseLP si è deciso di affiancare ai linguaggi logici Java. Visto il fermento intorno a tale linguaggio, la continua evoluzione dello stesso, aprire alla programmazione in Java significa accedere a molte delle novità tecnologiche degli ultimi tempi, estendendo il bacino di possibili utenti di CaseLP.

1.3.2 Obiettivi

Distribuzione effettiva.

CaseLP offre unicamente una distribuzione logica, che permette di simulare l'esecuzione distribuita degli agenti. In questo modo l'utente del tool percepisce

[5]Anche se vista la minimalità dei vincoli sull'architettura nessuno impedisce all'utente di programmare l'agente senza usare l'astrazione di architettura.

l'esecuzione come se gli agenti fossero residenti su macchine differenti, mentre, di fatto, sono eseguiti su una macchina sola con all'ausilio di uno schedulatore. La distribuzione logica offre un alto livello di astrazione rispetto alla comunicazione, è più che sufficiente se si vuole analizzare un problema senza entrare nei dettagli implementativi, quali livello applicativo e livello di presentazione di rete. Tale tipo di astrazione può essere insufficiente se si vuole andare oltre lo studio dei requisiti verso la realizzazione effettiva di un prodotto. Illustriamo alcune situazioni che giustificano l'introduzione di una distribuzione effettiva per agenti nel tool:

scalabilità: vi sono casi in cui la simulazione o l'esecuzione del MAS è computazionalmente pesante, ed è necessario distribuire il carico;

integrazione di risorse remote: [6] si consideri il caso in cui si stia realizzando un MAS, nel quale alcuni agenti debbano fare riferimento a risorse locate in remoto. In un modello di sviluppo per il software come quello a spirale, è plausibile che si abbia bisogno di accedere ai dati delle risorse anche nelle prime iterazioni dello sviluppo. Nel caso in cui lo sviluppo del software avvenga in una sede differente da dove sono locate le risorse, si deve in qualche modo replicare la risorsa o il suo contenuto. La replicazione di tali risorse può comportare alcuni problemi:

- non si dispone delle licenze software per replicare i server che forniscono informazioni;
- la risorsa richiede requisiti di sistema che non si possiedono;
- la risorsa è hardware, magari è una particolare strumentazione propria del dominio applicativo in cui lavora il cliente, non è quindi facilmente replicabile.

In generale si risolvono i problemi di questo tipo costruendo risorse "fantoccio" che si comportano come le risorse reali, fornendo dati fuori linea simili ai reali. La costruzione di risorse fantoccio non è sempre la scelta ottimale: richiede spesso una notevole mole di lavoro aggiuntivo, e pone il problema di come sostituire le risorsa reale con quella fantoccio alla messa in opera del sistema.
Fornendo la distribuzione fisica degli agenti è possibile costruire, già in fase di prototipazione, un agente da utilizzare come "interfaccia" alla risorsa. Questo tipo di agente, una volta progettato, non presenta i problemi di sostituzione propri delle risorse "fantoccio", se progettato bene è potenzialmente l'agente che comparirà nel MAS definitivo. Il paradigma

[6] Con risorse intendiamo basi di dati, legacy software, dispositivi hardware che forniscono informazioni.

ad agenti, sta emergendo come astrazione per risolvere problemi distribuiti con risorse eterogenee e in uno scenario di questo tipo, la distribuzione fisica degli agenti può essere un prerequisito per soluzioni che permettano un'integrazione di legacy software [7] più efficace;

accesso parziale alla risorsa: il cliente vuole fornire un accesso controllato alla risorsa. Tale eventualità non è così remota, per risorse come le base di dati, dove spesso vi è un server, si hanno strumenti di autenticazione e gestione dei diritti con i quali limitare l'accesso alla risorsa, ma in altri casi (es. hardware) queste funzionalità non sono disponibili. Se disponiamo di una distribuzione effettiva degli agenti potremmo pensare che gli agenti interfaccia siano forniti da terzi, (es. la stessa azienda cliente) e funzionino da filtro fornendo una visione parziale dei dati;

testing di applicazioni: visto che vogliamo accompagnare il progettatore del MAS lungo tutto il modello a spirale, vorremmo aiutarlo anche nelle fasi di testing del codice che sta realizzando e fornigli degli strumenti per distribuire naturalmente e con poco sforzo l'applicazione che ha progettatto e testato in regime di distribuzione logica. Abbiamo già detto che vorremmo che l'applicazione finale del nostro utente fosse un raffinamento dei prototipi che aveva realizzato, vorremmo quindi permettergli di recuperare lo sforzo fatto nel realizzare il prototipo e non costringerlo a riscriversi tutto il codice relativo alla comunicazione. Prevedere la distribuzione fisica significa fornire gli strumenti che gli permettano di passare dalla fase di distribuzione logica a quella di distribuzione fisica.

Comunicazione.

Gli agenti per loro natura sono distribuiti e caratterizzati da una comunicazione di tipo asincrono. La comunicazione tra gli agenti coinvolge problematiche a diverso livello di astrazione, come illustrato in [47]. Affinché la comunicazione possa avvenire si devono definire:

- ontologie;
- linguaggio di comunicazione tra agenti;
- protocollo di trasporto.

Nelle fasi iniziali di progetto (prime iterazioni della spirale), è naturale preoccuparsi delle problematiche relative ad ontologie e linguaggio di comunicazione

[7] Per una definizione di cosa si intende con legacy software si veda il paragrafo "Integrazione legacy software " a pagina 21.

tra agenti, una distribuzione logica degli agenti è quindi più che sufficiente. Nelle fasi successive cambiano le esigenze, i problemi che si affrontano coinvolgeranno anche aspetti relativi ai protocolli di trasporto sui quale i messaggi vengono scambiati. Pensiamo ad una applicazione basata su agenti che fornisce degli assistenti virtuali, è abbastanza naturale pensare che questi assistenti possano girare su hardware differente (palmari, cellulari, personal computer, portatili) è ragionevole pensare che un palmtop e cellulari impieghino differenti protocolli di comunicazione (es. WAP) rispetto al personal computer di casa con connessione a internet. In domini applicativi come "l'assistente elettronico" che ci segue ovunque i protocolli di trasporto giocano un ruolo primario, e fornire soluzioni versatili implica includere la progettazione di questo livello di comunicazione nel progetto.

Monitoring di agenti.

Una delle fasi che bisogna considerare in qualsiasi progetto è il testing delle applicazioni sotto sviluppo, il monitoring dell'esecuzione degli agenti è requisito minimale al testing. In un ambiente distribuito si deve poter usufruire di qualche tool che presenti una visione complessiva dell'esecuzione distribuita, dando almeno la possibilità di osservare le comunicazioni fra gli agenti. In CaseLP il problema del monitoring degli eventi che occorrono nell'esecuzione del sistema sotto sviluppo era stato oggetto della tesi [50], si vorrebbero fornire strumenti analoghi per D-CaseLP.

Simulazione.

Abbiamo già detto che D-CaseLP è di fatto un'evoluzione di CaseLP e che un tool di simulazione era già presente nella versione originale. In CaseLP era possibile simulare dei ritardi sui canali di comunicazione degli agenti. In futuro si vorrebbe continuare a usufruire in D-CaseLP degli strumenti di simulazione inclusi in CaseLP e aggiungerne di più raffinati. Una estensione che in prospettiva potrebbe rivelarsi utile è la possibilità di salvare e caricare in simulazione società di agenti predefinite per effettuarne il testing e studiarne il comportamento. Questa caratteristica potrebbe ad esempio facilitare lo studio di tecniche e modelli di coordinazione tra agenti.

Integrazione legacy software.

Riteniamo che la capacità d'integrare risorse e programmi precedentemente sviluppati (*legacy software*) sia fondamentale in ambiti applicativi dove la tecnologia ad agenti affianca software preesistente. A titolo di esempio, pensiamo ad un'applicazione per la gestione del traffico aereo, il cui principale obiettivo è una

razionalizzazione del traffico aeroportuale. Questo caso di studio è stato analizzato da Kinny e altri in [38] come esempio per illustrare la metodologia BDI. Nella soluzione descritta vi è un agente Aircraft per ogni aereo, il quale interagisce con un agente nella torre di controllo per contrattare l'ora di atterraggio. L'agente Aircraft deve essere in grado di interagire con il sistema dell'aeroplano per svolgere i propri compiti e prevedere a che ora potrà atterrare in aeroporto. In questo esempio il sistema software dell'aereo si può considerare *legacy software*.
L'integrazione del legacy software non è una problematica esclusiva dello sviluppo dei sistemi ad agenti: sono allo studio da anni soluzioni per tali problematiche[8]. Ciò che si può osservare è che nelle applicazioni basate sul paradigma ad agenti il problema viene accentuato dalla natura eterogenea dei risolutori, per causa stessa della definizione di MAS data in 1.2. Probabilmente, fornire soluzioni per una integrazione efficace, si può considerare come una precondizione affinché la tecnologia ad agenti prenda piede al di fuori dei progetti di ricerca. Nell'articolo [32] sono presentati tre approcci per agentificare il legacy software schematizzati in figura 1.4 di cui diamo una breve descrizione:

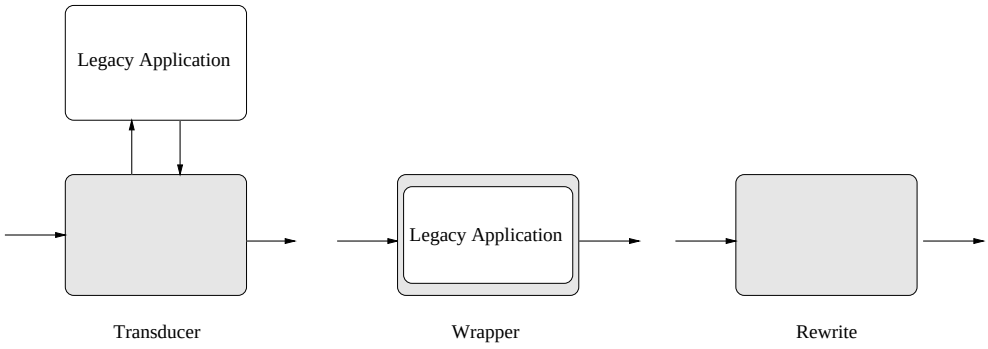

Figura 1.4: Tre modi per agentificare legacy software.

Transducer: accetta messaggi da altri agenti e li traduce in una forma comprensibile alla *legacy application*. Quindi accetta le risposte dell'applicazione e le spedisce utilizzando messaggi scritti secondo un linguaggio di comunicazione per agenti (es. KQML);

Wrapper: interagisce con il legacy software , attraverso una maggiore coesione: può esamininare, accedere, modificare direttamente le strutture dati della legacy application;

[8]Si pensi ad esempio al successo della architetture *three tier* e middleware come CORBA che nascono proprio per aiutare l'integrazione di software eterogeneo.

Rewrite code: si riscrive l'applicazione legacy da capo. Si sceglie questa strada nel caso in cui non sia possibile utilizzare nessuno dei precedenti approcci.

Per utilizzare un approccio a *transducer* è necessario che l'applicazione sia in grado di comunicare con l'esterno secondo qualche protocollo rispetto a cui l'agente funge da interprete. Nel caso in cui non vi sia la possibilità di interagire con l'applicazione tramite un protocollo ma si disponga del codice sorgente, si può costruire un *wrapper*. Tale approccio permette di effettuare chiamate dirette alle procedure del software da integrare; non essendovi alcun protocollo può risultare più efficiente rispetto al primo. Anche se allo stato dei fatti all'interno di D-CaseLP non trova spazio uno studio più approfondito di tali problematiche, vista l'importanza in ambiti applicativi si prevede in futuro di analizzare il problema in modo più dettagliato. Riteniamo possa essere interessante fornire delle linee guida metodologiche che aiutino a gestire l'integrazione del legacy software.

1.3.3 Linee guida metodologiche

In questa sezione illustriamo brevemente le linee guida che abbiamo seguito nelle fasi iniziali della progettazione di D-CaseLP, iniziando dalla concettualizzazione del software che dà Wieringa.

Concettualizzazione del software: Subject Domain, System e Context Boundary.

Definizione 1.5 *Il subject domain è definito come l'insieme di tutti i soggetti dei messaggi che passano dall' environment al sistema.*

Definizione 1.6 *Con environment ci riferiamo all'ambiente in cui il sistema che stiamo progettando è posto.*

Vediamo un esempio al fine di illustrare i due concetti: consideriamo un sistema di monitoring di pazienti in un reparto ospedaliero. Includeremo nel subject domain tutti i termini del dominio applicativo che sono rilevanti per il sistema, quindi nel caso in cui un paziente stia male potremmo passare al sistema le informazioni relative alla temperatura, il battito cardiaco del paziente, ecc. In tale contesto l'environment sarà composto dal paziente, le apparecchiature di monitoring, le infermiere, i medici, e tutto ciò fa parte della realtà che stiamo cercando di modellare. All'interno dell'ambiente possiamo individuare *context boundary* e *system boundary*: abbiamo bisogno di questi due concetti in quanto nell'ambiente non tutte le entità saranno ugualmente significative per la progettazione del sistema.

Definizione 1.7 *Il context boundary rappresenta il confine oltre il quale le entità che appartengono all'ambiente non sono così rilevanti per i nostri scopi da essere modellate.*

Definizione 1.8 *Il system boundary rappresenta il confine all'interno del quale le entità non vengono modellate ma progettate.*

Continuando a ragionare con l'esempio del monitoring di pazienti, potremmo includere nel context boundary il monitor che campiona la pressione o il battito cardiaco con una certa frequenza temporale. Dal momento che apperecchiatura del genere si trovano in commercio è ragionevole non progettarne una da capo ma lavorare su un suo modello. Al contrario, la base di dati sulla quale registriamo i campionamenti della pressione, essendo dipendente dai requisiti del progetto sia la scelta delle informazioni da registrare nella base di dati, sia il formato e le associazioni con cui rappresentiamo le informazioni[9], sarà posta all'interno del system boundary.
Definiamo ora come le informazioni attraversano un generico sistema una volta fissati i system e context boundary nell'environment. In figura 1.5, possiamo individuare

Event: evento che ha luogo nell'ambiente circondante il sistema, di cui il sistema deve tenere conto, per esempio il fatto che un paziente ha una crisi cardiaca.

Stimuli: ogni evento sollecita il sistema con uno o più stimoli che saranno elaborati. Gli stimoli sono le informazioni sull'evento che abbiamo deciso essere rilevanti, ad esempio battito del cuore, temperatura corporea, ecc...

Response: risposte in reazione agli stimoli che il sistema elabora, ad esempio necessità di dare un allarme.

Action: disposizioni che vengono attuate, agenti sull'ambiente del sistema in corrispondenza delle risposte elaborate, es. accensione di una spia di allarme.

L'infermiera farà parte dell'ambiente del sistema, la spia che si accende sarà all'interno del context boundary.

[9]Consideriamo la scelta dei dati dipendente dai requisiti in quanto solo conoscendo quali interrogazioni verranno effettuate sulla base di dati siamo in grado di determinare quali informazioni sia necessario memorizzare. Anche il formato con cui memorizziamo i dati può essere dipendente dai requisiti, almeno se con formato intendiamo lo schema delle tabelle in cui esprimiamo un dato che non è un tipo di dato semplice.

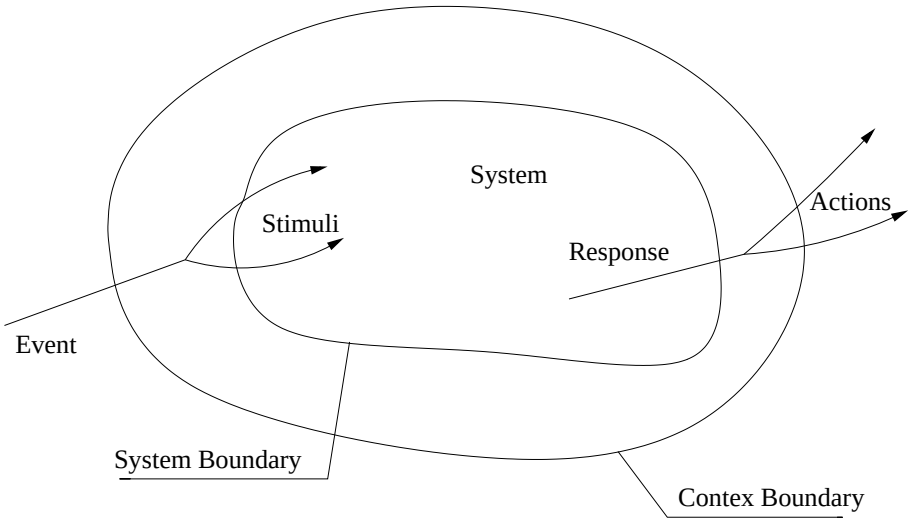

Figura 1.5: Schematizzazione dei concetti con cui viene caratterizzato il software da Wieringa.

Function Refinement Tree (F.R.T.).

Un Function Refinement Tree è una lista gerarchica delle funzionalità che vogliamo siano presenti nel sistema. Può essere rappresentato graficamente come albero vero e proprio o in forma testuale come una lista indentata. Ogni nodo è etichettato con una funzione, le funzioni che etichettano nodi figli rappresentano funzionalità più specifiche rispetto alla funzionalità che etichetta il nodo padre.

Vediamo alcune delle motivazioni che giustificano l'impiego dei function refinement tree.

- Chiarire *system mission*: definire brevemente quale sia lo scopo principale del sistema che stiamo progettando. Il nodo radice del function refinement tree dovrebbe essere la missione del sistema.

- Evitare il *gold plating*: impedire l'aggiunta di funzioni fantasiose e inutili rispetto i requisiti durante le fasi successive.

- Evitare il *function drift*: impedire la perdita di alcune delle funzioni originariamente previste.

- Fornire una visione indipendente dall'implementazione: piché siamo a livello di requisiti le funzionalità devono essere vicine al linguaggio del dominio applicativo, illustrando *cosa* vogliamo ottenere e non *come* lo vogliamo ottenere.

Un Function Refinement Tree dovrebbe essere costruito con la collaborazione del cliente. È consigliabile annotare ognuna delle funzioni desiderate nel sistema su un post-it, e organizzare gerarchicamente i post-it concludendo il raffinamento quando il livello di dettaglio delle funzioni non è più comprensibile dal cliente.

Linee guida: modellazione dell'ambiente e analisi dei requisiti.

L'approccio illustrato da Wieringa per sviluppare software ad oggetti, si compone di quattro attività principali[10]:

Specifica dei requisiti: fase in cui si individuano e specificano i requisiti, tramite *use case diagram* e *Function Refinement Tree*.

Modellazione dell'ambiente: si fornisce una descrizione dell'ambiente in cui il software dovrà operare, di solito si costruisce un Entity Relationship Diagram.

Progettazione dell'architettura: gli use case sono analizzati per definire l'architettura del sistema e suddividerla in sotto parti. I diagrammi tipici in questa fase sono *class diagram, statechart, interaction diagram*.

Implementazione del progetto: si associano le classi all'architettura del sistema attraverso *component diagram* e si mostra come le componenti sono allocate nei nodi fisici della rete.

Le fasi sopra descritte non avvengono tutte sequenzialmente, la specifica dei requisiti e la modellazione dell'ambiente vanno di pari passo. L'ambiente in cui il sistema è posto viene modellato al fine di comprendere i requisiti del sistema. L'idea è che il sistema sotto sviluppo debba aiutare l'ambiente a raggiungere i propri obiettivi ed i requisiti siano quindi una derivazione delle esigenze dell'ambiente. Continuando sull'esempio del monitoring di pazienti, non è detto che il modello dell'ambiente si limiti a considerare la strumentazione presente in reparto col quale il sistema deve interagire. Potrebbe risultare utile estendere l'ambiente all'organizzazione del personale per avvertire le persone "giuste" a seconda del tipo di emergenza che si presenta. Cosa viene incluso nel modello e cosa no dipende da dove si pone il Context Boundary.
Le linee guida dettano principi che possono aiutare a verificare la bontà della progettazione effettuata. Per esempio il criterio di **Traciability**, dice che deve essere possibile seguire il trasformarsi degli eventi in stimoli, risposte e azioni all'interno dei differenti livelli di dettaglio, man mano che si completa la progettazione. Ciò che si vuole, è poter verificare come evolvono le cose all'interno delle specifiche

[10] I diagrammi di seguito nominati si riferiscono alla notazione UML [48].

quando si verifica un certo evento, e poter scegliere a quale livello di dettaglio guardare. Possiamo quindi vedere il sistema come se fosse formato da un certo numero di strati annidati. Ogni strato rappresenta una visione del sistema ad un livello di astrazione differente. Vorremmo essere in grado di determinare come si trasformano le informazioni dello strato precedente man mano che passiamo agli strati più interni. Considerato che ogni livello di astrazione è descritto con diagrammi differenti, il principio di traciability si applica in ultima istanza ai diagrammi. Per esempio, nella fase di progettazione dell'architettura, si dovrà porre attenzione che il class diagram offra le funzionalità per realizzare ogni use case, associando ad ogni use case le classi, le operazione e i metodi[11] che realizzano lo use case.

1.3.4 Un Function Refinement Tree per D-CaseLP

Abbiamo costruito un function refinement tree per D-CaseLP al fine di delineare i requisiti che volevamo includere nel progetto cercando di includere tutte le funzionalità che dovevano essere fornite, se non in un futuro immediato, in prospettiva.

1. Modellazione del MAS e creazione di codice per il prototipo di MAS.
   (a) Modellazione ed integrazione di componenti riutilizzabili (indipendenti dall'applicazione).
      i. Modellazione ed integrazione di architetture.
      ii. Modellazione ed integrazione di ruoli (role models).
      iii. Modellazione ed integrazione di ontologie.
   (b) Istanziazione del MAS.
      i. Scelta dei ruoli necessari nel MAS.
      ii. Scelta dei protocolli tra i vari ruoli.
      iii. Istanziazione della classi.
         A. Scelta dei ruoli che caratterizzano ogni classe.
         B. Scelta della architettura che caratterizza la classe, ed istanziazione del "programma" che caratterizza la classe.

[11] Si considerano gli uni o gli altri a seconda di quanto sia astratto il class diagram che si sta analizzando. Alla fine del design del progetto, il class diagram sarà completo con tutte le classi e i relativi metodi; quindi si potrà effettuare una verifica su di essi. Nelle fasi iniziali della progettazione, si disporrà di classi e operazioni più astratte, in cui non sarà ancora chiara l'associazione funzionalità-operazioni-metodi.

iv. Istanziazione delle ontologie.
v. Istanziazione del MAS (architettura statica del MAS): definizione delle istanze di agenti mediante istanziazione dello “stato”.

(c) Creazione delle specifiche/eseguibili.

i. Controllo statico della sintassi del “programma” che caratterizza le classi e dello “stato” che caratterizza le istanze di agenti.
ii. Creazione automatica del codice eseguibile delle istanze degli agenti.

2. Realizzazione e validazione di un prototipo della applicazione sotto sviluppo.

(a) Installazione degli strumenti offerti da D-CaseLP.

- Supporto all’installazione via procedure di installazione.
- Documentazione per l’installazione e l’utilizzo.

(b) Implementazione prototipo del MAS.

i. Integrazione.
   - Integrazione legacy software.
   - Integrazione semantica.
ii. Supporto alla comunicazione.
iii. Supporto alla mobilità.

(c) Monitoraggio e Simulazione dell’esecuzione del prototipo del MAS.

i. Configurazione dell’esecuzione.
   - Come distribuire gli agenti.
   - Dove recuperare il codice degli agenti.
ii. Monitoring grafico.
   - Visualizzazione della distribuzione degli agenti.
   - Visualizzazione delle comunicazioni fra gli agenti.
iii. Strumenti di simulazione.
   - Parametri della simulazione.
     – Ritardi sui canali di comunicazione o canali inaffidabili.
     – Input agli agenti in simulazione.
     – Attivazione degli agenti durante la simulazione.
     – Gestione del tempo.
   - Supporto per la visualizzazione dei risultati.
     – Visualizzazione On-Line.

– Visualizzazione Off-Line.

3. Implementazione della applicazione finita.

4. Utilizzo della applicazione finita.

Suddivisione del lavoro.

Il functional refinement tree è divisibile in due parti:

1. Modellazione del MAS e creazione di codice per il prototipo di MAS.

2. Realizzazione e validazione di un prototipo e dell'applicazione finita.

Mantenendo questa suddivisione abbiamo organizzato il lavoro in due sotto-progetti:

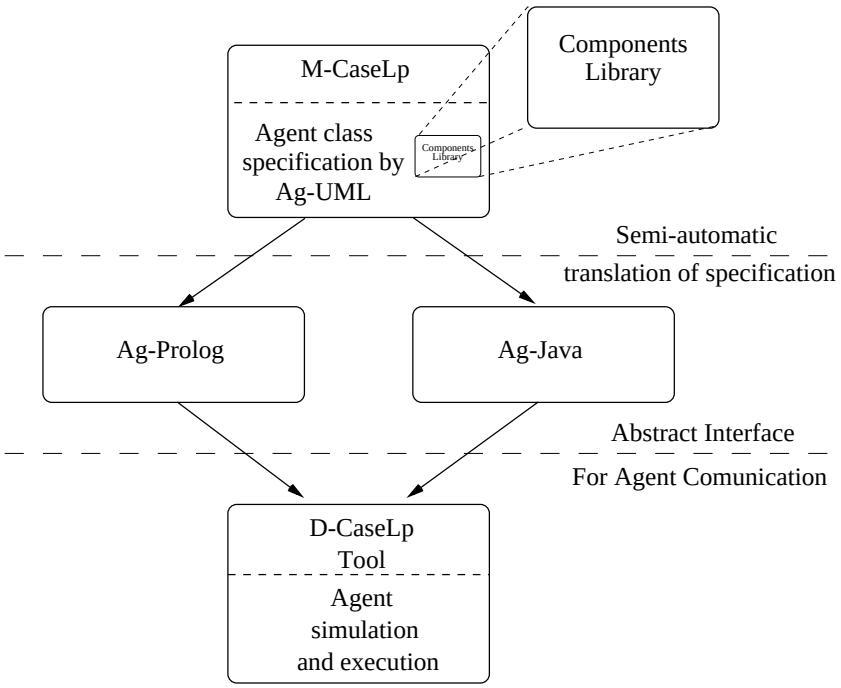

Figura 1.6: Un prima suddivisione del progetto D-CaseLP.

M-CaseLP. Nel quale sono affrontati gli aspetti metodologici con i seguenti obiettivi:

- integrare la metodologia proposta in CaseLP con tecniche di role modeling[12];
- definire una notazione derivata da UML (Ag-UML) per specificare gli agenti;
- fornire indicazioni sulle problematiche relative alla traduzione semiautomatica delle specifiche Ag-UML in agenti prototipali.

D-CaseLP. Nel quale si trovano tutti gli aspetti relativi all'esecuzione degli agenti prototipali, nell'intento di fornire un ambiente in cui si possano

- distribuire ed eseguire gli agenti;
- accedere a strumenti per il testing durante l'esecuzione agenti;
- accedere a funzionalità di simulazione che siano supporto all'esecuzione ed il testing degli agenti.

Come illustrato in figura 1.6 gli agenti descritti nella specifica Ag-UML vengono tradotti in agenti prototipali (Ag-Prolog o Ag-Java) attraverso una traduzione semiautomatica, dopo di che, possono accedere alle funzionalità di D-CaseLP attraverso l'interfaccia astratta delle primitive di comunicazione. L'ambiente fornito in D-CaseLP oltre a tutte le funzionalità di testing e simulazione, offrirà l'implementazione di queste primitive.

La suddivisione in M-CaseLP e D-CaseLP corrisponde a grandi linee anche a come ci siamo suddivisi i compiti all'interno del nostro gruppo di lavoro. Stefano Miglia e Alessandro Rolando si occuperanno delle problematiche correlate ad M-CaseLP. In particolare, Stefano Miglia degli aspetti relativi la definizione della metodologia, la notazione con cui esprimere gli agenti Ag-UML e la traduzione automatica. Alessandro Rolando si occuperà di arricchire la libreria di componenti, implementando alcune architetture per agenti di supporto alla progettazione e traduzione semiautomatica. Invece, le problematiche inerenti a D-CaseLP saranno trattate in questo lavoro di tesi, focalizzandosi prima di tutto sulla comunicazione e distribuzione degli agenti. Tutti gli aspetti vengo coordinati e seguiti dalla dottoressa Viviana Mascardi e il professor Maurizio Martelli sulla base degli obiettivi e l'esperienza CaseLP.

Ovviamente l'albero definito sopra è solo una prima approssimazione dei requisiti del progetto D-CaseLP, ci aspettiamo che la struttura e il contenuto dell'albero subisca delle modifiche durante il progetto in funzione dell'emergere di nuove esigenze.

[12] Role modeling è la modellazione che si basa sul concetto di ruolo, per il quale una entità sia essa agente o oggetto offre delle funzionalità in base ai ruoli che in un certo istante gioca.

1.3.5 Un'interfaccia astratta per far comunicare gli agenti

L'esigenza di suddividere il lavoro in più tesi, ha portato a definire come anello di congiunzione tra la progettazione-specifica degli agenti e l'esecuzione di questi un'interfaccia astratta di comunicazione. L'interfaccia è l'insieme delle primitive con le quali gli agenti comunicano ed hanno la possibilità di fornire e richiedere servizi. Le primitive di cui si compone l'interfaccia sono elencate in tabella 1.1, tutte hanno un parametro msg che svolge una funzione differente a seconda

Primitiva	Tipo di comunicazione
send	asincrona
receive_sync	sincrona
receive_async	asincrona

Tabella 1.1: primitive di comunicazione degli con cui possono comunicare gli agenti D-CaseLP.

che sia il argomento di una primitiva di ricezione o si spedizione. In entrambi i casi il parametro msg farà riferimento ad un qualche Agent Comunication Language (ACL). Senza entrare troppo nei dettagli, un messaggio scritto in un ACL da un punto di vista astratto corrisponde ad una tupla di coppie etichetta-valore rappresentati informazioni quali

- il destinatario del messaggio che supponiamo contrassegnato dall'etichetta :receiver;
- il mittente del messaggio che supponiamo contrassegnato con :sender;
- l'atto comunicativo che si vuole esprimere con il messaggio contrassegnato con :performative);
- il corpo del messaggio contrassegnato con :body.

Un semplice esempio di messaggio scritto in un ACL fittizio, potrebbe essere:

```
(:sender Pippo, :receiver Pluto,
   :performative Richiesta,
   :body "Una cassa di mele")
```

Questo messaggio esprime tre cose:

1. un agente di nome "Pippo" comunica con l'agente di nome "Pluto";
2. la comunicazione corrisponde ad un atto di tipo "Richiesta";

3. la richiesta riguarda una cassa di mele.

Mentre ci aspettiamo che il msg parametro della primitiva di send sia una tupla con tutte o quasi le componenti istanziate[13], nel caso delle primitive di ricezione, in genere non è così. Di solito un agente in ricezione non può specificare tutti i campi del messaggio che aspetta perché non li conosce: magari conosce la tipologia di messaggio che sta aspettando (es. una richiesta) o il mittente del messaggio nel caso voglia servire qualcuno in particolare, ma di solito è interessato a selezionare i messaggi che gli arrivono. Per permettere ciò i msg delle primitive di ricezione possono essere tuple del tipo precedentemente decritto ma con qualche campo vuoto, in modo da rappresentare una specie di schema per i messaggi che si desiderano selezionare ad un certo punto della computazione. Alla base della selezione dei messaggi vi è il meccanismo del **Pattern Matching**, il quale partendo da un messaggio parzialmente istanziato, lo schema di messaggio detto anche pattern, individua quali sono i messaggi compatibili con esso. Facciamo un esempio, abbiamo l'agente "Pluto" di prima, che è un fornitore di frutta, ed è quindi interessato ad tutte le richieste da parte di tutti gli utenti. Tale agente specificherà nel parametro msg qualcosa del tipo

```
(:sender *, :receiver Pluto,
   :performative Richiesta, :body *)
```

dove lo '*' individua l'assenza di un valore nel campo[14]. Il primo messaggio compatibile con lo schema scelto sarà così selezionato all'interno della mailbox e restituito dalle primitive di ricezione in un parametro di ritorno o sovrascrivendo il parametro che individuava lo schema.

1.3.6 Dalle specifiche in Ag-UML all'esecuzione in D-CaseLP

In questa sezione descriviamo più dettagliatamente l'iter di massima che gli sviluppatori potranno seguire nella realizzazione dei loro sistemi multi-agente.

Tralasciando gli aspetti legati al processo di progettazione che coinvolgono la metodologia M-CaseLP, in figura 1.7 abbiamo illustrato uno schema che fornisce l'idea di come avviene la traduzione a partire dalle specifiche di agente espresse in notazione Ag-UML fino agli agenti prototipali. Le specifiche di agenti in Ag-UML contengono l'indicazione su quali carateristiche (es. ruoli, architetture per agenti, protocolli d'interazione per gli agenti) sono state attribuite all'agente durante

[13] Nell'esempio le specifichiamo tutte anche perchè l'ACL che stiamo utilizzando è molto semplice. In genere negli ACL realmente utilizzati per la comunicazione fra agenti vi sono ontologie ed altre informazioni (es. un riferimento al protocollo d'interazione utilizzato).

[14] Si sarebbe potuto anche lasciare uno spazio bianco o utilizzare un altro carattere purché non vi siano ambiguità, per il nostro livello di astrazione la scelta del carattere non ha importanza.

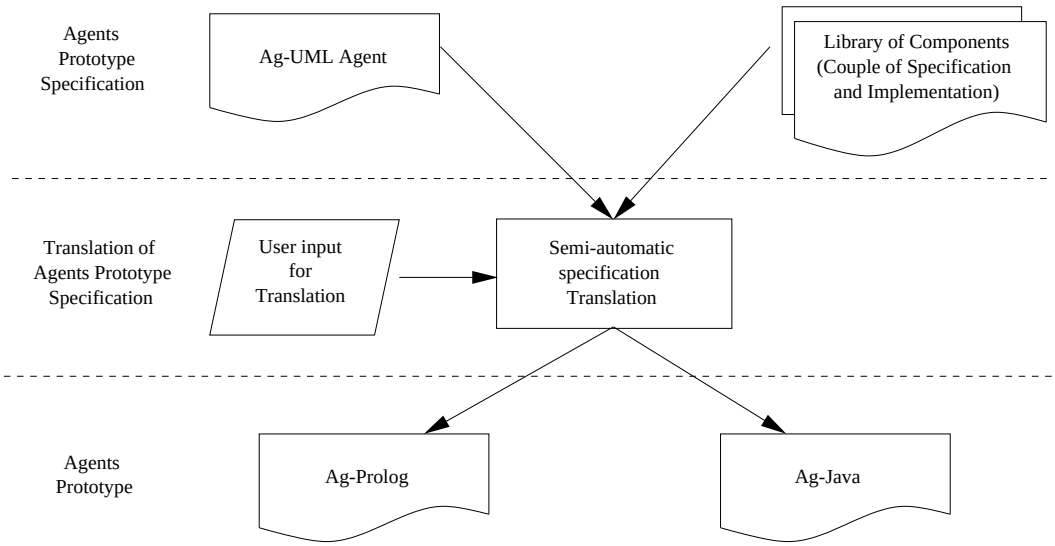

Figura 1.7: Iter seguito in M-CaseLP.

la prima fase di progettazione del MAS. Ipotizziamo che la libreria delle componenti contenga tutte le implementazioni delle componenti che compaiono nelle specifiche Ag-UML di un certo agente e che di conseguenza la traduzione automatica consista essenzialmente nell'assemblare le realizzazioni di tali componenti pescando dalla libreria. In prima approssimazione si è prevista una traduzione semi-automatica delle specifiche Ag-UML che fornisce degli "scheletri" ai quali si dovrà aggiungere la logica dipendente dal dominio secondo le esigenze della soluzione che si sta approntando. Gli agenti prototipali Ag-Prolog e Ag-Java sono agenti scritti rispettivamente in codice Prolog e Java.

Facciamo un semplicissimo esempio per chiarire il tipo di intervento che ci aspettiamo dall'utente.

Si consideri una specifica di agente nella quale si sia indicato l'utilizzo di una certa architettura. La parte di traduzione che possiamo ottenere in modalità automatica è limitata all'inserimento dell'implementazione dell'engine dell'architettura nel codice dell'agente, mentre il program e lo stato dell'agente essendo dipendenti dalla particolare soluzione e dal dominio applicativo non sono conosciuti a priori. Il program e lo stato dell'agente in questo caso rappresentano la logica dipendente dal dominio applicativo, l'unico in possesso di tale logica è il progettista, ed è quindi lui che dovrà interagire con il processo di traduzione per fornirla. Le informazioni ottenute grazie all'intervento dell'utilizzatore possono essere viste come la logica del dominio con cui si specializzano le componenti generiche indipendenti dal dominio.

In figura 1.8 abbiamo illustrato come il procedimento di traduzioni continui

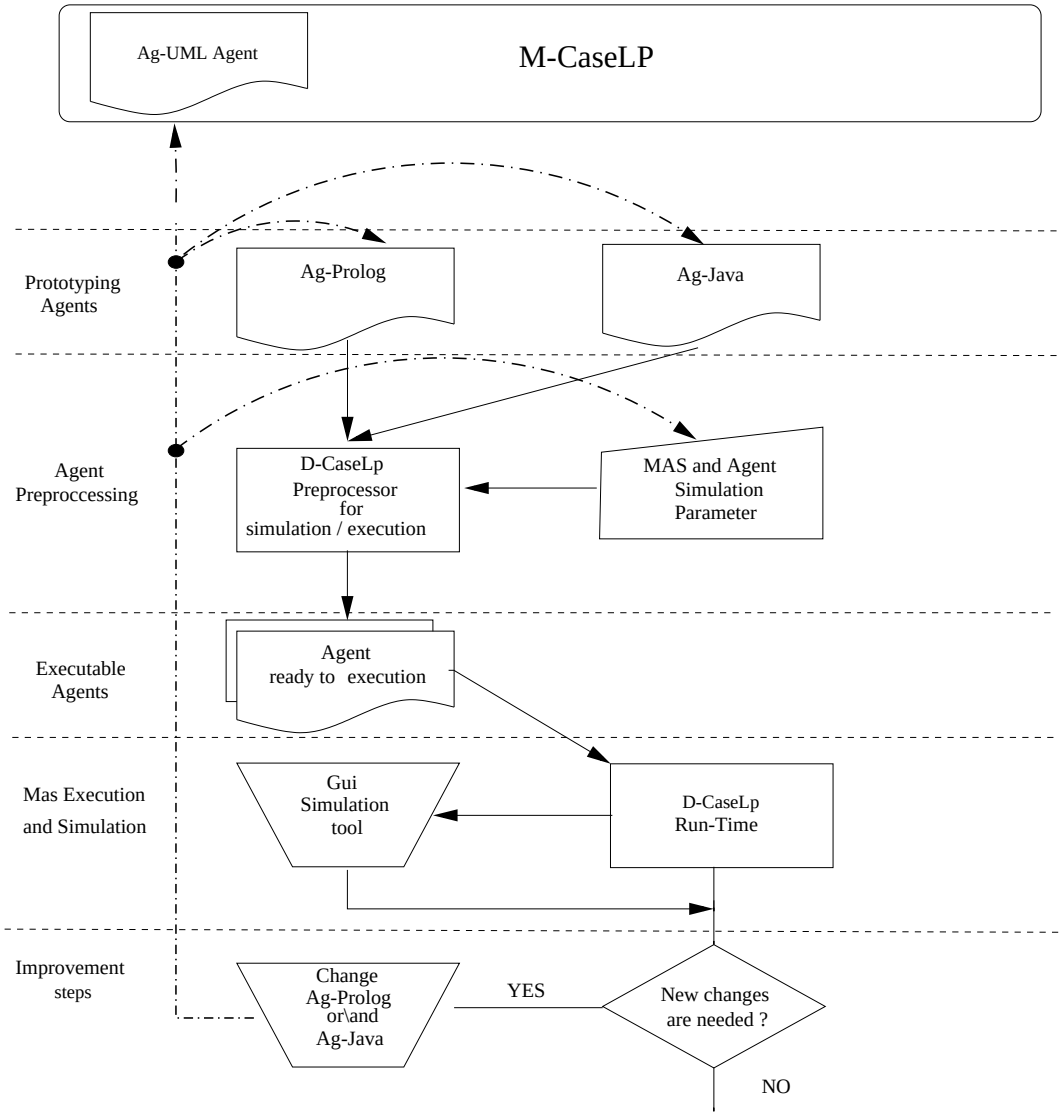

Figura 1.8: Iter seguito in D-CaseLP.

dopo la creazione degli agenti prototipali. Lo scopo di questa parte di iter è aggiungere agli agenti prototipali le informazioni neccessarie alla loro esecuzione nel Run-Time D-CaseLP. Si prevede una fase di preprocessing in cui tenendo conto degli eventuali parametri di simulazione viene manipolato il codice degli agenti al fine di aggiungere i layer di comunicazione/esecuzione ed ottenere del codice eseguibile con l'ausilio del D-CaseLP run-time. Durante l'esecuzione del run-time sarà possibile accedere a strumenti di monitoring degli agenti, utili alla verifica del corretto funzionamento del MAS sotto sviluppo. A seconda dell'esito di tale esecuzione si andrà a ritoccare il codice degli agenti, i parametri di simulazione o nel caso in cui i problemi riscontrati coinvolgano la prima parte della progettazione del MAS le specifiche Ag-UML. Gli schemi dell'iter presentati in questa fase hanno lo scopo di dare un'idea di come si potrà procedere nell'utilizzo del tool da noi prodotto una volta completato, ovviamente in questa fase di progetto non si può che dare un iter di massima. Si avrà cura di precisare le modifiche a cui sottoporre gli schemi man mano che si individueranno gli strumenti le funzionalità saranno offerte.

Nei prossimi capitoli verrà illustrato lo studio di realizzazione del tool D-CaseLP considerando diversi strumenti per realizzare la distribuzione e la comunicazione degli agenti Ag-Prolog e Ag-Java.

Capitolo 2

Valutazione di CORBA per realizzare MAS

In questo capitolo illustriamo lo studio della distribuzione degli agenti in D-CaseLP utilizzando gli strumenti offerti da CORBA. Nelle prime sezioni si introducono i concetti e le principali caratteristiche di CORBA, successivamente al fine di individuare un insieme minimale delle funzionalità da realizzare si presenta un sottoinsieme del F.R.T descritto nel capitolo precedente. Si conclude illustrando un primo design della comunicazione realizzata in CORBA, cercando di evidenziare se e quanto gli strumenti offerti da CORBA siano adatti alla realizzazione del nostro tool.

CORBA gode di una certa popolarità come tecnologia per integrare componenti distribuite ed eterogenee. Queste caratteristiche hanno spinto alcuni ricercatori fra i quali Jennings, Sycara e Wooldridge in [35] a candidare il middleware per oggetti come strumento per la distribuzione di agenti eterogenei. Il capitolo ha come obiettivo dare gli strumenti che permettano di rendersi conto delle difficoltà legate all'utilizzo di CORBA, affinché appaia evidente quanto un middleware per oggetti sia di per sé inadeguato al fine di offrire la distribuzione degli agenti[1] e scoraggiare da tale impresa chi si trovi a sviluppare la distribuzione di agenti sotto ipotesi progettuali simili alle nostre.

2.1 CORBA

Come illustrato in [59], i fattori che hanno decretato un generale interesse verso strumenti come CORBA vanno ricercati nel crescente utilizzo di applicazioni

[1] Ciò non esclude che riempendo la differenza di astrazione che intercorre tra la comunicazione *client-server* e quella *peer-to-peer* si possa utilizzare CORBA per realizzare la comunicazione tra agenti.

distribuite su reti eterogenee. Emblematico esempio di questo interesse è il fenomeno Internet e l'impatto che questa tecnologia ha avuto sul lavoro di milioni di persone. Internet è a tutti gli effetti una rete eterogenea: incorpora un insieme di tipi di reti svariato (Ethernet, TCP/IP, Novell Netware) con differenti protocolli e differente hardware (workstation e server UNIX, PC Microsoft Windows, Apple Macintosh).
La convivenza di reti eterogenee può aumentare la flessibilità del sistema produttivo. Giusto per dare un'idea di quali vantaggi siano legati all'eterogeneità delle reti, consideriamo il sistema informatico di una grande azienda. È plausibile che una grande azienda sia suddivisa in reparti o divisioni: ogni divisione avrà compiti specifici, e potrebbe aver bisogno di strumenti informatici (hardware o software) differenti dalle altre. Allo stesso tempo, essendo le divisioni parti della stessa azienda è ragionevole prevedere che vi sia la necessità di condividere alcune applicazioni per le procedure ed i compiti comuni a tutte le divisioni. Poter scegliere strumenti eterogenei e poterli integrare fra di loro, significa avere la possibilità di utilizzare software e hardware ad hoc per i compiti di ogni singolo reparto o struttura, offrendo allo stesso tempo procedure di fatturazione o richiesta d'acquisto unificate attraverso un applicativo condiviso da tutte le componenti dell'azienda.
Nell'aprile del 1989 su iniziativa di alcune aziende (Canon, 3Com Corporation, Sun Microsystems, ecc.) si forma l' Object Management Group (OMG). Il documento costitutivo include fra gli obiettivi dell'organizzazione la produzione di specifiche e linee guida al fine di fornire un strutturazione del software comune e favorire la progettazione e lo sviluppo di applicazioni interoperanti. Fra le specifiche che sono state fino ad oggi prodotte vi sono la Object Model Architecture (OMA), la Common Object Request Broker Architecture (CORBA) e la Unified Modeling Language (UML). La prima delle tre specifiche ha come obiettivo dare un modello delle applicazioni ed individuare una certa strutturazione delle componenti; la seconda specifica descrive un middleware con il quale le componenti possono comunicare; la terza specifica definisce una notazione grafica che può essere utilizzata per la descrizione delle specifiche. Essendo fra gli scopi di questa tesi la distribuzione degli agenti e non la specifica di essi, nelle prossime sezioni analizzeremo solo due delle precedenti specifiche: l'OMA e CORBA.

2.1.1 Object Management Architecture (OMA)

Come illustrato in [59] il OMA è un modello che indica come sviluppare ed impiegare oggetti distribuiti interoperanti in un ambiente eterogeneo. In questo modello ogni oggetto è incapsulato in una entità con una immutabile identità. I servizi che offre possono essere acceduti esclusivamente tramite una ben definita interfaccia, allo stesso tempo l'implementazione dei servizi dell'entità e la locazione della stessa sono trasparenti all'entità cliente. Il modello OMA è composto da un Ob-

ject Model ed un Reference Model. Il primo definisce i concetti che sono alla base della specifica, illustrando l'accezione in cui sono utilizzati termini come identità, richiesta, operazione, tipo, segnatura, ecc. Il secondo descrive la suddivisione in componenti e le interazioni in cui sono coinvolti. In figura 2.1 vi è una schematizzazione del reference model in cui la componente principale, Object Request Broker (ORB), è responsabile della comunicazione tra clienti ed oggetti. Le altre componenti individuano quattro tipologie di interfacce per entità/oggetti:

Object Services: interfacce di oggetti indipendenti dal dominio applicativo che offrono servizi di utilità generale alle altre entità. Esempi di tali servizi in CORBA sono il *Name Service* ed il *Trading Service*. Il primo permette ai clienti di trovare gli oggetti in base all'identificativo, il secondo in base ad una descrizione dei servizi offerti.

Common Facilities: sono interfacce di oggetti orientate alle applicazioni. Rientrano in questa tipologia le interfacce che permettono l'iterscambio di oggetti fra un applicativo ed un'altro sulla base di un modello di documento. Un esempio sono le interfacce che permettono di inserire un foglio di calcolo in un documento prodotto con un elaboratore di testi.

Domain Interfaces: sono interfacce che analogamente alle common facilities e agli object services forniscono servizi, solo che le facilitazioni fornite sono legate a domini applicativi specifici, quali telecomunicazioni, domini finanziari, ecc.

Application Interfaces: sono interfacce sviluppate per apposite applicazioni. Essendo le funzionalità offerte fortemente dipendenti dalle particolari applicazioni, OMG non si occupa di produrre interfacce standard per questa categoria.

2.1.2 Terminologia in CORBA

Di seguito riportiamo la terminologia presentata in [54], la quale introduce i termini che usualmente si usano per parlare della tecnologia e dei concetti propri di CORBA. Tale terminologia ci aiuta a comprendere quale sia il ciclo di vita degli oggetti CORBA, e come le richieste siano servite attraverso l'associazione tra servant e CORBA object.

CORBA object: un'entità virtuale che possiede un identificatore univoco. Attraverso questo identificatore l'ORB è in grado di consegnare le richieste dei clienti al CORBA object.

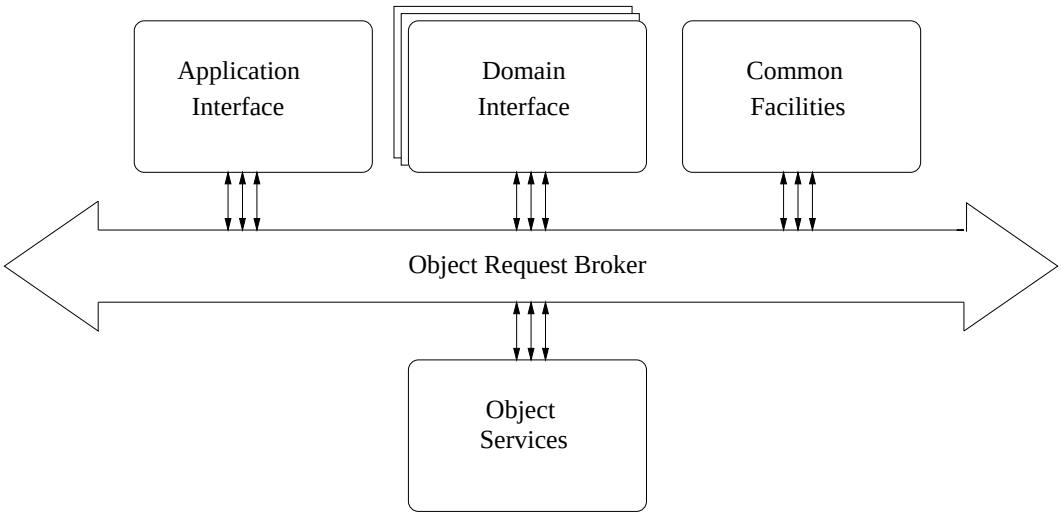

Figura 2.1: Il Reference Model dell'OMA.

Servant: l'entità che fornisce l'implementazione dei servizi indicati in un CORBA object. Il CORBA object riceve le richieste di servizi da parte di una entità cliente. Ad ogni CORBA object è associato almeno un servant che fornisce l'implementazione delle operazioni delle quali si fa richiesta. In CORBA sono previsti differenti linguaggi con cui implementare i servant: Ada, C, C++, Java, Cobol, ecc. Per i linguaggi che sono orientati agli oggetti esiste una corrispondenza tra i metodi dell'oggetto e l'implementazione dei servizi. Nei linguaggi non orientati agli oggetti, il servant è implementato come un insieme di funzioni per manipolare dati; ad esempio in C, un servant potrebbe essere implementato con una struttura. Proprio perché non sono i CORBA object ad eseguire le richieste si definiscono "entità virtuali".

Skeleton: entità che nelle invocazioni statiche permette la connessione di un servant con un Object Adaptor (OA). Ad esempio, in C, uno skeleton è una collezione di puntatori a funzioni dei servant; in C++ uno skeleton è una classe dalla quale tutti i servant per un certo CORBA object devono essere derivati.

Stub: controparte dello skeleton, che permette ai clienti di fare richieste al CORBA object.

Object Id: è un identificatore unico all'interno di un Object Adaptor specificato dal sistema o dall'utente. Non si richiede quindi che tale identificatore

sia universalmente unico, è sufficiente una unicità entro il singolo processo server.

Activation: è l'atto di attivare un CORBA object. Affinché le richieste dei clienti possano essere soddisfatte è necessario che al CORBA object sia associato almeno un servant.

Deactivation: l'atto opposto dell'activation, richiede che sia distrutta l'associazione tra CORBA object e servant. La deactivation non implica la distruzione del CORBA object, infatti, un CORBA object può essere riattivato dopo una disattivazione. La disattivazione può implicare, tuttavia, la distruzione del servant legato al CORBA object che si sta disattivando.

Incarnation: l'atto di associare un servant ad un CORBA object. In altre parole, la incarnation fornisce un "corpo" con le necessarie implementazioni all'oggetto virtuale CORBA object.

Etherealization: è l'atto che distrugge l'associazione tra CORBA object e servant. Dopo l'eterealizzazione il CORBA object continua ad esistere come entità virtuale, ma non avendo servant, e quindi corpo, non può portare a termine le richieste.

Active Object Map: è una tabella che tramite l'object adapter mantiene l'associazione tra CORBA object e i suoi servant. I CORBA object attivi sono identificati in tabella con il loro Object Id.

L'interazione tra gli oggetti nelle applicazioni CORBA si basa su un modello di tipo *client-server*; questo, abbinato a concetti propri della programmazione orientata agli oggetti, fornisce un modello che prevede l'esistenza di un oggetto "client" ed un oggetto "server". CORBA è spesso utilizzato quando vi sono moduli che devono interagire scritti in differenti linguaggi d'implementazione. Tale impiego è una conseguenza all'utilizzo dell'Interface Definition Language, una specie di lingua franca con quale possono interagire le differenti entità. In pratica, ad ogni CORBA object è associata un'interfaccia IDL che descrive quali servizi sono offerti ed a partire da essa si generano i due moduli stub e skeleton.
Abbiamo detto che OMG gestisce unicamente la produzione e l'aggiornamento delle specifiche, esistono di conseguenza differenti implementazioni di tali specifiche, alcune commerciali e altre sotto licenza General Public Licence (GPL). Inizialmente l'esistenza di differenti implementazioni abbinata all'assenza di una specifica del formato con cui interagiscono oggetti e ORB aveva fatto emergere formati proprietari che impedivano di fatto l'interazione tra le differenti implementazione dell'ORB. Questo problema è stato superato con CORBA 2.0, grazie all'introduzione di un'architettura e protocollo per l'iteroperabilità tra ORB

denominata General Inter-ORB Protocol (GIOP). Il blasonato Internet Inter-Orb Protocol (IIOP) non è altro che l'istanza del (GIOP) su un trasporto TCP/IP.

2.1.3 La comunicazione tra client e server in CORBA

Al fine di prendere confidenza con la tecnologia CORBA abbiamo programmato un primo semplice esempio in cui abbiamo un cliente e un server. Il server ha fra i suoi servizi una funzione che stampa un messaggio sul terminale della macchina sulla quale risiede, mentre il client si limita a spedire un certo numero di richieste. Questo esempio è stato realizzato con l'implementazione di CORBA Orbacus [49], distribuita liberamente almeno per utilizzi non commerciali, basandosi sulla documentazione disponibile al download [34].

In figura 2.2 sono illustrate le componenti coinvolte in una richiesta di servizio ad un CORBA object.

La richiesta di servizio parte dal client; il client conosce l'interfaccia dei servizi del server e attraverso l'IDL stub è in grado di formulare la richiesta dei servizi al CORBA object. La richiesta viene consegnata dall'ORB il quale coopera con l'Object Adapter al fine di assicurare che arrivi all'appropriato servant. L' Object Adapter è responsabile del processo di activation e deactivation dei CORBA object e di incarnation e etheralization dei servant. È l'Object Adapter che genera i riferimenti agli oggetti ad esso registrati. Dopo aver individuato il servant destinatario della richiesta, attraverso l' IDL skeleton vengono passati i parametri della richiesta alle funzioni offerte dal servant. Gli eventuali parametri di ritorno frutto dell'elaborazione del servizio vengono direttamente restituiti al client che ha effettuato la richiesta.

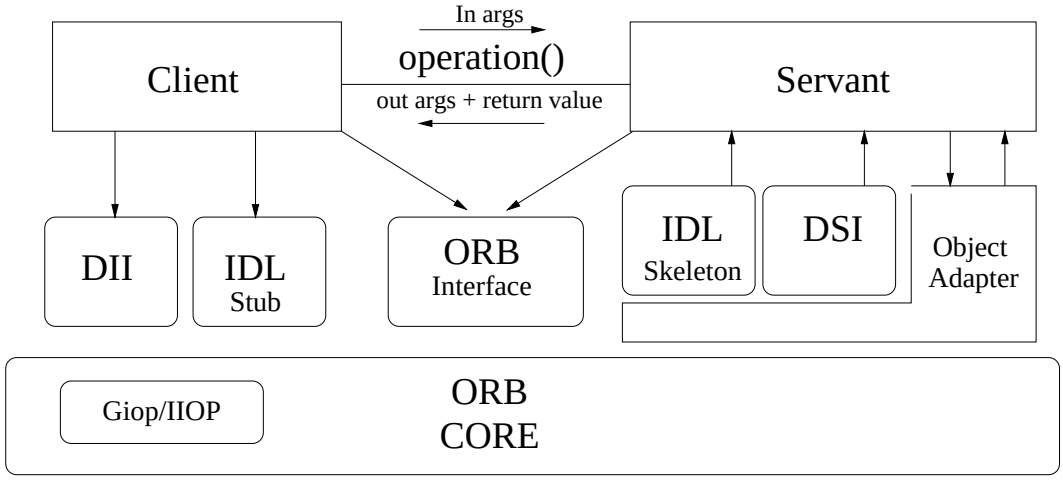

Figura 2.2: Componenti coinvolti in una comunicazione client-server CORBA.

In realtà questo non è l'unico modo in cui può avvenire la comunicazione, vi sono infatti due modi per effettuare l'invocazione di un servizio di un CORBA object:

- tramite invocazione statica;
- tramite invocazione dinamica.

Fino ad ora abbiamo discusso di comunicazioni di tipo client-server, nelle quali le interfacce dei servizi offerti dal server erano fissati staticamente a tempo di compilazione. Questo tipo di invocazione appartiene alla prima delle due categorie. Esiste la possibilità di effettuare invocazioni **dinamiche** grazie all'impiego della *Dynamic Invocation Interface* (DII) e della *Dynamic Skeleton Interface* (DSI). La DII permette ad un'applicazione cliente di effettuare richieste ad un oggetto senza avere nessuna conoscenza a tempo di compilazione dei servizi da esso offerti mentre la DSI permette di scrivere i server senza che lo skeleton debba essere staticamente compilato con il codice del server. Il procedimento con cui avviene l'invocazione dinamica è simile al precedente solo che al posto dell'IDL skeleton e dell'IDL stub vengono impiegate la DSI e la DII. La comunicazione dinamica offre la possibilità a tempo di esecuzione di scoprire nuovi oggetti e utilizzare i servizi ad essi associati.

2.1.4 Servizi CORBA

L'Object Managment Group (OMG) definisce un certo numero di servizi sotto le Object facilities che sono in generale utili allo sviluppo di applicazioni. Come per tutte le specifiche OMG, è compito dei produttori fornire delle implementazioni a ciò che è stato specificato. Elenchiamo alcuni dei servizi previsti:

Name Service: permette di registrare e rendere accessibili oggetti tramite un identificativo.

Event Service: fornisce un meccanismo attraverso il quale gli oggetti CORBA possono spedire e ricevere eventi.

Object Trade Service: è simile al Name Service, solo che permette di registrare e individuare i servizi offerti da un oggetto invece che la locazione via identificativo.

Time Service: servizio che abilita l'utente ad ottenere un'ora corrente condivisa da tutti gli oggetti CORBA.

Security Service: interfacce attraverso le quali si accede alle funzionalità di sicurezza come l'identificazione e l'autenticazione degli utenti, le autorizzazioni che essi hanno e strumenti come la firma digitale per ottenere la Non-repudiation[2].

Tali strumenti vengono solitamente distribuiti in moduli separati dall'ORB, e vengono denominati CORBA service.

2.2 Primo design

Lo scopo di questa sezione è illustrare a livello astratto come potrebbero essere utilizzate le funzionalità offerte da CORBA per implementare gli strumenti previsti in D-CaseLP.

2.2.1 Funzionalità da fornire nell'immediato

Innanzi tutto abbiamo cercato di definire un sottoinsieme di funzionalità da realizzare nel primo prototipo del tool D-CaseLP a partire dal sottoinsieme del Function Refinement Tree inerente a questa tesi (seconda parte del F.R.T. presentato in sezione 1.3.4). L'idea è evidenziare quelle che considereremo un nucleo di funzionalità indispensabili dal quale partire nella valutazione di come e quanto CORBA ci aiuti nella realizzazione del nostro progetto.

Commentiamo brevemente le funzionalità foglia evidenziate in grassetto nell'albero in figura 2.3, in modo da dare per ognuna qualche idea di come potrebbe essere ottenuta e se vi siano strumenti offerti da CORBA che ci aiutino nella loro realizzazione.

Documentazione per l'installazione e utilizzo.

È ovvio che se si vuole fornire uno strumento di una qualche utilità a terzi, è necessario inserire tra gli obiettivi di progetto di D-CaseLP la produzione di una documentazione ragionevolmente chiara su come si installano ed si utilizzano i pacchetti software che si produrranno. Nel caso in cui si decida effettivamente di realizzare il tool di esecuzione e simulazione in CORBA, si produrrà un manuale d'installazione in cui per ciò che riguarda l'installazione del run-time CORBA si farà riferimento al manuale utente della particolare implementazione utilizzata

[2] La Non-repudiation è il principio secondo cui un cliente che richiede un servizio, non può successivamente sostenere di non averlo richiesto.

1. Modellazione e creazione di codice per il prototipo del MAS.
   ... ...

2. Realizzazione e validazione di un prototipo della applicazione sotto sviluppo.

   (a) **Installazione degli strumenti offerti da D-CaseLP.**
      - Supporto all'installazione via procedure di installazione.
      - **Documentazione per l'installazione e l'utilizzo.**

   (b) **Implementazione prototipo del MAS.**
      i. Integrazione
         - Integrazione legacy software.
         - Integrazione semantica.
      ii. **Supporto alla comunicazione.**
      iii. Supporto alla mobilità.

   (c) **Monitoraggio e Simulazione dell'esecuzione del prototipo del MAS.**
      i. **Configurazione dell'esecuzione.**
         - **Come distribuire gli agenti.**
         - **Dove recuperare il codice degli agenti.**
      ii. **Monitoring grafico.**
         - Visualizzazione della distribuzione degli agenti.
         - **Visualizzazione delle comunicazioni fra gli agenti.**
      iii. Strumenti di simulazione.
         - Parametri della simulazione.
           - Ritardi sui canali di comunicazione o canali inaffidabili.
           - Input agli agenti in simulazione.
           - Distribuzione degli agenti durante la simulazione.
           - Gestione del tempo.
         - Supporto per la visualizzazione dei risultati.
           - Visualizzazione On-Line.
           - Visualizzazione Off-Line.

3. Implementazione della applicazione finita.

4. Utilizzo della applicazione finita.

Figura 2.3: FRT con evidenziate le funzionalità che saranno supportate dal primo prototipo di D-CaseLP.

(Orbacus), mentre per i package da noi prodotti si specificherà dove andranno collocati e come dovranno essere utilizzati in documentazione completamente nuova.

Supporto alla comunicazione.

La questione centrale fra le problematiche della tesi è come implementare la distribuzione degli agenti derivanti dalla specifica secondo la metodologia CaseLP o D-CaseLP. Abbiamo impostato il lavoro in modo da racchiudere l'interazione tra i livelli di specifica ed il livello di comunicazione in una unica interfaccia presentata in tabella 1.1; tale interfaccia nasce appunto come anello di congiunzione fra la specifica e l'esecuzione. La questione quindi si trasforma in come implementare le primitive di comunicazione appartenenti all'interfaccia. Si rimanda la discussione alla sezione 2.2.2.

Come distribuire gli agenti e dove recuperare il codice degli agenti.

Con queste funzionalità intendiamo la possibilità di indicare con quale codice e su quali macchine gli agenti devono essere attivati. CORBA non offre strumenti propri sull'attivazione di processi in remoto, dovremo quindi appoggiarci agli strumenti offerti dal sistema operativo sul quale le applicazioni verranno eseguite o programmare noi delle applicazioni che forniscano un "front-end"[3].

Visualizzazione delle comunicazioni fra gli agenti.

Anche per questa funzionalità non vi è supporto. Essendo CORBA un middleware per oggetti distribuiti non ci aspettavamo l'esistenza di un tool specifico per agenti, piuttosto, una qualche specifica standard per servizi di debugging delle chiamate ai servlet a partire dalla quale realizzare il visualizzatore delle comunicazioni tra agenti. L'inesistenza di specifiche standard, non esclude che le differenti implementazioni forniscano estensioni proprietarie. Non sembra che vi sia supporto in questa direzione nell'implementazione da noi candidata all'utilizzo. Rimandiamo una trattazione più dettagliata a quando avremo spiegato come sia possibile realizzare la comunicazione.

2.2.2 Come realizzare la comunicazione tra agenti via CORBA

Abbiamo visto quanto la comunicazione che in CORBA è di solito associata ad una richiesta di servizio, sottointenda un modello di tipo client-server. In questa

[3]Con l'espressione front-end intendiamo un'applicazione che coordini e faccia da interfaccia fra l'utilizzatore e le funzionalità precedentemente descritte.

ottica, implementare le primitive di comunicazione significa definire un'interfaccia IDL contenente la segnatura dei servizi forniti dall'oggetto CORBA; nel nostro caso rifacendoci a questo modo di impostare le cose, dovremmo definire un'interfaccia contenente le nostre primitive di comunicazione e prevedere differenti oggetti CORBA che permettano alle nostre specifiche di parlare fra di loro. In figura 2.4, abbiamo schematizzato un'architettura per la comunicazione nel caso di due agenti. In tale ipotesi avremo per ogni agente un *placeholder*: oggetto CORBA che offre le funzionalità di server di comunicazione, e si occupa di mantenere una mailbox con la quale l'agente potrà interagire.

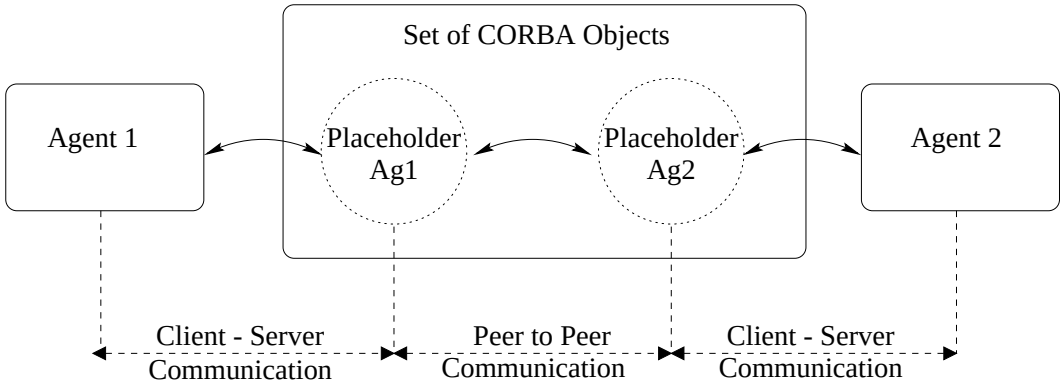

Figura 2.4: Un modo possibile di per far comunicare gli agenti con l'impiego di placeholder.

Assumiamo che l'argomento delle primitive di comunicazione siano messaggi espressi in un qualche Agent Comunication Language [22, 24], a grandi linee le tre primitive di comunicazione che abbiamo previsto potrebbero essere così progettate:

send(msg): l'agente che vuole spedire il messaggio msg lo passa al suo placeholder; quest'ultimo esamina il contenuto del messaggio ed in particolare estrae l'identificativo del destinatario; quindi contatta il placeholder del destinatario e gli consegna il messaggio.

receive_sync(msg): l'agente aspetta che il suo placeholder gli passi un messaggio con determinate caratteristiche (es. mittente o performative particolari). Il placeholder passa il messaggio all'agente in attesa prelevandolo dalla mailbox. Se non vi sono messaggi con le caratteristiche desiderate il placeholder aspetta che ne arrivi uno, nel frattempo l'agente rimane bloccato.

receive_async(msg): l'agente interroga il suo placeholder chiedendogli se vi è un messaggio con determinate caratteristiche. Se un messaggio con tali carat-

teristiche c'è viene restituito, comunque sia il controllo ritorna all'agente che prosegue la sua esecuzione.

Si osservi che una richiesta standard ad un oggetto CORBA si traduce in una esecuzione di un metodo o funzione e sarebbe quindi per sua natura sincrona. In realtà in CORBA sono previste tre tipologie di invocazioni sincrone:

Syncronous Invocation: il cliente invoca la richiesta e si blocca aspettando la risposta. Questa è la più comune tipologia di invocazione essendo supportata anche dall'invocazione statica.

Deferred Syncronous Invocation: il cliente invoca la richiesta e continua l'esecuzione mentre la richiesta viene eseguita, successivamente raccoglie le risposte. Questo tipo di invocazione permette di invocare più servizi ad elaborazione lunga contemporaneamente e raccogliere successivamente tutte le risposte invece che sequenzializzare le richieste.

Oneway Invocation: il cliente invoca un servizio e continua la sua esecuzione perché non vi è risposta da attendere.

La Deferred Syncronous Invocation è supportata solo da invocazioni dinamiche, ma l'OMG sta lavorando per renderla accessibile anche nelle invocazioni statiche. Nel caso delle primitive di comunicazione da noi definite la send potrebbe essere un'invocazione di tipo Oneway; la receive_sync(msg) potrebbe essere di tipo Syncronous Invocation; la receive_async potrebbe essere realizzata attraverso una Syncronous Invocation in cui nel caso non vi siano messaggi si restituisce il messaggio vuoto. I parametri delle due primitive di ricezione potrebbero essere passati per riferimento in modo da fungere anche da parametri di ritorno.
La comunicazione da agente a placeholder è di tipo client-server, mentre la comunicazione da placeholder a placeholder è di tipo **peer-to-peer**, in cui i due attori della comunicazione giocano a turno il ruolo di cliente e poi di servitore. Questa impostazione presuppone risolvere alcuni problemi, illustriamo con un esempio il primo. Consideriamo il caso in cui si stiano eseguendo tre agenti A,B,C su tre processori differenti, ed il placeholder di A stia comunicando con il placehoder dell' agente C, ed il placeholder dell'agente B voglia iniziare anch'esso una comunicazione con il placehoder di C. È evidente che il placehoder deve essere in grado di gestire concorrentemente più richieste, non è comunque pensabile che il placehoder B che deve consegnare un messaggio rimanga impegnato fino a che il placeholder di C non abbia terminato la comunicazione con A.
La situazione è aggravata dal fatto che le componenti placeholder possono giocare il ruolo di server per altri placeholder e anche per l'agente alla quale sono associati. La soluzione a questi problemi consiste nel strutturare il placehoder in Thread, uno per ogni richiesta. In CORBA è prevista la possibilità di strutturare i CORBA

object in modo che ad ogni chiamata venga automaticamente generato un thread, tale funzionalità è stata potenziata nel nuovo Object Adapter: POA [60, 34].

2.2.3 Configurazione degli agenti prototipali per l'esecuzione e simulazione

Una volta descritte le idee su come si possano sfruttare le funzionalità che CORBA offre circa la comunicazione, illustriamo come sia possibile aggiungere lo strato di comunicazione agli agenti prototipali risultanti dalle fasi di specifica. Innanzi tutto, osserviamo che l'aver creato specifiche sull'astrazione dell'interfaccia di comunicazione, rende possibile avvalersi di differenti implementazioni nelle fasi di sviluppo del MAS e come questo possa essere utilizzato all'interno del processo di sviluppo a vantaggio dell'utente. Consideriamo a titolo di esempio le seguenti implementazioni delle primitive di comunicazione:

Implementazione CORBA. Si fornisce uno strato di comunicazione basato su CORBA simile a quello descritto in precedenza.

Implementazione con monitoring. Si fornisce uno strato di comunicazione basato su CORBA o un'altra tecnologia in cui per ogni messaggio che spedisce l'agente, se ne spedisce una copia ad un agente preposto ad essere il monitor della comunicazione. In maniera analoga può essere ottenuto il monitoring per la ricezione di messaggi facendo spedire il messaggio copia alle receive piuttosto che nella send.

Implementazione fornita dall'utente. L'utente fornisce implementazioni da lui sviluppate in cui offre un'implementazione dell'interfaccia di comunicazione basata su tecnologie diverse da CORBA.

Ipotizzando di avere a disposizione le tre implementazioni precedenti delle primitive, si potrebbe impiegare l'implementazione con le primitive che permettono il monitoring nelle prime fasi di sviluppo, e successivamente una delle altre due.

Affinché l'utente trovi effettivamente vantaggio nella possibilità di intercambiare le diverse realizzazioni delle primitive, dovremo prestare attenzione ai seguenti fattori:

- impatto che l'aggiunta dello strato di comunicazione ha sul codice degli agenti prototipali;
- assunzioni che si sono fatte sul linguaggio col quale sono stati scritti gli agenti prototipali.

Circoscrivendo le modifiche che è necessario operare nel codice per cambiare l'implementazione della comunicazione diventa semplice fornire strumenti automatici che operino il cambiamento da un'implementazione all'altra. Allo stesso tempo, onde evitare di legare l'iterscabiabilità delle implementazioni solo ad agenti sviluppati in uno dei due linguaggi con i quali un agente può essere scritto (Java, o estensioni del Prolog), nel fornire una soluzione si dovrà fare attenzione a non utilizzare caratteristiche troppo peculiari di uno di essi. Prolog e Java supportano entrambi la programmazione modulare e offrono quindi la possibilità di organizzare il codice in più moduli. Si potrebbe sfruttare questa caratteristica per isolare la definizione delle primitive di comunicazione in moduli separati rispetto al codice degli agenti derivante dal processo di specifica. In questo modo la sostituzione dell'implementazione delle specifiche consisterebbe nella sostituzione del modulo che le implementa, e al limite, nel caso in cui non vi sia loading dinamico di moduli in una ricompilazione del codice. In figura 2.5 abbiamo schematizzato le componenti coinvolte nella comunicazione tra agenti.

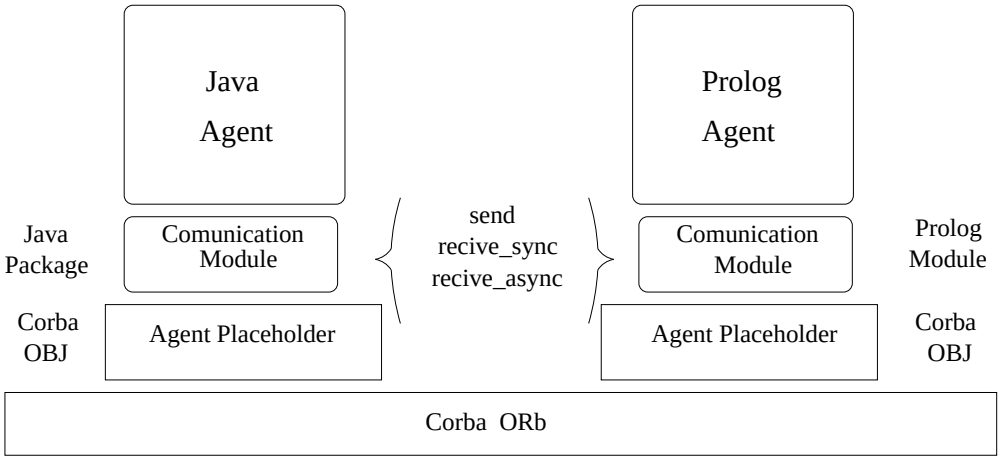

Figura 2.5: Integrazione delle primitiva di comunicazione negli agenti prototipali di D-CaseLP.

Dalla documentazione su CORBA difficilmente ci si rende conto che un programma, nel nostro caso un agente, per poter essere cliente di oggetti CORBA deve avere nel suo codice un qualche tipo di registrazione al ORB. Questo fatto ha conseguenze un po' più generali di ciò che ci si aspetterebbe, in quanto è presumibile che non sia una esigenza esclusiva di CORBA, la conseguenza più immediata è che prima di fare una qualsiasi operazione fra quelle dell'interfaccia di comunicazione l'agente deve registrarsi all'implementazione delle primitive. Nel caso specifico di CORBA abbiamo bisogno che l'agente si sia iscritto ai servizi dell'ORB e abbia risolto ciò che è il riferimento logico standard del ORB con un riferimento fisico. La questione a questo punto diventa come e dove inserire il

codice relativo alla connessione al strato di comunicazione nel codice dell'agente. Le istruzioni per la connessione al ORB o al generico layer di comunicazione, potrebbero essere inserite nel codice degli agenti in uno dei seguenti modi:

- fornendo un traduttore che trasforma gli agenti prototipali in agenti da simulazione;
- estendendo l'interfaccia di comunicazione con una primitiva di iscrizione al livello di comunicazione sottostante gli agenti.

La prima delle due alternative prevede la manipolazione automatica del codice degli agenti, e l'inserimento di un "cappello" nel codice in cui vengono scritte le istruzioni o i predicati necessari alla connessione con il servizio di comunicazione.[4]
La seconda ipotesi è più elegante in quanto non buca l'astrazione fornita dall'interfaccia delle primitive di comunicazione, ma allo stesso tempo delega all'utente la responsabilità di effettuare la connessione con il layer di comunicazione attraverso una primitiva di "iscrizione" ai servizi di comunicazione.

2.2.4 Funzionalità aggiuntive

Indichiamo qui, un insieme di funzionalità che non sono state previste nel F.R.T ma che possono aiutarci a fornire un tool più versatile, in cui ad esempio sia possibile aggiungere a MAS già progettati nuove istanze di agenti. Supponiamo di avere un'"ambiente" in cui gli agenti vivono con un qualche identificativo associato. Indipendentemente da chi e come sia fornito questo ambiente (sistema operativo, un platform, un middleware) i seguenti servizi si possono includere in esso:

Directory Service: funzionalità che permette di risolvere l'identificativo simbolico associato ad un agente con l'indirizzo "fisico" dell'agente.

Yellow Pages Service: funzionalità che permette di individuare, data la descrizione di un certo servizio, l'identificativo degli agenti che offrono tale servizio.

Queste funzionalità non compaiono all'interno del F.R.T. su cui abbiamo ragionato fino ad ora, in quanto potrebbero anche non essere fornite dalle implementazioni di D-CaseLP. Si potrebbe infatti pensare di demandare la loro progettazione all'utente di D-CaseLP, che di volta in volta include nella sviluppo del proprio MAS anche due agenti che implementino le funzioni di directory service e yellow

[4] Si dovrà pensare anche all'inserimento del codice di sconnessione.

pages service.
In realtà se si considerano i domini applicativi e le problematiche che normalmente ci si propone di risolvere con la tecnologia agenti, vi sono buone motivazioni per includere nel nostro tool la progettazione di entità che forniscano tali servizi. Illustriamo le motivazioni utilizzando un semplice esempio in cui vi sono due agenti $\mathcal{A}$ e $\mathcal{B}$. All'attivazione ognuno di essi si registra al servizio di connessione, dopo di che $\mathcal{A}$ spedisce un messaggio a $\mathcal{B}$. Questo esempio è molto semplice ma in esso abbiamo fatto già alcune assunzioni:

- $\mathcal{A}$ conosce l'esistenza di un agente $\mathcal{B}$;
- $\mathcal{A}$ conosce l'"indirizzo" di $\mathcal{B}$.

La prima delle due assunzioni significa che a "tavolino" si è deciso che nel nostro MAS vi sarà l'agente $\mathcal{B}$, ed esso non nasce né muore improvvisamente ma è componente costante dell'architettura del MAS. In altre parole abbiamo presupposto che l'architettura del MAS si componga di un numero di istanze di agenti prefissato. La seconda delle assunzioni indica che qualcuno ha passato l'indirizzo attuale di $\mathcal{B}$ ad $\mathcal{A}$, tale indirizzo è stato presumilmente passato in input ad $\mathcal{A}$ dopo l'inizio dell'esecuzione di $\mathcal{B}$. Nel caso in cui si disponga di un directory service almeno la seconda assunzione decade, infatti l'agente $\mathcal{A}$ può chiedere al directory service qual è l'indirizzo attuale di $\mathcal{B}$. La presenza di un yellow pages service ci aiuta ad evitare la prima ipotesi, che in realtà sottointende anche la conoscenza del fatto che esiste un agente di nome $\mathcal{B}$, col quale l'agente $\mathcal{A}$ deve interloquire al fine di arrivare ai suoi obiettivi. Con l'introduzione del yellow pages service non è più necessario che $\mathcal{A}$ conosca l'esistenza di un agente di nome $\mathcal{B}$: gli basta sapere quale tipo di servizio necessita e sulla base di esso interrogare il yellow pages server circa chi fornisce tale servizio.
Facciamo notare quanto le due assunzioni possano essere limitanti, in particolare se si pensa ai contesti in cui il paradigma ad agenti dovrebbe trovare maggiore applicazione, contesti caratterizzati dalla necessità di un buon livello di scalabilità in cui spesso vi è la necessità di creare dinamicamente agenti in funzione dell'evolversi della computazione.

Alle considerazioni precedenti vanno unite alcune osservazioni derivanti dagli obiettivi che si pone D-CaseLP. D-CaseLP vuole essere prima di tutto un ambiente che faciliti la prototipazione e lo sviluppo rapido di sistemi multi-agenti. In questo senso la presenza di un directory service ed un yellow pages service non può che essere vista positivamente. Allo stesso tempo fornire queste due componenti può significare aumentare la mole di lavoro in maniera notevole. Se vogliamo mantenere l'interscambiabilità delle diverse implementazioni delle comunicazioni, bisogna fornire un directory server e uno yellow server pages che

non dipendano dal tipo di implementazione del livello di comunicazione e offrano ad esempio un formato di indirizzi fisici per gli agenti abbastanza generico da includere indirizzi di posta elettronica fino ad indirizzi IIOP.

Ciò che dobbiamo chiarire in questa fase di valutazione circa l'effettivo vantaggio che si ha nell'adozione di CORBA ai nostri scopi, è quanto i servizi in esso offerti, quali Name Service e Object Trader Service, possano risultare utili alla costruzione di un directory server o un yellow page server.

Name Service e Directory Service.

Per sfruttare il name service come Directory service è necessario associare un agente ad un oggetto. Si è molto discusso sulla differenza che intercorre tra oggetto ed agente e non è nostra intenzione rimettere in discussione il fatto che le due cose sono profondamente differenti. Ciò che è importante tenere presente è che questa differenza non impedisce di sviluppare sistemi ad agenti con linguaggi orientati agli oggetti. Se si individua un oggetto che faccia da rappresentante dell'agente, ad esempio il placeholder nel nostro caso, non vi dovrebbero essere grosse difficoltà nell'utilizzo del Name Service come directory service.

Object Trader Service e Yellow Pages Service.

L' Object Trade Service fornisce gli strumenti che permettono agli oggetti CORBA di importare ed esportare funzionalità. L'idea si basa sull'esistenza di un Trade Object che fa da intermediario tra gli oggetti. Quando un oggetto CORBA vuole pubblicizzare un suo servizio tramite l'operazione di esportazione dà una descrizione al Trade Object del servizio che vuole esportare. Se un oggetto CORBA vuole invece importare una funzionalità fornisce al Trade Object una descrizione del servizio di cui necessita; l'intermediario applicando un qualche criterio di similarità con la descrizione che ha ricevuto restituisce i fornitori che dichiarano di offrire tali servizi. Il fulcro della discussione a questo punto si sposta sul formato della descrizione dei servizi. Le funzionalià offerte da CORBA sono pensate per oggetti e non per agenti; ci serve capire quanto riusciamo ad adattare la descrizione di un servizio di un oggetto CORBA con i servizi di un agente.

A questo scopo illustriamo la descrizione di un servizio nella specifica del Trade Object Service:

- una interfaccia che definisce un segnatura computazionale del servizio;
- zero o più "named property types" che rappresentano il comportamento del servizio catturando gli aspetti non computazionali che non sono esprimibili nella segnatura.

Una property type è definita come una tripla (nome,tipo,modalità) dove la modalità esprime se la proprietà che descrive è obbligatoria od opzionale, eccetera.

È bene ricordare che si accede ad un servizio fornito da un oggetto tramite una chiamata a metodo del oggetto, mentre la richiesta di servizio per agenti avviene tramite una comunicazione fra gli agenti espressa in un qualche Agent Comunication Language. Un messaggio espresso in un qualche agent comunication language conterrà informazioni quali il destinatario, il tipo di azione linguistica che è contenuta nel messaggio (performative), il corpo del messaggio, un'ontologia che associa una semantica ai termini nel corpo del messaggio.

La prima osservazione che si può fare è che un servizio di un agente potrebbe non avere una segnatura, o almeno non avere una unica segnatura. È chiaro che a livello astratto si potrà associare ai servizi offerti dagli agenti delle funzioni e quindi una qualche segnatura, ma non è detto che l'associazione sia così immediata come per gli oggetti. Ciò ci dovrebbe far intuire che vi sarà qualche difficoltà nell'adattare Object Trade Server ai nostri scopi.

2.2.5 Considerazioni di valutazione

È evidente che le idee che abbiamo fornito richiedano ancora molto lavoro per essere realizzate, vi sono alcuni punti nella progettazione che sono rimasti aperti. Tuttavia, la progettazione fornita è sufficiente ad effettuare le considerazioni di valutazione che seguono sull'impiego di CORBA per realizzare la distribuzione degli agenti D-CaseLP. Partiamo da quelli che sono tradizionalmente considerati i punti di forza di CORBA: comunicazione di componenti distribuiti e eterogeneità delle componenti.

Comunicazione di componenti distribuiti.

Indipendentemente dai vincoli progettuali imposti da D-CaseLP la comunicazione tra oggetti o generiche componenti software è profondamente differente dalla comunicazione fra agenti. La prima è basata su una logica **client-server** in cui vi è un cliente che richiede un servizio ad un servitore. Non vi è interpretazione ne da una parte ne dall'altra della comunicazione ed il servitore non può rifiutare di servire una richiesta. La seconda utilizza una logica di tipo **peer-to-peer** in cui non vi sono né servitori né clienti ma due entità paritarie che "discutono" e collaborano per ottenere un obiettivo. Il paradigma ad agenti presuppone una comunicazione simile a quella umana che a differenza di quella fra oggetti coinvolge tre livelli di astrazione: trasporto, linguaggio di comunicazione per agenti ed ontologie[5].

[5]Nel caso della comunicazione umana i tre livelli di astrazione potrebbero essere: il mezzo di comunicazione (una lettera, la voce, ecc.), la lingua utilizzata (italiano, tedesco, inglese, ecc.),

Qualora si riesca ad ottenere una comunicazione peer-to-peer in CORBA seguendo la progettazione che abbiamo illustrato si otterrebbero solo gli strumenti per effettuare il livello di astrazione più basso della comunicazione tra agenti: il trasporto.

La comunicazione offerta da CORBA è utile alla realizzazione della comunicazione tra agenti solo per il livello di astrazione più basso: il trasporto.

Eterogeneità delle componenti.

In D-CaseLP vogliamo integrare nell'esecuzione del MAS due tipologie di agenti Ag-Prolog ed Ag-Java rispettivamente scritti in Prolog ed in Java.

Uno dei motivazioni che hanno portato alla valutazione di CORBA per la distribuzione di agenti in D-CaseLP è la possibilità che esso offre di integrare componenti scritte in differenti linguaggi di programmazione. L'OMG specifica l'associazione tra l'Interface Definition Language (IDL) e vari linguaggi di programmazione fra cui C, C++, Java, Lisp. Purtroppo, essendoci alcuna definizione di questo tipo per Prolog, non è possibile integrare Prolog e Java attraverso CORBA rimanendo all'interno delle specifiche OMG.

In [61] è indicata una strada proprietaria per integrare CORBA e SWI-Prolog, una particolare implementazione di Prolog. Tale integrazione sembra offrire un compilatore IDL basato sulla possibilità di far interagire Prolog con il C. Al di la del fatto che vi sono degli elementi che sembrano legare la soluzione proposta a particolari implementazioni CORBA, non è stato possibile comprendere dalla documentazione l'effettivo stato di realizzazione degli strumenti descritti. Per questi motivi si ritiene utile valutare altre strade al fine di ottenere l'integrazione dell'esecuzioni degli agenti Ag-Prolog e Ag-Java.

2.2.6 Conclusioni

L'analisi di CORBA e della sua applicabilità nel realizzare i nostri obiettivi ha richiesto un certo investimento di risorse, quindi riteniamo di dover concludere illustrando schematicamente le ragioni che hanno portato all'abbandono di questa strada, in modo che altri di fronte a necessità simili alle nostre possano trarre vantaggio dalla nostra esperienza. Abbiamo suddiviso le motivazioni che ci hanno portato a non adottare CORBA in due categorie: inerenti al dominio degli agenti e dipendenti dal progetto D-CaseLP. Le inerenti al dominio degli agenti sono i problemi e le difficoltà che si incontrano nel cercare di utilizzare CORBA indipendentemente dalle ipotesi del nostro progetto per il solo fatto che si sta utilizzando uno strumento che non è un middleware per agenti, mentre le dipendenti

l'accezione dei termini che vengono utilizzati nella discussione.

dal progetto D-CaseLP sono le difficoltà e gli impedimenti che sono da mettersi in relazione alle particolari ipotesi che abbiamo fatto.

- Inerenti al dominio degli agenti.

  Modello di comunicazione. CORBA offre una comunicazione che è basata sul modello client-server e non sul peer-to-peer.

  Astrazione della comunicazione. La comunicazione per gli agenti coinvolege differenti livelli di astrazione, CORBA è utilizzabile solo per il livello di trasporto, per offrire un mezzo di comunicazione.

- Dipendenti dal progetto D-CaseLP.

  Integrazione agenti Ag-Prolog e Ag-Java. Come abbiamo spigato nella sezione precedente non c'è possibilità di utilizzare CORBA per ottenere un'integrazione portabile fra agenti scritti in Java ed agenti scritti in Prolog.

  Considerazioni sulle risorse con cui affrontiamo il progetto. Vista la mole di lavoro necessaria per raggiungere solo parte degli agli ambiziosi obiettivi che ci siamo posti, è giusto valutare bene le conseguenze di una scelta come quella che si prospetta essere CORBA. Con le attuali risorse di forza-lavoro di cui disponiamo partire da una distribuzione basata CORBA significa rischiare di avere i primi risultati quando saranno obsoleti.

Dalle osservazioni precedenti si comprende che il supporto che CORBA offre in ambito di distribuzione ed integrazione di componenti eterogenene, diventano piuttosto limitato nel contesto della distribuzione degli agenti in particolare sotto le ipotesi da noi fatte. Per questo abbiamo deciso di scegliere una politica di progetto differente da quella con la quale eravamo partiti, ed adoperarci nel recupero di componenti offerte da terzi.

Capitolo 3

FIPA

In questo capitolo presenteremo quella che è iniziativa di standardizzazione FIPA, illustrando gli obiettivi che essa si prefigge e parti delle specifiche che sono state proposte. La iniziativa della FIPA hanno reppresentato un punto di riferimento di fondamentale importanza per questo lavoro di tesi: la progettazione di D-CaseLP è stata influenzata in modo sostanziale dalle specifiche che presenteremo in questo capitolo ed in particolare dall'architettura astratta [28]. Il capitolo può essere suddiviso in due parti: la prima illustra la struttura, gli obiettivi ed il modo di lavorare dell'organizzazione; la seconda fornisce una panoramica su un sottoinsieme delle specifiche prodotte dalla FIPA inerente agli scopi di questa tesi. Per concludere illustreremo brevemente alcuni progetti di validazione delle specifiche FIPA ed alcune nostre valutazioni sull'iniziativa. Si tenga conto fin da subito che la standardizzazione FIPA sta riscuotendo molto interesse sia in ambito accademico che in ambito industriale. Tale sforzo congiunto potrebbe far diventare appetibile la tecnologia ad agenti per sviluppare applicazioni "reali". Anche nel caso in cui ciò non accadesse, il lavoro della FIPA avrebbe comunque contribuito a mettere in evidenza problematiche sulle quali focalizzare la ricerca.

3.1 Introduzione alla FIPA

La FIPA [23], acronimo di Foundation for Intelligent Physical Agents, nasce nel 1996 con lo scopo di promuovere attraverso la produzione di specifiche standard liberamente fruibili, la diffusione delle tecnologie basate su agenti software e hardware. Tale organizzazione è composta da un certo numero di gruppi di ricerca ed un forum di aziende, fra le quali alcune operanti nel dominio delle telecomunicazioni. Giusto per dare una idea della portata dell'organizzazione e dell'interesse che ha suscitato, si può dire che a settembre del 2000 la FIPA vedeva tra i suoi soci rappresentanti di circa 65 società, tuttora vi sono associati rappresentati di aziende

come British Telecommunications, France Telecom, IBM, Hitachi, Intel, Motorola, Siemens, Sun, Toshiba. Gli obiettivi di tale organizzazione sono esplicitati nel "mission statement" costitutivo che riportiamo in figura 3.1[1].

La promozione delle tecnologie e specifiche per l'interoperabilità che permettano l'interoperazione fra sistemi ad agenti intelligenti in contesti commerciali e industriali moderni.

Figura 3.1: Mission statement costitutivo della FIPA.

Nonostante i propositi iniziali che prevedevano la produzione di specifiche per agenti hardware (Robots) e software, la FIPA ha finito col concentrarsi nella produzione di specifiche per agenti del secondo tipo. La forte attenzione prestata dalla FIPA alle problematiche legate all'interoperabilità e comunicazione degli agenti, ha indotto alcuni ha ribattezzare l'acronimo dell'organizzazione in Foundation for InteroPerable Agents.

Nelle sottosezioni successive, analizzeremo alcuni dei fattori che hanno portato all'intento di standardizzazione, illustrando poi i principi e la portata del processo sulla base delle osservazioni evidenziate in [52] ed in [14].

3.1.1 Fattori dai quali scaturisce lo standard

Descriviamo quelli che sono i due fattori che hanno portato all'iniziativa di standardizzazione di cui si è fatta carico la FIPA.

Molti framework differenti e visioni poco conciliabili.

Le ricerche sugli agenti iniziarono molti anni addietro al 1995, in quegli anni però, la situazione che si andava delineando vedeva da una parte una rapida crescita dell'interesse intorno agli agenti, dall'altra l'emergere di una varietà di framework con approcci e visioni differenti che davano luogo a sistemi incapaci di interagire tra loro.

Oltre il confine della ricerca accademica.

Al di fuori di quelli che sono gli ambienti di ricerca spesso vi è una certa resistenza ad adottare tecnologie o "Standard" in cui non vi sia l'accordo di un certo numero di compagnie. Per far si che il paradigma ad agenti prendesse piede era necessario che vi fosse un certa legittimazione da parte degli ambienti industriali. La

[1] Riportiamo qui anche la frase originale: "The promotion of technologies and interoperability specifications that facilitate the end-to-end interworking of intelligent agent systems in modern commercial and industrial settings".

standardizzazione portata avanti dalla FIPA è frutto del crescente interesse dimostrato negli ambiti industriali per il paradigma ad agenti ma è anche un tentativo di ottenere questa legittimazione[2].

3.1.2 Assunti e principi nella produzione FIPA

Descriviamo i principi di cui è permeata la produzione FIPA, avremo modo di osservare quanto questi si riflettano a vari livelli nelle specifiche e nelle scelte progettuali della FIPA.

La prima cosa che bisogna chiarire è che la FIPA si propone di produrre specifiche per sistemi multi-agente "aperti", ovvero sistemi che godono della proprietà di **openness**. Definiamo tale proprietà a partire dalla descrizione presentata in [14]:

Definizione 3.1 *La* **openness** *è una proprietà che possono avere i MAS che permette agli agenti di unirsi o lasciare un MAS a tempo di esecuzione senza che sia necessaria alcuna ricompilazione o una riconfigurazione del sistema.*

Per illustrare l'utilità della proprietà riconsideriamo l'esempio del sistema per il controllo del traffico aeroportuale di cui avevamo parlato in 1.3.2. Nell'esempio in questione si aveva un agente di classe Aircraft associato ad ogni aeroplano che voleva atterrare nell'aeroporto di cui il MAS gestisce il traffico. In tale contesto è presumibile che per ogni nuovo aeroplano che entri nella zona controllata dall'aeroporto si crei un agente Aircraft che contratti l'atterraggio dell'aereo con la torre di controllo. Allo stesso tempo, quando un aereo lascia la zona controllata dall'aeroporto, l'agente Aircraft dell'aereo in questione uscirà dal MAS. È evidente che la creazione e la cancellazione di agenti deve avvenire dinamicamente, non si può pensare di riavviare il sistema ogni volta che cambia l'insieme degli aerei con cui la torre di controllo interagisce. È dunque questo un esempio in cui torna utile supportare la proprietà di openness.

Il secondo principio alla base delle specifiche FIPA potrebbe essere denominato principio di **minimalità**. La minimalità è intesa ad evitare che le specifiche prodotte si leghino ad un particolare hardware, sistema operativo o linguaggio di programmazione. Con il principio di minimalità si vuole assicurare che le specifiche offrano una certa genericità e astrazione, in modo da fornire un modello che possa essere una visione comune a diversi approcci, una specie di "fondamento" dal quale partire.

[2] Si tenga presente che la standardizzazione è ancora in piena corsa d'opera, e che una piena legittimazione ha come prerequisito l'esistenza di uno standard.

3.1.3 La portata delle specifiche FIPA

Quando ci si propone di operare un processo di standardizzazione è importante individuare fino a dove si spinga tale intento: quale sia il suo confine. Nel caso degli agenti abbiamo un dominio molto ampio e vario, se potessimo pensarlo come uno spazio geometrico avrebbe probabilmente parecchie "dimensioni", ognuna rappresenterebbe una proprietà con cui si possono caratterizzare i MAS. Non è semplice capire se tutte queste dimensioni descrivono uno spazio "ortogonale" e di conseguenza se sono un buon sistema di riferimento per caratterizzare tutto lo spazio, comunque vincolare alcune di queste dimensioni a precisi intervalli ci aiuta a porre dei limiti nel campo di azione del processo di standardizzazione.

Intelligenza esterna.

La FIPA si propone di standardizzare la comunicazione esterna agli agenti evitando di trattare come gli agenti possano processare i dati ottenuti dalla comunicazione[3]. Ciò significa che la FIPA si limita ad occuparsi dei problemi legati al macro-livello[4] degli agenti, evitando di fare assunzioni o fornire specifiche sul micro-livello dell'agente, o peggio, legarsi ad una particolare architettura per agenti.

Neutralità.

Viene applicato il principio di minimalità di cui si discuteva prima anche per capire il livello di astrazione che deve essere supportato dalle specifiche FIPA. È negli intenti dell'organizzazione mantenere le specifiche il più possibile generali e neutrali rispetto a

- servizi specifici[5];
- impiego strumenti software, infrastrutture hardware e linguaggi d'implementazione.

[3]Non si occupa del micro-livello, anche se alcuni ad esempio in [14] hanno fatto notare che la semantica di FIPA ACL è stata fornita in termini di credenze (belief), desideri (desire) e intenzioni (intention) e questo potrebbe indurre a fare confusione su questo punto.

[4]Il macro-livello nella progettazione dei sistemi multi-agente si riferisce alle problematiche che si occupano dell'iterazione, delle ralazioni e della concorrenza tra gli agenti appartenenti ad uno stesso MAS. Il termine micro-livello, invece, si riferisce alle problematiche inerenti al singolo agente, i comportamenti e le attitudini di cui deve essere dotato, l'architettura più adatta per esprimere le sue attività.

[5]Deve essere possibile aggiungere servizi dipendenti dal problema e dal dominio applicativo.

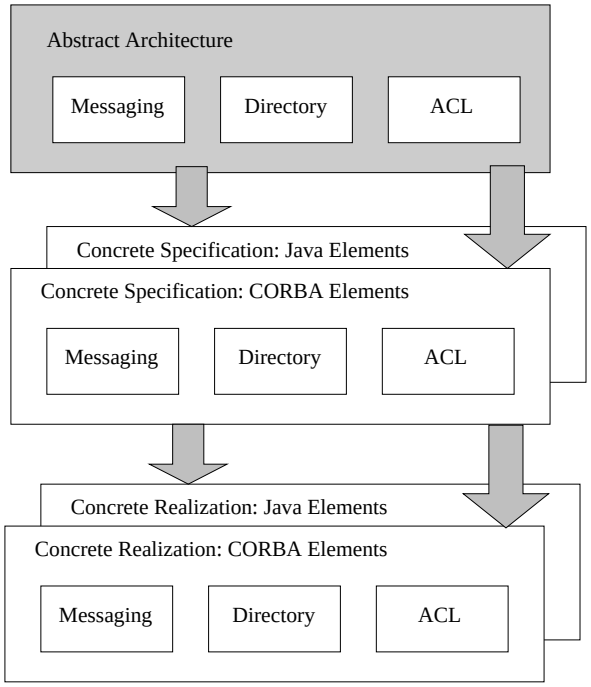

Figura 3.2: L'architettura astratta e le sue realizzazioni (specifiche di realizzazione).

Per assicurare la neutralità viene specificata un'archittetura astratta (Abstract Architecture) [27] che fornisce un modello in cui sono presentati ad alto livello i seguenti insiemi di servizi:

Directory Service. Insieme di servizi che offrono funzionalità che permettono agli agenti di conoscere caratteristiche relative ad altri agenti. Fra le caratteristiche che è possibile conoscere vi sono gli indirizzi con cui è possibile contattare gli agenti e i servizi da essi offerti[6].

Message Transport Service. Insieme di servizi che forniscono le funzionalità necessarie affinché i messaggi vengano spediti e consegnati a destinazione indipendentemente dalle caratteristiche ed il tipo di rete sulla quale avviene la comunicazione.

Agent Communication Language. Insieme di specifiche che definiscono le funzionalità ed i formati che permettono di organizzare i messaggi che si scambiano gli agenti.

[6]Si noti che a questo livello di astrazione non è particolarmente importante distinguere il servizio di pagine bianche da quello di pagine gialle.

Si noti quanto queste componenti astratte siano tutte componenti in qualche maniera orientate alla comunicazione. Non è possibile implementare direttamente la specifica astratta ma a partire da essa è possibile dare una "specifica concreta" nella quale si fornisce l'associazione fra le componenti astratte e le tecnologie con cui possono essere realizzate. Poi a partire da una specifica concreta si potrà implementare una realizzazione concreta. In figura 3.2 si esplicitano visivamente le relazioni tra architettura astratta (Abstract Architecture), specifica concreta (Concrete Specification) e realizzazione concreta (Concrete Realization).

Recupero di tecnologie preesistenti.

Un altro elemento per descrivere la portata delle specifiche consiste nell'individuare diversi livelli di cui un MAS può essere composto e capire per quali livelli si voglia fornire una standardizzazione e per quali esista già una tecnologia consolidata. Nel caso in cui sia possibile impiegare una tecnologia già esistente uno dei problemi diventa come gli agenti possano essere uniti alle infrastrutture preesistenti.

Non tutte le componenti sono agenti.

Si potrebbe essere indotti a pensare che i componenti che offrono servizi agli agenti debbano essere anch'essi agenti. Non è necessariamente così. In alcune delle prime specifiche prodotte dalla FIPA, il *Message Transport* era specificato come un agente. Questo tipo scelta pur essendo molto flessibile risultava inefficiente. Spieghiamo dove fosse l'inefficenza illustrando i passi della comunicazione fra un agente "sender" ed un agente "receiver" impiegando un agente message transport.

1. L'agente "sender" passa all'agente "message transport" un messaggio destinato all'agente "receiver".

2. L'agente "message transport" comunica il messaggio spedito dal sender al receiver.

L'inefficienza consisteva nell'impiego di due comunicazioni ogni qualvolta se ne doveva effettuare una. Si può eliminare tale problema implementando il Message Transport come un modulo software accessibile attraverso una Application Program Interface (API), piuttosto che come agente.

L'esempio precedente illustra come a volte non sia conveniente esprimere le componenti dell'infrastruttura[7] in termini di agenti e sia invece necessario prevederne alcune nella forma di moduli e oggetti. Poiché le componenti con cui si

[7]Con infrastruttura si intende come nel resto del capitolo se non specificato altrimenti, l'insieme delle componenti che offrono servizi di "ambiente" agli agenti.

interagisce ad un certo livello di granularità non sono più agenti, si cerca di recuperare pattern risolutivi proposti in contesti diversi dal dominio del paradigma ad agenti. Da qui un ulteriore parametro per decidere quali siano le problematiche di cui la FIPA si deve occupare: quando il livello di dettaglio esce da ciò che è il paradigma ad agenti e ci si addentra in qualche altra tecnologia le specifiche FIPA non possono andare molto oltre che fornire la standardizzazione delle interfacce con quale accedere alle funzionalità delle componenti sottostanti. Se non si facesse così si legherebbero le specifiche a qualche tipo di tecnologia o tecniche di implementazione, perdendo parte della neutralità e contravvenendo al principio di minimalità.

3.1.4 Organigramma della FIPA

Forniamo qui qualche informazione su come la FIPA è strutturalmente organizzata. Le informazioni circa la struttura della FIPA ci aiuteranno a capire il ciclo di vita che seguono le specifiche ed è comunque interessante di per sè. In figura

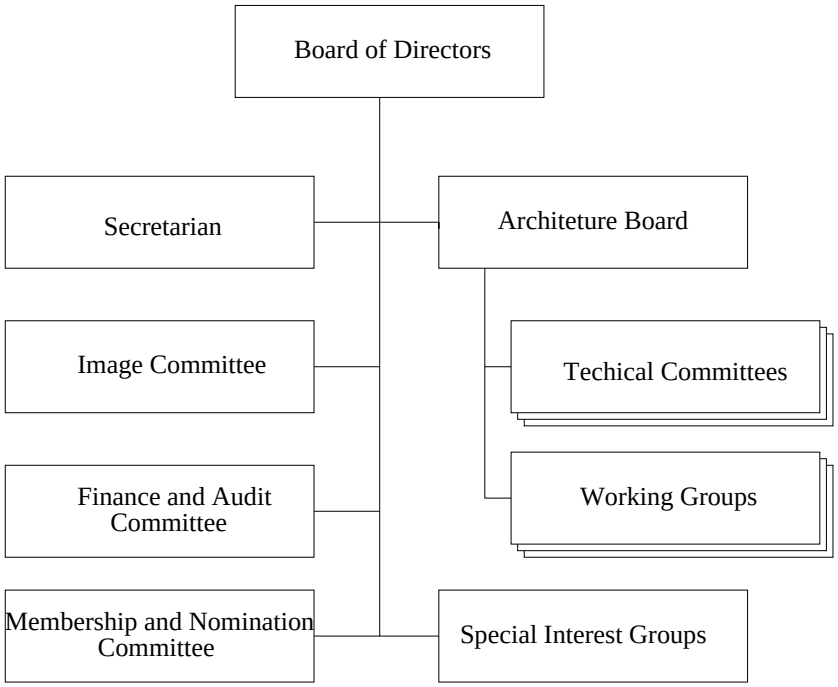

Figura 3.3: La struttura della FIPA.

3.3 si può osservare la suddivisione dell'organizzazione, di seguito forniamo una breve descrizione dei compiti di ognuna:

Board of Directors: i direttori responsabili della gestione e della condotta della FIPA, decidono sulle politiche e le procedure che la FIPA adotterà, definiscono la direzione del lavoro che dovrà essere intrapreso.

Secretariat: si occupa della gestione delle adesioni e degli aspetti finanziari della FIPA.

Image Committee: coordina le attività di promozione delle attività della FIPA e gestisce la news letters "FIPA Inform!".

Financial and Audit Committee: si occupa della revisione dei conti e prepara i rapporti di verifica per il Board of Directors.

Membership and Nomination Committee: analizza le qualifiche di ogni candidato membro, propone le nomine per l'elezione del Board of Directors facendo attenzione di assicurare un certo bilanciamento geografico nell'assegnazione delle cariche.

Architecture Board: l'autorità responsabile della consistenza, accuratezza, ed adeguatezza del lavoro tecnico intrapreso dalla FIPA, sovraintende al lavoro dei Technical Committees e dei Working Groups.

Technical Committees (TCs): sono diversi comitati ognuno specializzato su un determinato argomento. Portano avanti il lavoro tecnico della FIPA secondo un piano approvato dalla Architecture Board. Il loro lavoro consiste sia nella creazione di nuove specifiche sia nella modifica delle preesistenti.

Working Groups (WGs): sono progettati per portare avanti altri aspetti del lavoro della FIPA, non necessariamente tecnici. Possono avere il compito di concentrarsi su una precisa applicazione o essere responsabile dell'interazione con altri standard. Di solito un WG crea documenti informativi o propone modifiche a specifiche esistenti.

Special Interest Groups (SIGs): vengono costituiti per portare avanti lavoro ausiliario sul quale vi sia un particolare interesse da parte dei membri.

3.1.5 Il processo di produzione dello standard

La FIPA mette a disposizione di chi vuole il proprio lavoro rendendolo accessibile via Internet [23]. In questo modo si ha la possibilità di avere i documenti in differenti fasi di lavorazione, non solo standard ma anche documenti preliminari o sperimentali. Se si vuole trarre il maggior vantaggio possibile dall'impiego delle specifiche FIPA è importante saper distinguere lo stato di avanzamento di ognuna

di esse. Anche per aumentare la trasparenza nel 2000 è stato ufficialmente adottato un iter che regola l'avanzamento del lavoro contenuto nei documenti accessibili, individuando un vero e proprio "life cycle" per le specifiche FIPA. In [52], si trova la descrizione del life-cycle che riportiamo in figura 3.4.

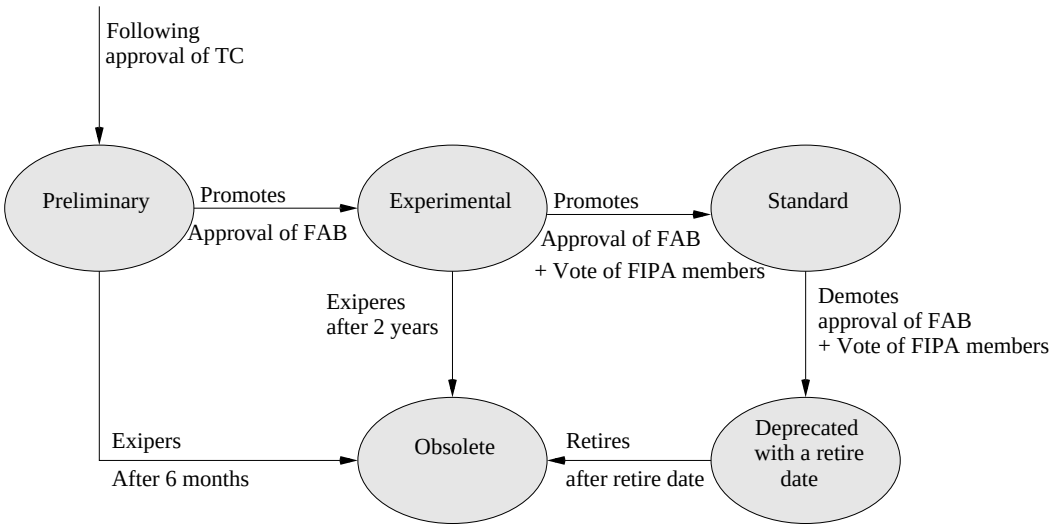

Figura 3.4: Il lyfe-cycle delle specifiche FIPA.

In figura sono schematizzati i vari stati in cui una specifica si può trovare e i meccanismi che portano una specifica a diventare standard ed eventualmente obsoleta. Una specifica inizia la sua vita nello stato preliminare (Preliminary) quando un comitato tecnico (TC: Technical Committee) crea una bozza di specifica. Nel momento in cui il comitato decide che la bozza è pronta per essere implementata la specifica entra nello stato sperimentale (Experimental), dove può rimanere per un periodo massimo di due anni. Se la sperimentazione porta a buoni risultati la specifica può diventare standard e venire considerata a tutti gli effetti stabile e pronta per essere implementata dalle aziende o organizzazioni che lo vogliano. Nel caso in cui a specifica diventi inutile o venga "sorpassata" da nuova specifica viene messa nello stato deprecated per un periodo di sei mesi. In questi sei mesi chi avesse adottato la specifica divenuta deprecated ha il tempo di aggiornare le proprie implementazioni secondo le nuove specifiche. Si osservi che allo stato attuale non vi sono ancora specifiche che abbiano raggiunto lo stato di standard ed i lavori in fase più avanzata sono nello stato Experimental.

3.2 Organizzazione delle specifiche FIPA

Dopo aver presentato l'organizzazione, i principi, e l'iter di specifica adottatati dalla FIPA, diamo una breve presentazione del lavoro di specifica da essa effettuato. L'insieme delle specifiche FIPA è abbastanza vasto quindi ci limiteremo a descrivere solo le parti che sono vicine alle problematiche che dobbiamo affrontare in D-CaseLP. Rimarchiamo ancora una volta che il fine delle specifiche non è descrivere come dovrebbero essere realizzate le componenti dell'infrastruttura per sistemi multi-agente, ma fornire le interfacce delle funzionalità attraverso le quali le diverse componenti di tale infrastruttura possono interagire. Le specifiche sono organizzate su differenti livelli di granularità, in questo modo ciò che nella specifica di livello più astratto viene definito come una componente che fornisce determinate funzionalità, nella specifica a granularità più fine viene suddiviso in diverse componenti le cui funzionalità vengono descritte con un maggior dettaglio. In figura 3.5 si possono veder tre livelli (layer) di astrazione in cui sono

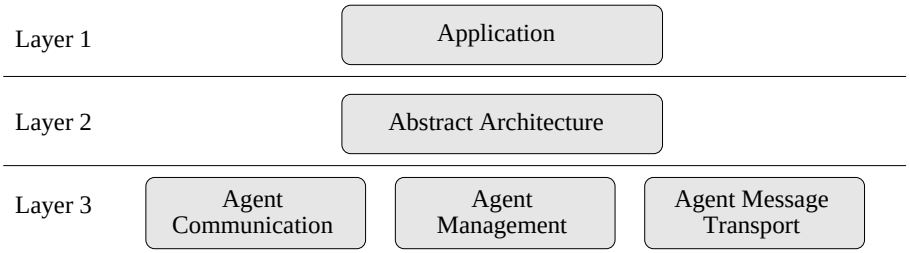

Figura 3.5: Le classi di specifiche della FIPA.

organizzate le classi di specifiche. Per alcune delle classi del terzo livello vi è una ulteriore suddivisione che offre una granularità ancora più fine. Descriviamo nel seguito le classi e le sottoclassi di specifiche.

Application. Specifiche sull'analisi e il design di applicazioni la cui soluzione potrebbe essere sviluppata mediante la tecnologia ad agenti. Vengono fornite specifiche sulle funzionalità che dovrebbero essere supportate da sistemi MAS che risolvono problemi in cui l'impiego del paradigma ad agenti dovrebbe risultare particolarmente utile. Per fare un esempio, una delle applicazioni considerate in questa classe è un sistema per la selezione dei programmi trasmessi da TV via cavo tenendo conto del profilo indicato da ogni utente.

Abstract architecture. Specifiche che definiscono in astratto le entità e i servizi necessari per fornire agli agenti un ambiente in cui possano comunicare ed interoperare tra di loro.

Agent Communication. Specifica dell'agent communication language FIPA ACL. All'interno di questa classe di specifiche ci sono ulteriori suddivisioni non visualizzate in figura:

- **Interaction Protocols.** Rappresentano schemi di comunicazione, veri e propri protocolli di comunicazione ai quali si possono riferire gli scambi di messaggi FIPA ACL.
- **Communication Acts.** Specifiche riguardanti le azioni comunicative che sono supportate da FIPA ACL.
- **Content Language.** Specifica di alcuni linguaggi che possono essere utilizzati come Content Language nei messaggi FIPA ACL. Con tali linguaggi si esprimono i corpi delle richieste, sottoscrizioni, affermazioni e proposte indicate dagli atti comunicativi.

Agent Management. Specifiche che descrivono la gestione ed il ciclo di vita degli agenti all'interno di una stessa *platform*[8] o fra platform differenti.

Agent Message Transport. Specifica di come i messaggi possono essere trasportati attraverso differenti protocolli di rete, anche questa classe di specifiche è suddivisa in tre sottoclassi:

- **ACL Representations.** Specifiche delle differenti rappresentazioni dei messaggi FIPA ACL. Sono previsti differenti formati, dalla rappresentazione a tuple alla rappresentazione XML.
- **Envelope Representation.** Specifiche delle differenti rappresentazioni delle "buste" in cui si possono incapsulare i messaggi FIPA ACL.
- **Transport Protocols.** Specifiche dei protocolli con cui possono essere scambiati i messaggi FIPA ACL sopra differenti trasporti di rete.

Nel seguito approfondiremo le classi di specifiche relative all'architettura astratta [27] e l'agent management [28]. Lasceremo fuori da questa presentazione, praticamente per intero, le parti relative all'Agent Communication ed Interaction Protocols che saranno trattate nel lavoro di tesi di Stefano Miglia e le specifiche appartenenti alla classe dell'Application.

3.3 Architettura astratta

L'architettura astratta è stata ottenuta basandosi sui principi di analisi e design propri della programmazione ad oggetti, usufruendo della progettazione a "pat-

[8] Il termine platform si riferisce all'implementazione delle componenti che forniscono le funzionalità previste nell'ambiente in cui gli agenti "vivono".

tern" ed utilizzando modellazione UML[9]. Il diagramma 3.6 illustra le relazioni tra le astrazioni definite nel documento e le loro istanziazioni concrete. Sempre nella stessa figura si possono osservare quattro livelli di astrazione, dall'alto verso il basso incontriamo prima il livello più astratto al quale troviamo le comunicazioni e la semantica delle comunicazioni, poi l'architettura astratta, le specifiche concrete, fino alla singola implementazione concreta dell'architettura che sta al livello di astrazione più basso.

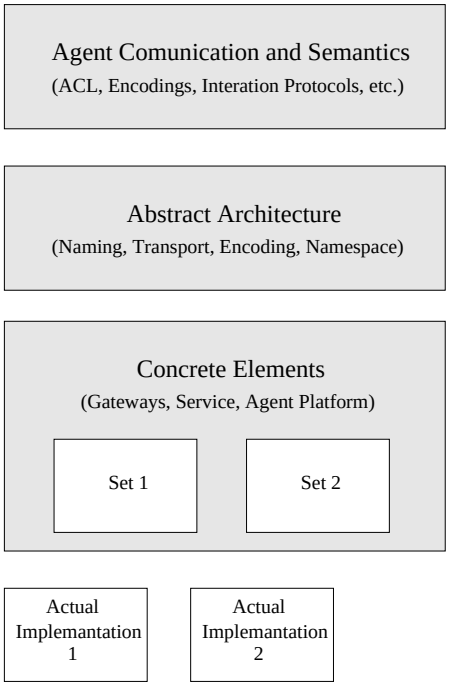

Figura 3.6: Relazione tra architettura astratta e concreta.

3.3.1 Tipo opaco ed iscrizione ai servizi

Vi sono due leitmotiv che permeano la progettazione l'architettura astratta: l'utilizzo del cosiddetto tipo opaco e il pattern "factory".

Cosa significa che alcuni degli elementi di cui l'architettura si compone siano di tipo opaco? Illustriamo il concetto con l'esempio del Message Transport Service (MTS). Come avremo modo di illustrare più dettagliatamente nel seguito, quando due agenti comunicano si vuole astrarre dai possibili protocolli di rete sui quali può essere trasmesso un messaggio. Per questo motivo la comunicazione è progettata secondo quello che è stato definito un approccio "agnostico"

[9]Nel documento di specifica dell'architettura astratta si trovano una serie di class diagram che descrivono le relazioni fra gli elementi che compongono l'architettura.

in cui l'implementazione del trasporto è opaca alle componenti che forniscono le funzionalità di comunicazione agli agenti. In questo modo si evitano assunzioni dipendenti dal particolare tipo di trasporto con cui di volta in volta avviene la comunicazione. Quando un messaggio viene spedito gli è associato un identificativo del tipo di trasporto che dovrà impiegare in funzione dei trasporti condivisi dal mittente e dal destinatario del messaggio. Sulla base di tale identificativo si selezionerà dinamicamente il tipo di trasporto adatto alla comunicazione.

Elementi di tipo opaco sono utilizzati sia nella codifica dei messaggi sia nei servizi di directory. Tale espediente offre una soluzione elegante che permette l'estensione dinamica delle funzionalità dell'ambiente in cui vivono gli agenti. Si possono infatti aggiungere dinamicamente nuovi tipi opachi contenenti nuove implementazioni di servizi. Si osservi che il meccanismo dei tipi opachi non è che un'istanza del concetto di polimorfismo.

Un secondo concetto ricorrente nella specifica riguarda "l'iscrizione a servizio" ovvero il modo in cui gli agenti accedono alle funzionalità dell'ambiente. L'applicazione del concetto d'iscrizione al servizio, prevede che un agente possa utilizzare solo i servizi dell'ambiente[10] ai quali si è iscritto. L'iscrizione restituisce all'agente un riferimento al servizio assicurando all'agente la possibilità di accedere allo stesso successivamente.
Questo modo di organizzare le cose corrisponde ad un "pattern" molto utilizzato denominato "factory".

3.3.2 Cosa specifica l'architettura astratta

La specifica dell'architettura astratta [27] offre una descrizione ad alto livello dei servizi e delle componenti che devono essere forniti dall'infrastruttura nella quale vivono gli agenti. L'obiettivo principale del documento è assicurare che le realizzazioni concrete derivate dall'architettura astratta siano in grado di interoperare tra di loro. L'interoperabilità delle differenti realizzazioni concrete è assicurata dal fatto stesso che siano istanza dell'architettura astratta, e condividano quindi un design appositamente pensato per interoperare con MAS che supportano una struttura simile.

Vista la centralità delle problematiche di comunicazione e interoperazione fra agenti, nel documento sono descritti i seguenti temi:

- l'interoperabilità col Message Transport;
- il supporto di varie forme delle rappresentazioni dell'Agent Communication Language;

[10] Nel senso di platform.

- il supporto a differenti forme di content language[11];
- il supporto alla rappresentazione di directory service multipli.

Nell'architettura si cerca di individuare le modalità con cui fornire servizi nei quali ACL, content language, MTS siano parametrici. Rimane fuori da tale documento:

- tutto ciò che non può essere descritto astrattamente (es. gestione del ciclo di vita degli agenti e sicurezza);
- le questioni sulle quali non c'è ancora un sufficiente accordo da prevedere una standardizzazione;
- le aree che sono sufficientemente specializzate da non necessitare di una standardizzazione;
- il ciclo di vita degli agenti e la loro gestione (trattato in [28]);
- la mobilità degli agenti;
- l'identità degli agenti;
- le politiche di conversazione degli agenti (es. interaction protocol).

È importante osservare che l'architettura astratta specifica gli elementi necessari ad ottenere la comunicazione e l'interoperazione fra sistemi multi-agente, e questo non significa che componenti o problematiche che non sono trattate dell'architettura astratta siano necessariamente proibite o ignorate dalla FIPA.

Non si è necessariamente costretti a utilizzate la specifica astratta per intero, si può anche fare riferimento ad essa per implementare un singolo elemento; ad esempio, si potrebbe creare una specifica concreta dell'elemento directory service che utilizza il Lightweight Directory Access Protocol (LDAP)[12]. Essendo quel particolare elemento un'istanza di una visione astratta potrebbe essere utilizzato in diversi sistemi ad agenti che si riferiscano alla stessa visione come componente di terza provenienza.

Una specifica di un'architettura concreta per essere FIPA compatibile deve includere il meccanismo per la registrazione e la scoperta degli agenti ed il trasferimento dei messaggi intra-agente.

Per spiegare quali elementi siano necessari e come interagiscono tra di loro, il documento di specifica si serve di un semplice esempio: due agenti che devono localizzarsi e comunicare l'uno con l'altro.

[11] linguaggi in cui si esprime la parte domain dependent in un messaggio FIPA ACL.

[12] È uno standard per i servizi su una rete Intranet o Internet. LDAP fornisce servizi di directory e viene spesso utilizzato per associare nomi a numeri telefonici e ad indirizzi e-mail.

Nel modello proposto dalla FIPA gli agenti comunicano scambiandosi messaggi codificati in un agent communication language, di cui daremo una brevissima descrizione più avanti.

La comunicazione avviene tramite due servizi di supporto agli agenti: *directory service* e *message-transport-service*[13].
I servizi in questione possono essere forniti come funzionalità offerte da agenti, ad esempio tramite un agente directory server e un agente message-transport; oppure in forma diversa fornendo le interfacce per le implementazioni Java, C++, IDL.
Nel caso in cui si scelga di fornire i servizi via agenti, avremo agenti con una limitata autonomia. Infatti, a differenza degli agenti autonomi che decidono volta per volta se fornire un servizio che gli è stato richiesto, gli agenti di cui parlavamo sopra sono vincolati a servire sempre le richieste. Nelle sottosezioni seguenti descriviamo più accuratamente i servizi di directory e transport message.

3.3.3 Directory Services

Il ruolo di un *directory-server* [14] è fornire una "locazione" dove gli agenti possono registrare una **directory entry**[15] . Nel directory server gli agenti possono trovare le informazioni necessarie per interagire gli uni con gli altri.

Una directory entry è in astratto una tupla di coppie valore-chiave consistente di almeno due elementi:

Agent-name: un nome unico per l'agente.

Locator: uno o più **Transport description** che descrivono il tipo di trasporto e l'indirizzo da utilizzare per comunicare con l'agente utilizzando un determinato trasporto. Notare che l'indirizzo è dipendente dal tipo di trasporto.

All'interno di una directory entry possono essere mantenuti altri attributi, ad esempio la descrizione, il costo dell'utilizzo, le restrizioni di impiego dell'agente. I directory service comprendono servizi per la registrazione, la cancellazione di una directory entry e l'interrogazione del directory server. Quest'ultimo servizio permette agli agenti di ottenere le informazioni relative altri agenti registrati.

Per dare un'idea di come si svolgono le cose, illustriamo cosa succede quando un agente $\mathcal{A}$ vuole publicizzare la sua esistenza. Innanzi tutto lega se stesso ad

[13]Nel seguito si è cercato di indicare in grassetto i termini che sono elementi specificati dalla FIPA. Continuiamo ad utilizzare l'italico per gli elementi che sono introdotti dalle specifiche ma che rappresentano insiemi di funzionalità o comunque non hanno un unico corrispondente nelle specifiche meno astratte.

[14]Con directory server intendiamo la componente o le componenti che offrono i directory services, non avendo un "preciso" corrispondente nella specifica astratta è in italico.

[15]Directory entry è in grassetto in quanto è un elemento ben precisato nella specifica astratta, e non individua un gruppo di elementi né un gruppo di funzionalità.

uno o più trasporti[16], dopo di che, costruisce la sua directory entry completa del suo nome e dei trasporti con i quali può essere contattato più attributi opzionali fra cui le ontologie che è in grado di comprendere. Infine registra la directory entry nel directory server.

Sempre a titolo di esempio supponiamo che l'agente $\mathcal{A}$ si registri al directory server con una directory entry che contenga le coppie chiave-valore illustrate in figura 3.7.

{agent-service,"com.dowjones.stockticker"}
{ontology, org.fipa.ontology.stockquote}

Figura 3.7: Coppie chiave-valore contenute nella directory entry dell'agente $\mathcal{A}$.

{ontology, {org.fipa.ontology.stockquote}}

Figura 3.8: Chiave-valore di ricerca.

A questo punto, supponiamo arrivi un agente $\mathcal{B}$, che effettui una ricerca per una directory entry che include le coppie chiave-valore in figura 3.8. Il risultato della ricerca che si vedrebbe restituire $\mathcal{B}$ è la directory entry dell'agente $\mathcal{A}$. Esaminando il contenuto del directory entry che gli è stata restituita, $\mathcal{B}$ è in grado di conoscere con quale trasporto ed a quale indirizzo può comunicare con l'agente $\mathcal{A}$.

3.3.4 Messaggi e Message Transport Services

Dopo aver dato un'idea di come un agente possa venire a conoscenza dell'esistenza degli altri agenti in un MAS che gode della proprietà di openness, presentiamo gli aspetti fondamentali della comunicazione fra agenti:

- la struttura dei messaggi che si scambiano;
- il trasporto dei messaggi.

Struttura dei messaggi e FIPA ACL.

Abbiamo già detto che gli agenti FIPA comunicano fra di loro scambiandosi messaggi espressi in un Agent Communication Language (ACL). La FIPA ha definito un suo agent communication language FIPA ACL, del quale diamo di seguito una brevissima descrizione. In astratto i messaggi sono strutturati come un tupla di coppie chiave-valore che contengono[17]:

[16] In alcune implementazioni questo compito è delegato al message-transport-service.
[17] La lista non comprende tutti gli elementi della struttura.

Sender. Il mittente del messaggio.

Receiver. Il destinatario del messaggio. Può essere indicato anche più di un destinatario per uno stesso messaggio.

Performatives. Identificano il tipo di atto comunicativo al quale si riferisce il messaggio. Ci dice se il messaggio è un'affermazione, una domanda, una risposta, un rifiuto o qualcosa d'altro.

Content language. Il linguaggio con cui è espresso il contenuto del messaggio. Esistono differenti linguaggi che possono essere impiegati come content language, fra i quali KIF o FIPA SL.

Content. Il contenuto vero e proprio del messaggio che è espresso in un content language. Ad esempio se il messaggio ha performative "request" la quale indica che il mittente richiede al destinatario di fare un'azione, il content conterrà la formulazione dell'azione richiesta.

Ontology. L'ontologia è il dominio al quale si riferiscono i termini del messaggio, ovvero elementi che appariranno all'interno del content ma non appartengono alla sintassi del content language. L'ontologia identifica l'accezione con cui sono utilizzati i termini ed il significato dei termini appartenti al dominio del discorso.

Nell'architettura astratta non si è obbligati ad adottare FIPA ACL come ACL: abbiamo dato una descrizione di FIPA ACL per aver presente quale può essere la struttura approssimativa di un messaggio scritto in un ACL. Per maggior dettaglio sulla struttura e gli atti comunicativi supportati da FIPA ACL si faccia riferimento a [24] e [30]. In figura 3.9 è fornita una schematizzazione riassuntiva della struttura di un messaggio espresso in un agent communication language. Si vede che il content language può essere riferito ad una ontologia [18] ed il messaggio contiene un mittente e dei destinatari espressi con il nome unico (**agent-name**) indipendente dal tipo di trasporto con il quale gli agenti sono raggiungibili.

Trasporto per i messaggi.

Vediamo cosa accade quando un messaggio viene spedito attraverso un particolare trasporto. Come illustrato in figura 3.10, un messaggio per essere spedito viene codificato in un **Payload** ed incluso in un **Transport Message**.
Il payload non è che una codifica del messaggio originale espressa in una forma

[18] Ricordiamo che l'ontologia definisce il significato dei termini nei contesti che possono essere discussi.

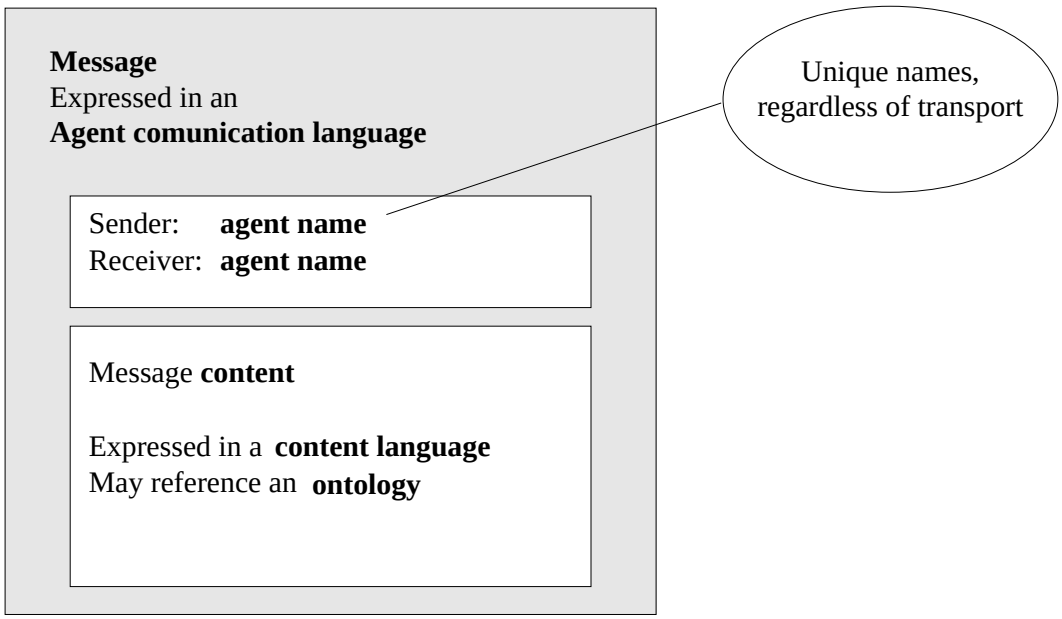

Figura 3.9: La struttura per un messaggio FIPA.

adatta ad essere trasmessa sopra un determitato trasporto. Supponiamo ad esempio che un messaggio debba essere trasmesso su una connessione *wireless*[19], al fine di ottimizzare la trasmissione, e quindi il numero di informazione che possono essere trasmesse in uno stesso intervallo di tempo, è conveniente impiegare una codifica del messaggio diversa dalla semplice codifica a stringa (es. FIPA Bit-Efficient Encoding).

Un transport message è formato da un envelope (la busta) ed il payload del messaggio che deve essere trasmesso. L'envelope contiene i **transport description** dell'agente mittente e del destinatario. Un transport description contiene le informazioni che permettono di spedire il messaggio sul trasporto: le caratteristiche del trasporto, l'indirizzo del destinatario, alcuni dettagli su come è realizzato il trasporto; oltre a informazioni aggiuntive come l'*encoding-rappresentation*, dati relativi alla sicurezza, altri dati che devono essere visibili al trasporto.

In figura 3.10 si può osservare una schematizzazione del processo che trasforma un messaggio in un transport message: a partire dal messaggio viene creato un payload in funzione del trasporto selezionato; dopo di che si crea un envelope con le informazioni utili al trasporto. Se necessario possono essere aggiunti ulteriori envelope, il tutto compone ciò che abbiamo definito come message transport.

[19] La caratteristica delle reti wireless (letteralmente senza fili) è offrire una banda di trasmissione piuttosto limitata.

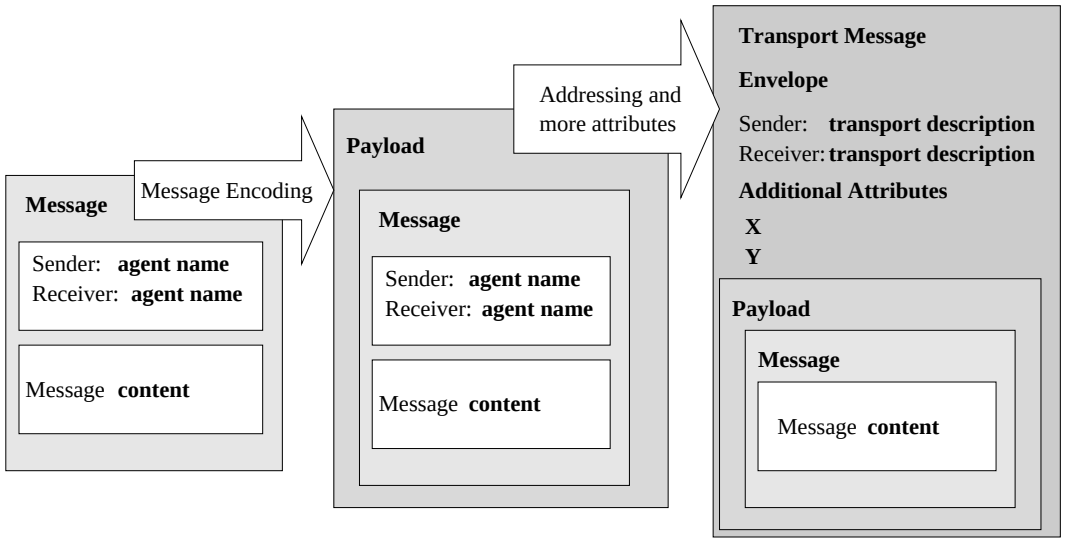

Figura 3.10: Trasformazione di un messaggio in un Transport-Message.

3.3.5 Esempio di comunicazione

Analogamente a quanto fatto per i servizi di directory service riportiamo da [27] un esempio che illustri cosa accade quando si spedisce un messaggio. Innanzi tutto chiariamo che ogni agente ha un **agent-name** unico ed immutabile. Un agente può avere uno o più **transport-description** che utilizza per spedire ad altri agenti i propri transport-message. Ogni transport-description è correlato ad un particolare **transport** (trasporto, es. IIOP, HTTP). Un **transport** è dunque un meccanismo per trasferire messaggi. Un transport message è un messaggio che viene spedito da un agente ad un altro in un formato che è appropriato per il transport usato. Un insieme di tranport-description può essere mantenuto in un **locator**. Ricordiamo che il locator è parte della directory entry di un agente.

Consideriamo un agente con agent-name "ABC" con una directory entry come quella schematizzata in tabella 3.1. Si osservi che tale agente può essere raggiunto attraverso due differenti trasporti, HTTP e SMTP. Supponiamo ora vi sia un agente

Agent-name: ABC		
Locator:		
Transport-type	**Transport-specific-address**	**Transport-specific-property**
HTTP	http://www.whitz.net/abc	(none)
SMTP	Abc@lowcal.whiz.net	(none)

Tabella 3.1: Directory entry per ABC

con agent-name ”123” che voglia comunicare con l’agente ”ABC” e supporti il transport HTTP. In figura 3.11(a) è illustrato il contenuto del **transport message** nel caso in cui la comunicazione fra i due avvenga tramite trasporto HTTP. Sempre

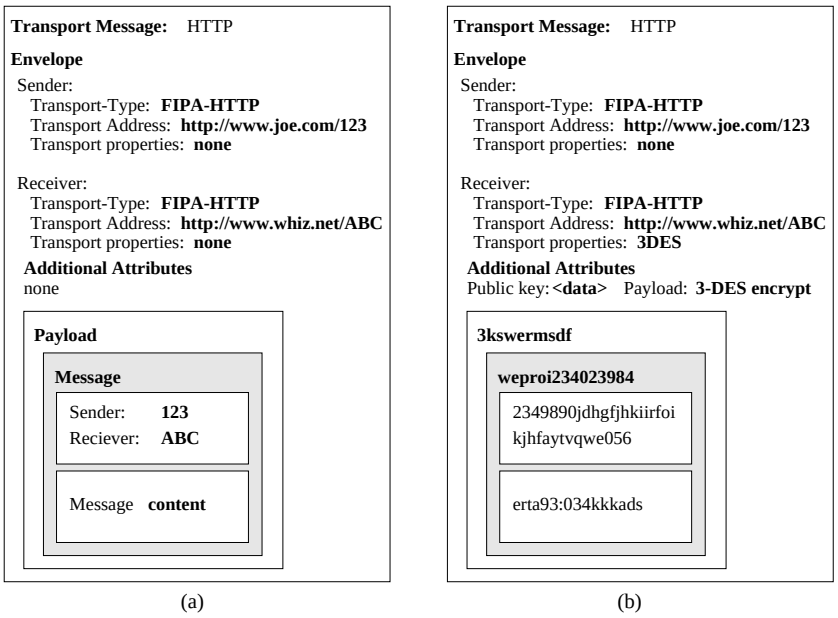

Figura 3.11: (a) Transport-Message per la comunicazione tra ”123” e ”ABC”. (b) Lo stesso Transport-Message criptato.

in figura 3.11(b) viene illustrato un esempio di utilizzo degli attributi addizionali, in questo caso per spedire messaggi criptati.

3.4 Specifica della gestione degli agenti FIPA

Nella sezione precedente abbiamo dato un’idea delle componenti astratte che permettono agli agenti d’interoperare. In questa sezione scendiamo di un livello di astrazione e presentiamo la specifica [28] che descrive con maggior dettaglio com’è fornita l’interoperabilità degli agenti illustrando inoltre come vengono gestiti gli agenti.

3.4.1 Un modello di riferimento

Prima di tutto andiamo a descrivere un modello di riferimento per la gestione degli agenti, sul quale illustreremo successivamente le operazioni di creazione, registrazione, locazione, comunicazione, migrazione e ritiro di agenti.

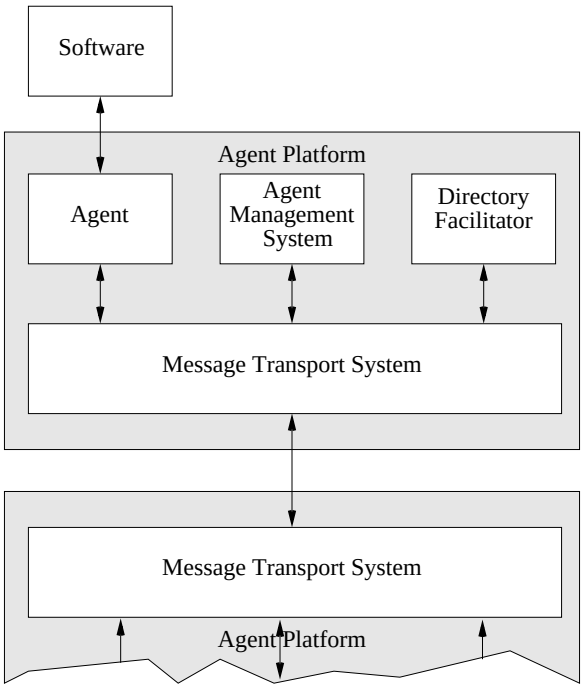

Figura 3.12: Il modello di riferimento FIPA per la gestione degli agenti.

Le entità contenute nel modello di riferimento illustrato in figura 3.12 sono insiemi logici di servizi, non è detto che vi sia corrispondenza di un elemento ad un elemento con le componenti fisiche di implementazione. Inoltre, nella specifica viene lasciata una certa libertà per i dettagli più implementativi delle platform. Descriviamo brevemente le singole entità.

Agente: è l'attore fondamentale nella Agent Platform (AP), può fare da intermediario ad un utente umano o ad un software preesistente. Deve avere un identificatore **Agent Identifier**[20] (AID) universalmente unico. Può essere registrato ad un certo numero di indirizzi di trasporto attraverso i quali può essere contattato.

Directory Facilitator (DF): è un componente obbligatorio della platform. Il DF fornisce servizi di pagine gialle agli altri agenti. Gli agenti possono attraverso esso registrare i loro servizi o chiedere quali servizi siano offerti dagli altri.

[20] Si faccia attenzione in quanto le specifiche FIPA si riferiscono all'AID intendendo spesso oltre all'identificatore unico e immutabile dell'agente anche un insieme di "parametri" dell'identificatore ad esempio i transport address supportati, questo doppio utilizzo del termine può essere motivo di confusione.

Agent Management System (AMS): è una componente obbligatoria della agent platform, che esercita il controllo e la supervisione degli accessi e l'uso della platform. Ce n'é uno per ogni Agent platform. Questa componente colleziona gli Agent Identifier (AID) per gli agenti registrati nella platform[21].

Message Transport Service (MTS): sono i servizi usuali di comunicazione per agenti locati sulla stessa o su differenti platform. La comunicazione interplatform viene specificata in un documento a parte [29].

Agent Platform (AP): fornisce l'infrastruttura fisica nella quale gli agenti possono essere impiegati. La platform può consistere in una macchina, un sistema operativo, un software per il supporto di agenti, un insieme di componenti misto agli agenti. La progettazione interna della platform non è oggetto di standardizzazione. Non è detto che vi sia una platform per ogni macchina, è possibile implementare delle platform distribuite. La FIPA prevede una varietà di differenti AP che vanno da un singolo processo che contiene agenti sotto forma di thread ad una AP totalmente distribuita costruita sopra middleware standard o proprietari. FIPA specifica solo con cosa e come la comunicazione avviene fra agenti. Gli agenti possono essere nativi della platform, agenti esterni alla platform, o agenti che dinamicamente si sono registrati ad essa.

Software: questa componente descrive tutto ciò che non è un agente ma deve essere accessibile dall'agente. L'agente può accedere a software per aggiungere nuovi servizi, protocolli di interazione, protocolli di negoziazione, algoritmi, protocolli per la sicurezza, eccetera[22].

3.4.2 Il nome degli agenti

Il modello di riferimento FIPA identifica un agente con una collezione di coppie chiave valore, chiamate Agent Identifier (AID). Queste collezioni di coppie possono contenere un nome ed altri parametri come gli indirizzi di trasporto, l'indirizzo di un servizio di risoluzione di nomi ed altro. La natura estensibile dell' AID dovrebbe permettere di arrichirlo a proprio piacimento con informazioni quali il nome sociale, un soprannome, i ruoli che può giocare e quant'altro si voglia. Gli

[21] Considerando che gli AID possono avere alcuni parametri tra cui i transport address supportati, l'AMS implementa il servizio di pagine bianche. Si osservi che a questo livello di dettaglio vengono distinti i due servizi di directory e quindi il servizio di pagine bianche viene offerto da una componente (AMS) differente da quella che offre il servizio di pagine gialle (DF).

[22] La FIPA ha prodotto delle specifiche su come importare funzionalità da software esterno. Si veda [25].

AID sono pensati per comparire dentro agli envelope dei messaggi nei campi :to e :from. I valori dei parametri di un AID possono essere editati e modificati da un agente, è ad esempio possibile che un agente voglia aggiornare la sequenza dei propri indirizzi di trasporto; tuttavia, i parametri obbligatori possono essere cambiati esclusivamente dall'agente a cui l'AID si riferisce. Il parametro :name di un AID è un identificatore globalmente unico. Un modo semplice per ottenere ciò è costruirlo a partire dal nome dell'agente e l'indirizzo della platform su cui risiede (home agent platform, abbreviato con HAP), separando i due con un carattere '@'. Supponiamo ad esempio di avere un agente di nome "ABC" che risiede sulla platform con HAP "miaplatform.disi.unige.it", l'AID risultante sarà "ABC@miaplatform.disi.unige.it"

3.4.3 Il Directory Facilitator

Il directory facilitator è un componente obbligatorio in ogni agent platform, il cui compito è fornire un servizio di pagine gialle agli agenti. Viene inteso come un custode benigno, nel senso che deve essere implementato in modo che gli sia impossibile ingannare gli agenti coscientemente fornendo informazioni maliziose o parzialmente errate. In una AP vi possono essere più DF che lavorano in collaborazione con il DF principale formando una "federazione".

Vediamo come interagiscono gli agenti con il directory facilitator. Se un agente vuole pubblicizzare i suoi servizi agli altri agenti, deve contattare il DF e richiedere la registrazione della descrizione dei servizi. L'atto della registrazione non implica nessun obbligo da parte dell'agente che si è registrato, al quale è permesso rifiutare una richiesta di un servizio da lui precedentemente pubblicizzato. Il DF da parte sua non assicura la validità e l'accuratezza delle informazioni che sono state registrate, né opera alcun controllo sul ciclo di vita di ogni singolo agente. La descrizione dei servizi con la quale l'agente pubblicizza le sue capacità ha una serie di parametri obbligatori, ed è comunque possibile aggiungere anche informazioni non previste dallo standard FIPA. Illustriamo le funzioni che devono essere fornite dal DF, esse corrispondono ad una semantica ben definita che non illustreremo in quanto piuttosto intuitiva e troppo di dettaglio per la nostra presentazione. I servizi comprendono:

- la registrazione dei servizi nel DF;
- l'eliminazione di una registrazione;
- la modifica di una registrazione;
- la ricerca dell'agente o degli agenti che offrono servizi con determinate caratteristiche;

Il directory facilitator che esiste in ogni AP ha un AID riservato simile a quello che segue:

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

Figura 3.13: AID tipo del Directory Facilitator.

Questa convenzione permette agli agenti che non conoscono l'indirizzo del facilitatore di registrare i propri servizi in qualsiasi platform FIPA alla quale abbiano accesso.

3.4.4 Agent Management System

La componente AMS è obbligatoria ed unica in ogni AP. Tale componente è responsabile della gestione degli agenti (es. creazione, cancellazione degli agenti) e di tutte le operazioni sulla platform comprese l'accettazione delle registrazioni di agenti preesistenti e la migrazione degli agenti da una AP ad un'altra[23]. Visto che le agent platform possono avere caratteristiche piuttosto eterogenee, l'AMS se interrogata in proposito offre una descrizione della relativa AP. Ad ogni agente è associato una ciclo di vita, questa associazione viene mantenuta dall'AMS. Nel caso in cui la platform sia distribuita su più macchine l'AMS rappresenta l'autorità in tutte le macchine. L'AMS può richiedere agli agenti di eseguire alcune funzioni di gestione come la terminazione e ha l'autorità di costringere l'esecuzione della richiesta se questa venisse ignorata. L'AMS mantiene un indice di tutti gli agenti che in un certo istante risiedono nella platform, mantenendo per ognuno il relativo AID. Un agente è residente in una platform se si è registrato con l'AMS. Per essere conforme al modello di riferimento della FIPA ogni agente deve registrarsi all'AMS. Con la registrazione l'agente ottiene l'autorizzazione ad accedere al MTS ed è quindi in grado di riceve e spedire messaggi. L'AMS ha il compito di controllare la validità di tutte le descrizioni di agenti che gli pervengono, in particolare controlla l'unicità del nome dell'agente nel AID. Le descrizioni degli agenti possono essere modificate per qualsiasi ragione ma le modifiche sono soggette a politiche di autorizzazione. Dopo la cancellazione di un agente, il relativo AID viene rimosso dall'AMS ed è quindi disponibile nel caso che qualche altro agente lo volesse utilizzare. Ogni AMS nel propi AP ha un AID riservato della seguente forma:

[23] Anche la mobilità degli agenti è gestita dall'AMS.

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

Figura 3.14: AID tipo del Agent Management System.

Analogamente a ciò che avveniva con il directory facilitator, anche qui, la convenzione sull'indirizzo dell'AMS permette agli agenti di registrarsi alla platform conoscendo soltanto l'indirizzo HAP.

Riepilogando l'AMS deve essere in grado di eseguire i seguenti servizi per gli agenti:

- registrazione;
- cancellazione;
- modifica;
- ricerca;
- restituzione delle descrizioni della platform.

Oltre alle funzionalità di gestione degli agenti:

- sospendere un agente;
- terminare un agente;
- creare un agente;
- recuperare l'esecuzione di un agente;
- invocare un agente;
- eseguire un agente;
- gestire le risorse

Le funzionalità precedenti sono fornite dall'AMS, vi è però una funzione che deve essere obbligatoriamente fornita dagli agenti: la funzione "quit"[24].

[24] Il fatto che questa funzionalità sia offerta dagli agenti è dovuta al fatto che tale funzionalità permette agli agenti di terminare in maniera controllata.

3.4.5 Ciclo di vita degli agenti

A questo punto dovrebbe essere chiaro che gli agenti FIPA esistono fisicamente in una platform[25] il cui compito è offrire le funzionalità e facilitazioni precedentemente descritte a beneficio degli agenti stessi. Nel contesto della platform un agente è un processo che deve essere gestito secondo un ciclo di vita che modella l'esistenza di ogni istanza di agente indipendente dall'applicazione che si sta sviluppando. In un dato istante ogni agente può trovarsi in uno solo degli stati contenuti del ciclo di vita descritto in figura 3.15.

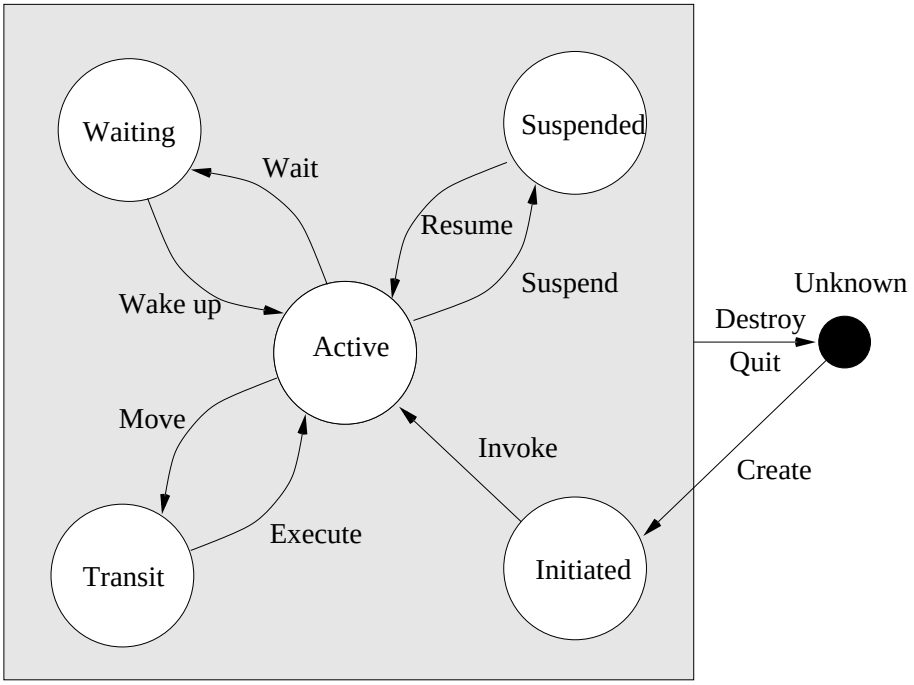

Figura 3.15: Il ciclo di vita degli agenti FIPA.

Vediamo ora in ogni stato del life-cycle degli agenti come viene gestita la comunicazione.

Active: Il MTS consegna i messaggi all'agente normalmente.

Initiated/Waiting/Suspended: in questi tre stati il MTS colleziona i messaggi fino a quando l'agente non ritorna attivo. Nel caso sia impostato un servizio di inoltro ad un'altra locazione per un agente che sia in uno di questi stati effettua l'inoltro.

[25] Ciò che ad un altro livello di astrazione abbiamo genericamente definito come ambiente in cui gli agenti vivono.

Transit: il MTS colleziona i messaggi fino a che l'agente non ritorna attivo. Un agente può ritornare attivo sia perché il tentativo di muoversi in una nuova destinazione è fallito, sia perché l'agente ha iniziato la sua esecuzione in un nuova AP. Solo gli agenti mobili possono entrare nello stato Transit.

Unknown: il MTS può sia collezionare i messaggi sia rifiutarli, dipende dalla politica del MTS e dalle richieste specificate nel trasporto dei messaggi.

Diamo ora una breve descrizione delle transizioni che permettano di passare da uno stato all'altro.

Create: la creazione o l'installazione di un nuovo agente.

Invoke: l'invocazione di un nuovo agente.

Destroy: la terminazione forzata di un agente, questa operazione dovrebbe essere iniziata solo dal AMS e non può essere ignorata dall'agente.

Quit: Una terminazione più gentile che può anche essere ignorata dall'agente.

Resume: mette l'agente sospeso nello stato active. Questa transazione può essere iniziata solo dall'AMS.

Suspend: mette l'agente nello stato suspended. Questa transizione può essere iniziata sia dall'agente che dalla platform.

Wait: transizione che mette l'agente nello stato di Waiting, ciò può avvenire solo per iniziativa dell'AMS.

Wake up: toglie l'agente dallo stato di waiting, questa transizione può essere fatta solo dall'AMS.

Vi sono poi due transizioni che possono essere usate solo da agenti mobili:

Move: mette l'agente nello stato di Transit, questa transizione può essere intrapresa solo su iniziativa dell'agente stesso.

Execute: toglie l'agente dallo stato Transit, questa transizione può essere intrapresa solo dall'AMS.

3.5 Le ontologie

Fino ad ora abbiamo parlato dei servizi offerti da una Agent Platform non occupandoci di come gli agenti possano accedere a questi servizi. Parlando dell'architettura astratta avevamo accennato al fatto che le componenti possono essere implementate sotto forma di agenti, oggetti, moduli. Nel caso in cui la componente sia implementata come un oggetto o un insieme di oggetti è possibile accedere ai suoi servizi tramite chiamate a metodo o attraverso API (Application Program Interface). Diversamente se i servizi sono offerti da agenti è possibile accedervi utilizzando delle comunicazioni di richiesta. Ricordiamo che che la comunicazione tra agenti avviene su tre livelli di astrazione distinti, che coinvolgono:

- ontologie;
- linguaggio di comunicazione per agenti;
- protocollo di trasporto.

Per i livelli di astrazione più bassi corrispondenti al linguaggio di comunicazione e protocolli di trasporto abbiamo già detto cosa prevedono le specifiche FIPA fornendo un descrizione per FIPA ACL e MTS. Descriviamo cosa prevedono le specifiche FIPA per le ontologie.

La FIPA assume che se la comunicazione coinvolge due agenti questi condividano almeno una ontologia in comune. Se fosse altrimenti i due non riuscirebbero a dare lo stesso significato ai termini dei messaggi che si scambiano.

Per un determinato domino applicativo si può pensare di utilizzare le ontologie in maniera esplicita o implicita. Utilizzare le ontologie in maniera implicita significa codificare le ontologie nel codice degli agenti. In questo modo non è possibile accedere all'ontologia se essa non è stata codificata durante lo sviluppo dell'agente. In alternativa a questo tipo di approccio si possono trattare le ontologie in modo esplicito.

Le specifiche FIPA permettono agli agenti di utilizzare ontologie esplicite attraverso quelli che sono gli ontology service. Gli ontology service sono forniti da un agente dedicato denominato Ontology Agent (OA) e comprendono:

- la scoperta di nuove ontologie pubbliche;
- la conservazione delle ontologie accessibili;
- la traduzione di espressioni tra due ontologie differenti[26];

[26]Questa funzionalità non deve essere ottenuta necessariamente in maniera automatica, ed è comunque spesso di difficile realizzazione. Per questo le specifiche FIPA prevedono che ad una richiesta di traduzione il OA possa rispondere che non è in grado di fornirla.

- rispondere a domande sulle relazioni che intercorrono tra due termini o due ontologie;
- facilitare l'individuazione di una ontologia comune tra due agenti;

Non è interesse di questa tesi entrare troppo nel dettaglio della questione, in generale le ontologie sono domain dependent, ed in [26] vengono illustrati dettagliatamente gli ontology services che potrebbero essere supportati dall'ambiente in cui gli agenti vivono. L'esistenza di un Ontology Agent in una platform non è obbligatorio nelle specifiche FIPA. Le specifiche FIPA oltre a prevedere gli strumenti che facilitino l'utilizzo delle ontologie nei contesti dipendenti dal dominio, definisce delle sue ontologie standard per permettere agli agenti di accedere alle funzionalità delle componenti descritte nelle specifiche.

La FIPA definendo le ontologie dei servizi offre gli strumenti per permettere all'agente di fare proprie richieste ai componenti dell'architettura. Infatti quando un agente vuole richiedere un servizio ad una componente implementata in forma di agente non può servirsi di una chiamata a metodo o una chiamata a procedura, ma necessita di spedire una messaggio in FIPA ACL. Per esprimere una richiesta di registrazione ad un DF implementato come agente, avrà bisogno di un'ontologia con cui esprimere i concetti relativi alla descrizione dei servizi che vuole registrare.

Per portare un esempio, viene proposta un'ontologia *FIPA-Agent-Management* nella quale sono definiti i termini del linguaggio con cui parlare del domino della gestione degli agenti FIPA. Standardizzare le ontologie con cui parlare dei servizi offerti dalle componenti FIPA è equivalente a standardizzare le "procedure" con cui si richiedono i servizi, si noti che ciò è simile a quello che abitualmente si ottiene nei contesti di programmazione orientata agli oggetti quando si definisce l'interfaccia con la quale accedere ai servizi degli oggetti, solo che nel caso degli agenti attraverso le ontologie lavoriamo ad livello di astrazione più elevato. Concludiamo osservando che nella specifiche FIPA le ontologie hanno una duplice utilità:

- descrivere gli elementi della specifica e la relativa semantica;
- fornire un strumento per richiedere i servizi alle componenti realizzate come agenti.

3.6 Progetti FIPA

Vi sono alcuni progetti a partecipazione internazionale, che hanno adottato le specifiche FIPA in modo da verificare l'applicabilità delle delle stesse. Ne descriviamo brevemente alcuni riportando ciò che è illustrato in [52].

FACTS: un progetto per validare il lavoro della FIPA e specifiche standard provenienti da altri gruppi attraverso la costruzione di sistemi basati sulle specifiche FIPA. L'obiettivo del progetto è controllare le effettive potenzialità dell'interoperabilità nei termini specificati dalla FIPA. Il progetto è suddiviso in due fasi. Nella prima fase vengono fatte interoperare platform di differente provenienza su ognuno dei tre campi applicativi specificati (audio-video broadcasting an entertainment, service reservation, elettronic commerce). Nella fase successiva si fanno interagire sistemi appartenenti a diversi campi applicativi.

LEAP: è il precursore della seconda generazione di platform per agenti FIPA compatibili. LEAP (Lightweight Extensible Agent Platform) individua le necessità relative ad infrastrutture aperte con servizi che supportano distribuzione su reti wireless, per questo particolarmete dinamiche e mobili. Le platform di questo tipo rappresentano una sfida dal punto di vista tecnico, in quanto necessitano di generare ed eseguire agenti in una grande varietà di ambienti run-time in modo da "girare" su differenti hardware (cellulari, palmari, PC) e comunicare con una varietà di meccanismi (TCP/IP, WAP, ecc.).

Agentcities: ha lo scopo di realizzare applicazioni di ricerca e commerciali basati su agenti che accelerino lo sviluppo di una successiva generazione di servizi Internet. Tale obiettivo viene perseguito in tre modi:

1. sviluppando una infrastruttura che sia pubblicamente e continuamente accessibile e possa ospitare i servizi per gli agenti;
2. sviluppando i servizi in modo da popolare la rete di una vasta offerta di servizi gratuiti e commerciali sul quale costruire altre applicazioni;
3. incoraggiare la collaborazione nella ricerca: promuovendo una rete che individui gli obiettivi di ricerca e sviluppo su grande scala quindi abiliti lo scambio di informazioni tra progetti esistenti e futuri.

Ognuno di questi progetti ha messo in luce nuove richieste rispetto alle esigenze d'interoperabilità FIPA scavando ulteriormente negli aspetti della semantica della interoperabilità. Progetti come Agentcities portano a lavorare ulteriormente sulle ontologie, le interazioni di servizi, le responsabilità e fiducia negli agenti.

3.7 Conclusioni

In questo capitolo abbiamo brevemente presentato parte del lavoro della FIPA, concentrandoci sugli aspetti più vicini alle problematiche di comunicazione e le

funzionalità che dovrebbero essere offerte dalle platform per agenti FIPA compatibili.

Abbiamo incominciato ad interessarci al fenomeno FIPA quando ci siamo imbattuti per la prima volta nelle specifiche dell'architettura astratta. Rendendoci conto che ciò che era specificato nel documento rappresentava una sorta di meta-MAS dal quale l'utente poteva istanziare il suo sistema ad agenti pensammo che tale framework concettuale ci potesse aiutare a focalizzare ulteriormente le esigenze e le problematiche che poteva incontrare l'utente di D-CaseLP.

Spesso abbiamo avuto a che fare con specifiche sviluppate da working group e technical committee differenti. Ciò unito al fatto che la specifica astratta è appositamente definita come una specifica *loose*[27] e che in una prima fase non disponevamo di altro materiale al di fuori delle specifiche FIPA per raccapezzarci fra gli obiettivi dell'intento di standardizzazione, non ha reso semplice il compito di esplorare le specifiche e metterle in relazione tra di loro. Quella che all'inizio doveva essere una veloce occhiata al lavoro altrui ha cambiato le prospettive in cui ci muovevamo svolgendo un ruolo fondamentale in questo lavoro di tesi. Man mano che prendevamo confidenza con le specifiche ci rendevamo conto che le ipotesi da noi operate trovavano conferma ed in certi casi una forma più definita nel lavoro effettuato dalla FIPA[28].

Il tentativo di standardizzazione operato dalla FIPA ha il merito di essere un passo verso una visione più concreta della tecnologia ad agenti. Siamo consapevoli che la standardizzazione non è la panacea a tutti i "mali" ma a nostro avviso può avere importanti benefici per ciò che riguarda l'affermarsi della tecnologia basata sul paradigma ad agenti.

Per ottenere una piena legittimazione da parte degli ambienti di produzione è necessario sviluppare un certo numero di progetti che individuino e mostrino contributo delle tecniche ad agenti nella risoluzione problemi concreti e l'effettivo vantaggio del loro impiego rispetto alle tecnologie più consolidate.

La ricerca di una legittimazione non va considerata come il mero tentativo di trovare un'ulteriore gratificazione economica al lavoro di ricerca, ma piuttosto come un passo fondamentale per lo sviluppo del paradigma stesso. Come osservato da Nwana e Ndumu in [47] al fine di avere un'effettivo riscontro di quelle che sono le potenzialità del paradigma ad agenti è di primaria importanza che la tecnologia sia ampiamente impiegata. Questo implica la necessità di risolvere problemi concreti ed accollarsi costi notevoli, due condizioni di cui probabilmente solo le

[27] Con il termine specifica loose intendiamo una specifica da cui è possibile ottenere implementazioni piuttosto diverse tra loro.

[28] Si pensi ad esempio all'esigenza di avere un'interfaccia verso la comunicazione che permettesse il supporto di differenti implementazioni della comunicazione o le posizioni sulla minimalità che portano entrambi a non specificare dettagli inutili del micro-livello degli agenti.

industrie si possono fare carico.

Sempre Nwana e Ndumu in [47] analizzano lo scenario che si prospetta nella produzione scientifica sull'argomento negli anni successivi al 1995 fino al 1999 e osservano alcuni fenomeni a loro dire abbastanza preoccupanti:

- alcuni dei lavori pubblicati presentavano ancora confusione tra Object Orientation e concetti relativi agli agenti;
- molte proposte erano caratterizzate da una prematura matematizzazione. La ricerca di basi matematiche aveva l'effetto di spostare prematuramente parte degli sforzi che potevano essere impiegati nell'individuazione dei vantaggi offerti dal paradigma ad agenti in una ricerca di fondamenti per il paradigma;
- in alcuni lavori si poteva osservare una rietichettatura[29] di concetti da tempo noti. Questo fenomeno non sempre di natura "dolosa" portava a lavori il cui contributo era di limitata consistenza.

A nostro parere la standardizzazione in corso ad opera della FIPA, per le modalità ed i principi che abbiamo parzialmente descritto, pur lasciando una sufficiente libertà di ricerca offre un framework concettuale che può fungere da fondamento per l'applicazione concreta del paradigma ad agenti. Questo può essere utile a focalizzare gli obiettivi della ricerca futura ed in parte eliminare i problemi evidenziati da Nwana e Ndumu.

Il fatto che la FIPA nasca come forum che vede la presenza di importanti aziende e gruppi di ricerca (anche di origine accademica) costituisce un elemento positivo che fa ben sperare in questo senso. Ciò che ci auguriamo è che tale iniziativa possa giocare un ruolo simile a quello che l'OMG ha giocato nell'ambito delle tecnologie ad oggetti. Durante i mesi in cui si è lavorato a questo lavoro di tesi, abbiamo avuto l'impressione che l'interesse rispetto all'iniziativa sia in continuo aumento, vi sono un certo numero di platform che si sono adeguate o hanno scelto di aderire alle specifiche FIPA. Nei prossimi capitoli illustreremo l'impatto che le specifiche FIPA hanno su D-CaseLP.

[29] Processo secondo il quale si danno nomi nuovi a concetti vecchi.

Capitolo 4

Inclusione del framework FIPA in D-CaseLP

In questo capitolo illustriamo quale impatto ha avuto l'introduzione delle specifiche FIPA sulla progettazione di D-CaseLP. Daremo una descrizione delle caratteristiche offerte dal tool JADE per lo sviluppo di agenti FIPA compatibili e l'apporto che questo fornisce nella realizzazione di D-CaseLP. Concluderemo con un confronto tra JADE e CORBA, il quale insieme alle motivazioni di progetto[1] ci fornirà gli elementi che giustificano la scelta di abbandonare l'impiego di CORBA e la decisione di progettare un D-CaseLP basato su JADE.

4.1 Perché includere le specifiche FIPA

Il modello concettuale proposto dalla FIPA ha cambiato le prospettive di progetto di D-CaseLP. L'impatto di questa variazione sul nostro lavoro è stata abbastanza consistente, nonostante ciò vi sono valide motivazione alla base di questo aggiustamento di rotta. A prescindere dal fatto che la realtà FIPA si sta evolvendo rapidamente e diventa sempre meno sensato ignorare l'operato di tale organizzazione, vale la pena illustrare quali fattori ci hanno spinto a mettere D-CaseLP in relazione con il lavoro FIPA.

Architetture per agente. La FIPA non standardizza l'intelligenza interna dei MAS, le specifiche si concentrano sugli aspetti relativi l'intelligenza esterna. Questo lascia ampia libertà rispetto alla scelta dell'architettura per agente che si deve adottare nella risoluzione dei propri problemi. Permettere al

[1] Le motivazioni di progetto sono tutte le considerazioni relative al contesto progettuale in cui stiamo lavorando.

progettista di scegliere quale architettura sia più adeguata invece che costringerlo ad adattare le sue soluzioni su un'architettura scelta a priori è uno dei punti cruciali di D-CaseLP e prima ancora CaseLP, il fatto che la FIPA faccia ipotesi analoghe alle nostre è incoraggiante.

Una visione concreta. La FIPA offre delle specifiche che si propongono di risolvere problemi da una prospettiva concreta. Il fatto che vi siano industrie coinvolte nell'iniziativa focalizza il lavoro verso obiettivi pratici, il cui fine ultimo è soddisfare le aspettative di applicazione degli agenti. Di questi aspetti il progetto D-CaseLP non può che far tesoro. Il gruppo di programmazione logica dell'università di Genova ha avuto esperienze significative di collaborazioni con l'industria per la risoluzione di problemi concreti si vedano gli articoli [16, 17, 3]. Queste esperienze però, sono limitate alla progettazione di prototipi di MAS ed all'individuazione di una soluzione logica per i problemi. Per tali progetti, infatti, il gruppo non ha curato la parte di realizzazione successiva alla definizione del prototipo. L'adozione delle specifiche FIPA ci permette di indirizzare alcune delle problematiche che si incontrano non appena si esce dalla visione astratta dei problemi e ci permette di recuperare da terzi parte dell'esperienza di cui deficitiamo.

Nuove direzioni di ricerca. Le specifiche FIPA pur avendo come scopo principale rendere fruibile la tecnologia ad agenti in contesti applicativi, a volte mettono in evidenza le problematiche su cui è utile focalizzare la ricerca. Fra le esigenze teoriche che si stanno evidenziato vi è un'evoluzione della comunicazione basata su Agent Communication Language verso il supporto di ruoli e interaction protocol. La FIPA non propone una metodologia di sviluppo propria, anche se in un certo senso il framework concettuale che emerge dalle specifiche ha anche una valenza metodologica: si pensi all'architettura astratta e l'utilizzo che se ne può fare. Il progettista di MAS può far uso dell'architettura astratta per individuare alcuni dei problemi da affrontare durante un design, e sfruttare tale specifica come una specie di "formina" con la quale progettare sistemi MAS capaci di interoperare con altri. L'architettura astratta non è ovviamente una metodologia vera e propria corrisponde al massimo ad un insieme di linee guida per il design di MAS, c'è quindi ampio spazio per proporre nuove metodologie per la realizzazione di MAS aderenti alla FIPA.

Le specifiche della FIPA coinvolgendo diversi gruppi di ricerca offrono una visione unitaria suddivisa in più livelli di astrazione di quelle che sono le problematiche ancora aperte, offrendo spesso anche un'indicazione indiretta di quali sono le più urgenti, se si vuole che il paradigma trovi un suo spazio nel mondo delle applicazioni.

Sistemi aperti. Quando il progetto D-CaseLP è partito ci si limitava a considerare ambienti chiusi, nei quali non si prevedeva che i MAS dovessero interagire con altri per risolvere i propri problemi. Questa ipotesi può risultare abbastanza limitativa. Nonostante sia di difficile realizzazione, e che la specifica FIPA indirizzi solo alcuni dei problemi che è necessario risolvere per ottenere praticamente sistemi aperti, è futuribile che tale caratteristica giocherà un ruolo importante nello sviluppo del paradigma ad agenti. Negli ultimi tempi sono apparsi anche articoli che iniziano a considerare l'impatto che potrebbe avere la commercializzazione di sistemi aperti nel contesto sociale [13, 51]. La rivoluzione di cui siamo stati testimoni negli ultimi anni con l'avvento di Internet, ha avuto un impatto sulla società che è sotto gli occhi di tutti[2], la realizzazione di comunità aperte di agenti potrebbe portare al completamento di questa rivoluzione. In una prospettiva a lungo termine ci sembra che la capacità d'interagire con altri MAS in maniera del tutto dinamica, ovvero senza che i sistemi coinvolti siano stati pensati per interagire uno con l'altro, per quanto difficile da ottenere sia una direzione che va almeno tenuta presente. Gli sforzi impiegati verso tale direzione se non porteranno al raggiungimento completo degli obiettivi possono comunque fornire interessanti spunti per la risoluzione dei problemi legati all'integrazione di software legacy.

4.1.1 Aggiornamento del FRT di D-CaseLP

La progettazione del tool e la valutazione delle specifiche FIPA per un certo periodo sono state due attività che abbiamo portato avanti contemporaneamente. Questo modo di procedere e la consapevolezza che il progetto D-CaseLP disponeva di una limitata forza lavoro ha fatto si che l'inserimento delle componenti prescritte dalla FIPA fosse graduale. Per quanto riguarda le caratteristiche FIPA da includere nel tool, l'idea era di accontentarsi di un tool parzialmente aderente alle specifiche FIPA, che contenesse almeno le caratteristiche minime relative a servizi di Directory Service e supporto a trasporti multipli (MTS).

In figura 4.1 illustriamo il FRT aggiornato con gli elementi della specifica FIPA di cui vogliamo tener conto già dai primi prototipi di D-CaseLP. Per quanto riguarda i servizi di directory si pensava di limitarci a fornire servizi equivalenti a quelli di pagine bianche, lasciando per eventuali lavori di estensione i servizi di pagine gialle. Invece, per i servizi di trasporto ci eravamo proposti di fornire

[2]Si pensi ad esempio all'uso della posta elettronica che in ambito lavorativo ha reso praticamente obsoleto l'utilizzo del FAX o al boom del Word Wide Web con servizi come l'Internet Banking, il Trading On Line, ma anche agli entusiasmi che hanno generato espressioni come "new economy".

1. Modellazione e creazione di codice per il prototipo del MAS.
   ... ...
2. Realizzazione e validazione di un prototipo della applicazione sotto sviluppo.
   (a) **Installazione degli strumenti offerti da D-CaseLP.**
   (b) **Implementazione prototipo del MAS.**
      ... ...
      i. **Supporto alla comunicazione.**
         - **Directory Name Service (FIPA)**
         - **Comunicazione via MTS (FIPA)**
3. Implementazione della applicazione finita. ... ...
4. Utilizzo della applicazione finita. ... ...

Figura 4.1: Aggiornamento dell'albero delle funzionalità D-CaseLP.

un unico trasporto implementato attraverso CORBA ma nascondere l'implementazione attraverso il meccanismo di degli MTS, implementando un trasporto che facesse uso degli envelope e dei message transport come prescritto dalla FIPA (si veda sez. 3.3.4). Così previa la realizzazione di altri trasporti avremmo ottenuto in modo elegante il supporto a diverse implementazioni delle primitive di comunicazione.

4.1.2 Le platform aderenti alle specifiche FIPA

L'esistenza di alcuni tool per la realizzazione di sistemi multi-agente che fanno uso del framework concettuale della FIPA, ci ha fatto sorgere il dubbio che ci fosse qualcosa di già fatto da poter riutilizzare ed eventualmente estendere per fornire gli strumenti previsti in D-CaseLP. Vi sono tool molto differenti fra di loro che offrono un supporto verso la realizzazione di MAS FIPA compatibili[3] Come evidenziato nella tabella 4.2, fra i tool aderenti alla FIPA ve ne è uno che ha alcune caratteristiche interessanti per i nostri scopi: JADE. Ciò che ci ha convinto ad analizzare questo tool è l'esplicito riferimento da parte dei progettisti [8] alla possibilità di implementare differenti architetture per agenti a partire dagli strumenti in esso forniti.

[3] La lista completa ed aggiornata dei tool è disponibile all'indirizzo http://www.fipa.org/resources/livesystems.html.

Nome tool	Linguaggio d'implementazione per tool	Linguaggio d'implementazione per (program) agenti	Architettura per agenti supportata
Zeus	Java	desire, belief, intention	BDI modificata
JADE	Java	Java + API	Qualsiasi purché esprimibile in Behaviour
April	C	April	Qualsiasi esprimibile in April
Fipa-OS	Java	Java	Architettura strutturata in stack di compiti, conversazioni e comunicazioni su differenti livelli

Figura 4.2: Progetti Open Source di tool per la realizzazione di MAS aderenti alle specifiche d'interoperabilità FIPA.

4.2 JADE

JADE (Java Agent Development Enviroment) è un progetto open source che offre gli strumenti software che permettono di sviluppare applicazioni basate ad agenti compatibili con le specifiche di interoperabilità FIPA. Vediamo alcune delle principali caratteristiche offerte da JADE:

- una agent platform distribuita;
- una API Java per spedire e ricevere messaggi FIPA ACL;
- una libreria di interaction protocol pronti per essere utilizzati;
- il supporto alla mobilità degli agenti;
- librerie per gestire ontologie definite dagli utenti;
- un'implementazione aderente alla specifiche FIPA del protocollo IIOP per connettersi ad altre platform;
- un'iterfaccia grafica per monitorare e controllare l'esecuzione distribuita degli agenti;

Nelle sottosezioni seguenti presenteremo il framework ricalcando la suddivisone utilizzata in [4], dove si presenta il tool prima da un punto di vista descrittivo

illustrando le varie componenti che permettono l'esecuzione degli agenti e poi dal punto di vista dell'utente che lo deve utilizzare, descrivendo quali sono le conoscenze e le classi Java che permettono di scrivere gli agenti.

4.2.1 JADE Run-Time

Agent platform.

La platform di JADE, può essere considerata la componente principale del Run-Time, include un Agent Management System (AMS) ed un Directory Facilitator (DF) realizzati sotto forma di agenti attivati automaticamente all'inizializzazione della platform.

La platform JADE può essere distribuita su differenti hosts attraverso quelli che vengono chiamati *Agent Container*. Ogni agent-container vive su una Java Virtual Machine distinta, è un Remote Method Invocation (RMI) server ed è responsabile per gli agenti in esso residenti. Il controllo del ciclo di vita degli agenti viene gestito in locale dal singolo agent container che ha la possibilità creare, sospendere, distruggere gli agenti in maniera indipendente dagli altri.

La suddivisione in container e di conseguenza la distribuzione della platform JADE è totalmente trasparente agli agenti esterni alla platform. Esiste, infatti, un agent container principale denominato *Main-Container* o *front-end* che incarna le funzionalità di una server IIOP[4], attraverso il quale gli agenti residenti su altre platform, non necessariamente platform JADE, possono interagire con gli agenti JADE. Il main-container sulla base delle informazioni contenute nell'AMS si occupa di ridistribuire le comunicazioni ai container sui quali risiedono gli agenti coinvolti nella comunicazione.

La comunicazione tra agenti in JADE.

Come illustrato in [7] la comunicazione in JADE è di tipo peer-to-peer[5] ed avviene tramite scambio di messaggi. È prevista una diversa modalità di consegna dei messaggi FIPA ACL a seconda di dove gli agenti sono collocati.

Comunicazione Intra-Container: avviene fra agenti residenti in uno stesso *Agent-container*. Per motivi di efficienza la comunicazione avviene tramite

[4]Un server che è in grado di accettare richieste con l'Internet Inter-Orb Protocol (IIOP). Tale protocollo permette di inoltrare richieste ad oggetti distribuiti attraverso il trasporto TCP/IP.

[5]Ricordiamo che questo tipo di comunicazione al contrario della comunicazione client-server, prevede che le due entità coinvolte nella comunicazione abbiano uguali "doveri" e "diritti".

un oggetto evento Java contenente il messaggio senza che siano effettuate le operazioni *marshalling* ed *unmarshalling* del messaggio[6].

Comunicazione Inter-Container: avviene fra agenti residenti su diversi *Agent-container* ma appartenenti alla stessa JADE platform. La comunicazione in questo caso è effettuata tramite Remote Method Invocation (RMI).

Comunicazione Inter-platform Avviene fra agenti residenti su platform distinte, non necessariamente *JADE Agent Platform*, in conformità con quanto specificato dalla FIPA.

La scelta del tipo di comunicazione tra due agenti viene effettuata a seconda di dove sono collocati i due in modo del tutto trasparente agli agenti ed al programmatore, in piena coerenza con la filosofia FIPA. Al fine di selezionare il meccanismo di trasporto più efficiente ogni agent-container mantiene una tabella delle locazioni degli agenti chiamata *Local-Agent Description Table* (LDAT). Il main-container oltre alla LDAT mantiene anche una tabella *Global-Agent Description Table* (GADT), che mappa ogni agente nel riferimento RMI del suo container. Per evitare che il main-container sia continuamente interpellato sono utilizzate tecniche di "caching" dei riferimenti per le comunicazioni inter-container. Le tecniche di caching permettono di mantenere localmente ai vari container le informazioni relative ai riferimenti dei container che ospitano agenti con cui è necessario comunicare.

È nella comunicazione Inter-Platform che entrano in gioco la possibilità di utilizzare differenti implementazioni dei trasporti. Come si accennava prima il trasporto utilizzato è in genere IIOP ma è possibile aggiungere o sovrascrivere altre implementazioni di MTS con le quali effettuare comunicazioni inter-platform. Gli autori dichiarano che la progettazione è stata fatta in modo da poter aggiungere altre tipologie di trasporto (WAP o quant'altro) senza grossi sforzi. Insieme al run-time di JADE è possibile ottenere alcune implementazioni di MTS, tra le quali il trasporto http ed il trasporto Orbacus IIOP.

Un agente software in aderenza alle specifiche FIPA ha un identificare globalmente unico (globally-unique identifier: GUID)[7] che può essere utilizzato da ogni agente per indicare negli appositi campi dei messaggi FIPA ACL i destinatari ed il mittente. Come avevamo visto nelle specifiche FIPA gli indirizzi degli agenti sono della forma <nome dell'agente>@<indirizzo della platform>.

[6]L'operazione di marshalling permette di trasformare i messaggi in un formato adeguato al trasporto su una certa rete, mentre l'unmarshalling è l'operazione inversa. In questo caso, trattandosi di una comunicazione tra due agenti che risiedono sulla stessa macchina la conversione del messaggio in un formato adeguato al trasporto sarebbe del tutto inutile.

[7]Nel capitolo relativo alla FIPA ci eravamo riferiti all'identificatore globalmente unico come Agent-Identifier AID.

Ontologie e content language definiti dall'utente.

Nel capitolo dedicato alla FIPA abbiamo già descritto il ruolo che il content language e le ontologie giocano nella comunicazione tra agenti, abbiamo anche precisato che la FIPA definendo un insieme di ontologie supportate dei servizi da lei specificati standardizza l'interfaccia con la quale si accede agli stessi. Al di là dell'uso delle ontologie indipendenti dal dominio applicativo della FIPA, l'utente per permettere di fornire i servizi che progetta agli agenti avrà bisogno di definire delle ontologie dipendenti dal dominio. Nelle versioni più recenti di JADE, sono forniti ai programmatori gli strumenti per dare la possibilità di crearsi le loro ontologie ed adottare i loro content language. Ogni agente JADE tiene una tabella dove sono elencati i content language e le ontologie conosciute.

Strumenti per la gestione ed il monitoring dell'esecuzione.

Oltre che un'implementazione della platform in cui possono essere eseguiti gli agenti, vengono forniti gli strumenti per il monitoring e la gestione dell'esecuzione distribuita. Tutti gli strumenti che andremo a descrivere nel seguito sono forniti sotto forma di agenti nella platform JADE sulla quale operano.

Remote Monitoring Agent (RMA). Il framework JADE offre i consueti strumenti grafici per effettuare il monitoraggio ed il controllo (start, stop, ecc) remoto degli agenti eseguiti. L'aspetto più interessante è che esiste una proposta presso la FIPA per standardizzare l'ontologia di monitoring utilizzata da JADE, e quindi l'iterfaccia con cui si accede ai servizi di controllo e monitoring degli agenti. Se tale proposta venisse accettata ed aggiunta all'ontologia *Fipa-agent-management*, RMA potrebbe potenzialmente essere utilizzato anche per il controllo e la gestione di agenti che non sono eseguiti dalla *JADE Agent Platform*. In figura 4.3 si può osservare come si presenta l'interfaccia grafica del RMA.

Dummy Agent. È un agente fantoccio che permette di spedire messaggi FIPA ACL agli altri agenti da un agente fittizio. In questo modo si possono stimolare gli agenti con richieste costruite artificialmente volte a testarne il corretto funzionamento. La preparazione dei messaggi FIPA ACL è facilitata da un'interfaccia che permette di editare tutti campi di cui i messaggi si compongono e gli envelope in cui racchiuderli.

Sniffer Agent. Questo agente permette di visualizzare i messaggi che si scambiano gli agenti residenti in una platform JADE. È possibile selezionare quali agenti spiare fra tutti quelli in esecuzione nella platform. Oltre a vedere chi comunica con chi tramite una specie di *sequence diagram* dei messaggi (figura 4.4), è possibile editare il contenuto delle comunicazioni.

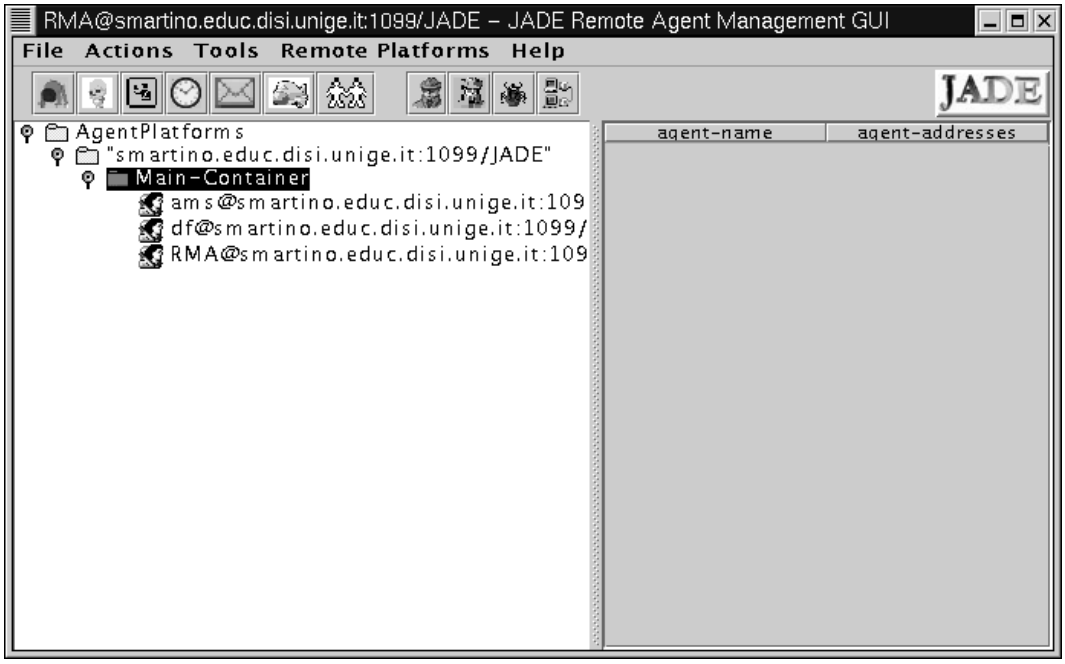

Figura 4.3: L'interfaccia grafica con cui si accedono le funzionalità del RMA.

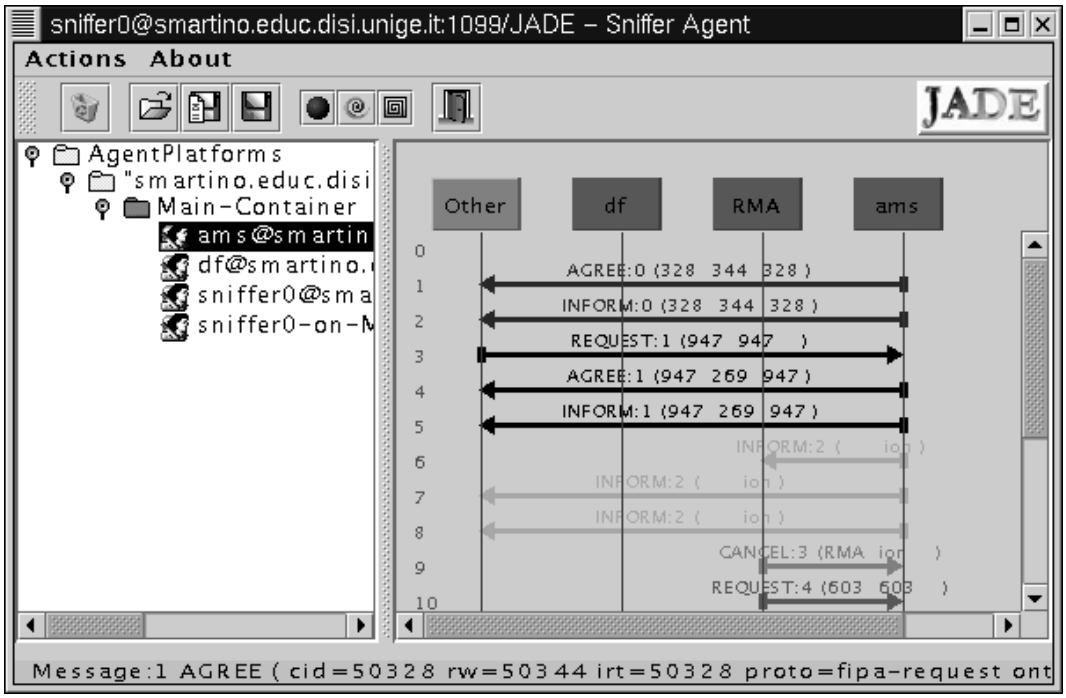

Figura 4.4: L'interfaccia grafica dello tool Sniffer.

Introspector Agent. Questo tool è stato aggiunto nelle ultime versioni di JADE, permette di visualizzare lo stato interno di un agente attraverso la visualizzazione della coda dei *Behaviour*[8] e la visualizzazione delle code dei messaggi in uscita ed in entrata all'agente. Vengono visualizzate anche le code dei messaggi pendenti che stanno aspettando di essere spediti o ricevuti (figura 4.5).

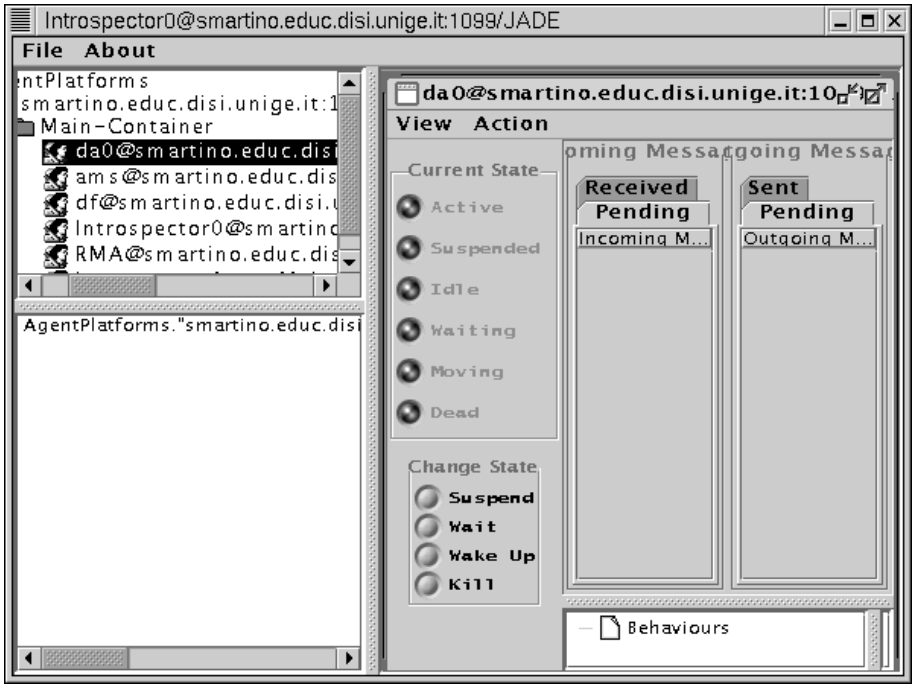

Figura 4.5: L'interfaccia grafica del tool Introspector.

4.2.2 Il modello di sviluppo degli agenti JADE

In questa sottosezione illustriamo il modello attraverso il quale si possono scrivere agenti JADE. In questo modello, sono offerte un insieme di funzionalità volte ad aiutare lo sviluppo degli agenti e la loro esecuzione sul run-time JADE. Ogni agente JADE è un thread eseguito da uno dei container della JADE platform. In concreto gli agenti JADE non sono altro che classi scritte in Java utilizzando le funzionalità offerte in una API che permette di accedere al run-time e di sfruttare una serie di classi predefinite per comporre e codificare messaggi FIPA ACL,

[8]Per il momento si considerino i Behaviour come sono insiemi di istruzioni che definiscono i compiti dell'agente che si sta monitorando, se ne darà una dettagliata descrizione nelle sezioni successive.

gestire ontologie, partecipare ad interaction protocol, eccetera. Nel seguito facendo riferimento a [6] illustriamo alcune delle classi che appartengono alle API JADE in modo da dare un'idea, seppur minima, degli strumenti offerti per la programmazione di agenti.

La classe Agent.

Ogni agente estende la classe Agent definita nelle librerie JADE. In tale classe è fornita l'implementazione dei metodi che permetteranno alla platform di controllare l'agente secondo il ciclo di vita specificato dalla FIPA. Sempre nella classe Agent sotto forma di metodi sono fornite le primitive che permettono all'agente di accedere ai servizi di comunicazione offerti dalla platform. Ogni agente ha una mailbox privata in cui la platform consegna i messaggi ad esso destinati. L'agente può accedere in diversi modi al contenuto della sua mailbox attraverso le primitive di ricezione. Estendendo la classe Agent tutte queste caratteristiche vengono ereditare dagli agenti definiti dal programmatore. Le primitive di comunicazione supportate sono:

blockingReceive(): questo metodo offre la possibilità di accedere alla mailbox in maniera bloccante, quindi, se al momento in cui è invocato il metodo non vi sono messaggi nella mailbox con le caratteristiche desiderate l'agente rimane in attesa. Il metodo in questione è *overloaded*, questo implica che esistono diverse segnature che permettono di specificare un numero variabile di argomenti nella chiamata del metodo. I parametri che è possibile specificare sono:

- un oggetto di classe *MessageTemplate*, che specifica le caratteristiche del messaggio FIPA ACL del quale si è in attesa. Applicando il pattern matching[9] sui messaggi contenuti nella mailbox è possibile selezionarne uno aderente allo schema indicato. Se questo parametro non è specificato significa che si è implicitamente in attesa di un messaggio qualunque;
- un timeout, il quale esprime quanti millisecondi si è disposti ad attendere prima che un messaggio con le caratteristiche desiderate arrivi. Se il messaggio arriva entro il tempo indicato, viene restituito nella forma di un oggetto, se il tempo scade prima viene restituito il riferimento all'oggetto nullo "null". Quando questo parametro non è specificato l'attesa è realizzata mettendo l'agente nello stato *Blocked*. L'assenza del timeout significa infatti che si è disposti ad attendere un tempo an-

[9] Abbiamo illustrato la tecnica del pattern matching nella sottosezione 1.3.5.

che illimitato l'arrivo di un nuovo messaggio, mettendosi nello stato blocked l'agente verrà risvegliato solo se arriva un nuovo messaggio.

receive(): è la primitiva che permette di effettuare una ricezione non bloccante dei messaggi. Anche per questa si ha la possibilità di indicare uno schema (pattern) di messaggio. Nel caso in cui non vi siano nella mailbox messaggi con le caratteristiche desiderate la primitiva ritorna immediatamente "null", altrimenti restituisce un oggetto con il messaggio selezionato;

send(): è la primitiva che permette di spedire in modalità asincrona un messaggio espresso in FIPA ACL indicato come argomento.

Fra le cose che l'agente che stiamo scrivendo eredita vi è anche uno scheduler per i *Behaviour*. I Behaviour sono le componenti che definiscono il comportamento degli agenti. Per scrivere un agente JADE è necessario estendere la classe Agent, ed implementare i compiti di un agente scrivendo uno o più sottoclassi di Behaviour. Andiamo ad approfondire alcune aspetti legati ai Behaviour, ponendo particolare attenzione al modello di concorrenza che è sottinteso nel loro utilizzo.

4.2.2.1 Il modello di concorrenza di JADE

In questo paragrafo discutiamo le soluzioni adottate da JADE nel fornire la concorrenza agli agenti. La comprensione del modello di concorrenza sottostante JADE non è forse di immediata assimilazione ma è indispensabile se si vogliono utilizzare correttamente gli strumenti forniti dal tool. Abbiamo cercato di fornire anche alcune informazioni su come viene gestito il multithreading in Java in modo che sia possibile rendersi conto dei problemi e i vincoli che bisogna risolvere nel fornire la concorrenza in un tool sviluppato in Java.

Gli agenti per le caratteristiche di "sociality" e "autonomy" necessitano di due livelli distinti concorrenza:

- concorrenza tra l'esecuzione degli agenti;
- concorrenza tra compiti indipendenti che un agente deve intraprendere.

Visto che non si può pensare che ogni agente o peggio ogni sua attività abbiano un processore a disposizione, bisogna simulare la concorrenza con il multitasking ed il multithreading. Riportiamo per maggior chiarezza le definizioni per multitasking, multithreading e thread illustrate in [46].

Definizione 4.1 *Per* **multitasking** *si intende la caratteristica di un sistema operativo di permettere l'esecuzione contemporanea (o pseudocontemporanea*[10] *[...]) di diversi processi.*

[10]Con pseudocontemporanea si ci riferisce al fatto che se una macchina possiede un solo processore l'esecuzione contemporanea non può che essere simulata.

Definizione 4.2 *Un* **thread** *è un flusso di esecuzione del processore corrispondente ad una sequenza di istruzioni all'interno di un processo. Analogamente ai processi, bisogna distinguere il concetto di esecuzione di una sequenza di istruzioni da quello di thread, poiché ci possono essere diverse esecuzioni parallele di uno stesso codice, che danno origine a thread diversi.*

Definizione 4.3 *Per* **multithreading** *si intende l'esecuzione contemporanea (o pseudocontemporanea, [...]) di diversi thread nell'ambito dello stesso processo. La gestione del multithreading può essere a carico del sistema operativo, se questo supporta i thread, ma può anche essere assunta dal processo stesso.*

La differenza fra processi e thread sta nel fatto che i thread, fatta eccezione per lo stack e i dati temporanei usati dalle funzioni, condividono lo stesso spazio di memoria. Questa condivisione rende più semplice la comunicazione tra i thread e meno dispendioso il cambio di contesto quando si passa dall'esecuzione di un thread ad un altro.

In generale vi sono due tipi di multitasking: il *preemptive* o *cooperative*. Il primo tipo di multitasking ha come caratteristica principale che ogni attività una volta in possesso della risorsa processore, può essere interrotta ed essere costretta a lasciare la risorsa a qualcun altro. In particolare nei sistemi detti a *time-sharing* l'interruzione avviene sulla base del tempo d'utilizzo di un processo, si può definire un time-slice oltre il quale se un processo non ha finito di utilizzare la risorsa viene interrotto.
Il secondo tipo di multitasking, invece, non ha la caratteristica di essere interrompibile e nessuno può appropiarsi della risorsa fino a quando non viene volontariamente rilasciata dal processo che la detiene.

Le tipologie di multitasking sui processi appena illustrate si adattano anche al multithreading. JADE è completamente implementato in Java che è un linguaggio semi-interpretato, il cui codice oggetto (bytecode) è eseguibile su qualsiasi piattaforma per cui sia disponibile un'implementazione della Java Virtual Machine (JVM). Essendo progettato per essere così portabile, il bytecode non può fare riferimento a funzionalità che stanno ad un livello troppo basso. Quando si tratta di offrire le funzionalità di multithreading deve basarsi su un'astrazione dei livelli sottostanti. Questa astrazione è definita dalle specifiche ufficiali della JVM.

Le specifiche della JVM assumono che i thread siano gestiti con uno scheduling di tipo preemptive chiamato *fixed-priority scheduling*. In questo schema si prevede che i thread siano schedulati in base ad una priorità, ed a parte situazioni particolari, i thread con priorità più alta prevalgano sugli altri nella conquista della risorsa. Anche se in molti sistemi operativi è fornito un multithreading basato su

time-slicing[11], le specifiche della JVM non lo prevedono. È possibile tuttavia, in alcune implementazioni della JVM forzare l'utilizzo dei thread di sistema e accedere a funzionalità non previste dalla specifica della JVM. Se si vuole mantenere la completa portabilità del codice che si produce si deve assumere di lavorare sui *green thread* , i thread che sono gestiti dai processi ed ignorati dal sistema operativo. IL fatto che le specifiche della JVM non prevedano il time-slicing nella schedulazione dei thread implica che affinché le cose funzionino correttamente i thread Java devono rilasciare spontaneamente le risorse.

Ritornando a JADE, abbiamo detto che la platform è distribuita in agent-container ognuno corrispondente ad un'attivazione differente della Java Virtual Machine. Ogni istanza della JVM è un processo mentre ogni agente è un thread residente in uno degli agent-container della platform. Quindi in JADE la concorrenza tra agenti viene ottenuta

- grazie alla distribuzione su più agent-container fra agenti residenti su differenti container;
- grazie all'esecuzione concorrente dei thread per agenti residenti sullo stesso container attraverso un preemptive multithreading senza time-slice.

Ci rimane da illustrare come JADE fornisca la concorrenza tra le attività degli agenti. Si potrebbe pensare di fornire la concorrenza tra i compiti degli agenti associando ad ogni compito un thread distinto. Così facendo aumenterebbero cosistevolmente il numero di thread necessari per eseguire un MAS e va considerato il fatto che ci possono essere dei limiti al numero di thread che si possono attivare. Per questo motivo gli sviluppatori hanno preferito utilizzare dei thread logici: i Behaviour. I Behaviour racchiudono singoli compiti dell'agente e si basano su un cooperative multithreading gestito dagli agenti. In un certo senso hanno il compito di racchiudere istruzioni non interrompibili (se Behaviour semplici) e definire dove un agente può lasciare spazio ad un altro agente. Nella libreria JADE sono forniti diversi tipi di Behaviour, di volta in volta a seconda delle esigenze si può scegliere quale tipo di Behaviour fra quelli presenti nella gerarchia di classi illustrata in figura 4.6 sia il più adatto ad esprimere un certo compito.

Dettagliamo le caratteristiche delle tipologie di Behaviour contenuti nella gerarchia.

[11] Il time-slicing permette di riapplicare l'algoritmo di schedulazione indipendentemente dal fatto che il thread abbia finito di utilizzare il processore.

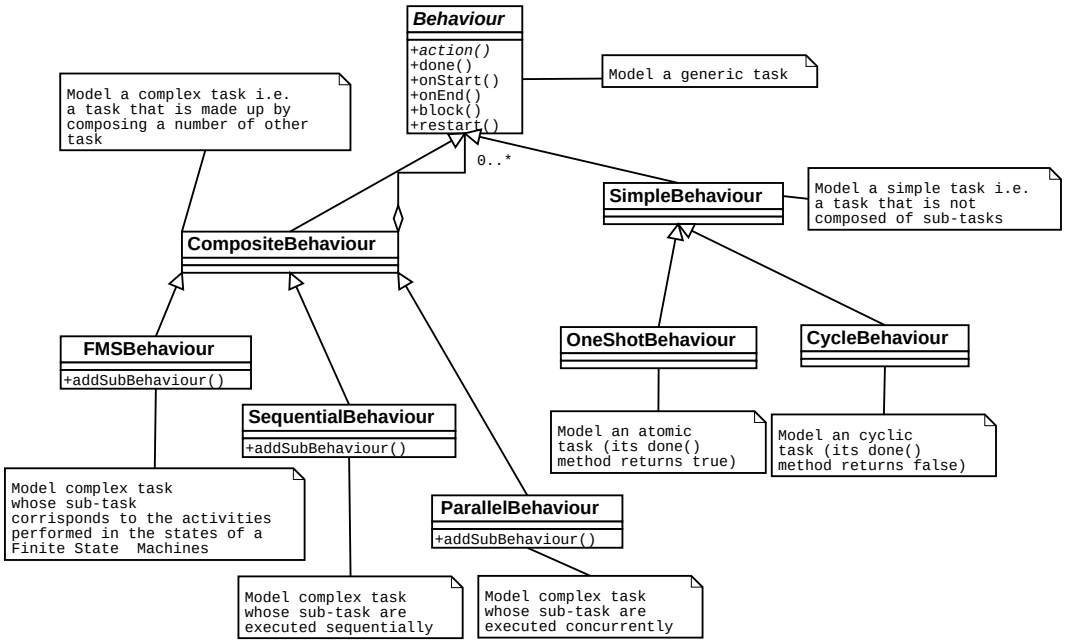

Figura 4.6: La gerarchia dei Behaviour in JADE.

La classe Behaviour.

La classe Behaviour fornisce una base astratta sulla quale costruire tutte le tipologie dei Behaviour. Per alcuni dei metodi definiti in questa classe è già data un'implementazione, per altri solo la segnatura.

action(): è un metodo astratto che restituisce il programmatore riempirà con il codice che esprime il compito rappresentato dal Behaviour.

done(): è anch'esso un metodo astratto che ritorna true quando il Behaviour ha terminato il suo compito.

onStart(): è un metodo che può essere ridefinito dall'utente per indicare quali istruzioni si debbano effettuare prima che il Behaviour venga messo nell'insieme dei Behaviour in esecuzione (attivi).

onEnd(): è un metodo che può essere ridefinito dall'utente per indicare quali istruzioni si debbano effettuare alla fine di un Behaviour. Questo metodo viene eseguito dopo che il Behaviour ha terminato la sua esecuzione ed è stato tolto dall'insieme dei Behaviour in esecuzione.

restart(): è il metodo che fa ripartire il Behaviour.

block(): è il metodo che permette all'oggetto Behaviour di interrompersi fino all'arrivo di un evento (es. un messaggio). La chiamata di questo metodo non impedisce agli altri Behaviour dell'agente di essere schedulati. Tutti i Behaviour bloccati sono rischedulati quando un messaggio arriva. È possibile che un oggetto Behaviour blocchi se stesso per un tempo limitato espresso in millisecondi. Un Behaviour bloccato può riprendere la sua esecuzione in uno di questi tre casi:

1. un messaggio è ricevuto dall'agente a cui il Behaviour appartiene;
2. il timeout specificato al momento in cui si è effettuato il blocco è scaduto;
3. il metodo *restart()* viene esplicitamente richiamato sul Behaviour bloccato.

Si osservi come i Behaviour permettano un controllo dettagliato del multitasking dell'agente.

Descriviamo le implementazioni della classe astratta Behaviour incluse nella libreria di JADE.

SimpleBehaviour. Questa classe astratta modella i Behaviour semplici ed atomici, ovvero tutti quei Behaviour che sono costituiti da istruzioni inscindibili. In questa classe il metodo reset[12] non fa nulla, ma può essere ridefinito se necessario dalle classi figlie.

OneShotBehaviour. Questa classe modella i comportamenti che devono essere eseguiti solo una volta, questo tipo di Behaviour non può essere bloccato. Il metodo done() restituisce sempre *true*[13]

CyclicBehaviour. È la classe astratta che definisce un comportamento atomico che deve essere rieseguito in continuazione.

CompositeBehaviour. Questa classe astratta modella i Behaviour che sono composti da un certo numero di Behaviour figli. Le operazioni che deve eseguire un Behaviour di questo tipo non sono scritte all'interno del Behaviour principale ma dentro ai Behaviour figli. La classe definisce un'iterfaccia per gestire i sotto-comportamenti (SubBehaviour) ma non implementa nessuna

[12] Nella documentazione che abbiamo consultato non è spiegato precisamente quale sia il compito del metodo reset, supponiamo che sia il metodo che rinizializza lo stato dei Behaviour.

[13] Da ciò che si è potuto intuire, il metodo done() restituisce sempre true perché non essendo questo tipo di Behaviour bloccabile quando viene attivato la prima operazione eseguita è l'action e il metodo done può essere eseguito solo dopo che l'action ha terminato la sua esecuzione. In altre parole questo Behaviour non necessita mai di essere rischedulato.

politica di schedulazione per essi. Tale compito è lasciato alle classi che estendono la CompositeBehaviour.

SequentialBehaviour. È una delle classi predefinite che fornisce una politica di schedulazione per i subBehaviour di tipo sequenziale. Termina quando tutti i sotto Behaviour sono terminati. Questa classe può essere utilizzata quando si hanno diversi insiemi di passi atomici che devono essere eseguiti in sequenza.

ParallelBehaviour. Questa classe esegue i subBehaviour concorrentemente e termina quando una particolare condizione nei suoi sottocomportamenti è raggiunta. Si possono definire comportamenti di questo tipo che terminano quando tutti i sottocomportamenti, oppure un certo numero di sottocomportamenti definito dall'utente, sono terminati. Questa classe può essere utilizzata nel caso in cui un compito possa essere formulato in un certo numero di sottocompiti alternativi con qualche codizione di terminazione dipendente dalle configurazioni di terminazione dei sottocompiti.

FSMBehaviour. Questa classe esegue i sottocomportamenti in accordo con una macchina a stati finiti definita dell'utente. Ogni figlio rappresenta un'attività che può essere eseguita dentro ad uno stato e l'utente definisce le transizioni tra gli stati della macchina a stati finiti. Quando un stato S_j è completato il suo valore è restituito dal metodo onEnd() e può essere usato per selezionare la transizione da seguire e di conseguenza il prossimo stato S_{j+1}. Alcuni dei comportamenti figli possono essere registrati come stati finali, dopo la cui esecuzione il Behaviour termina. Per una descrizione più completa il manuale per i programmatori [6] di JADE rimanda alla documentazione delle API.

L'architettura degli agenti JADE.

In figura 4.7 illustriamo quella che potrebbe considerarsi l'architettura degli agenti JADE, nel seguito descriviamo gli elementi di cui si compone.

Scheduler of behaviour: è lo scheduler dei Behaviour che viene ereditato estendendo la classe agent, abbiamo spiegato come esso si basi su il cooperative scheduling.

Private message inbox: contiene i messaggi che sono arrivati all'agente. Abbiamo già descritto le primitive di ricezione che permettono di interagire con essa.

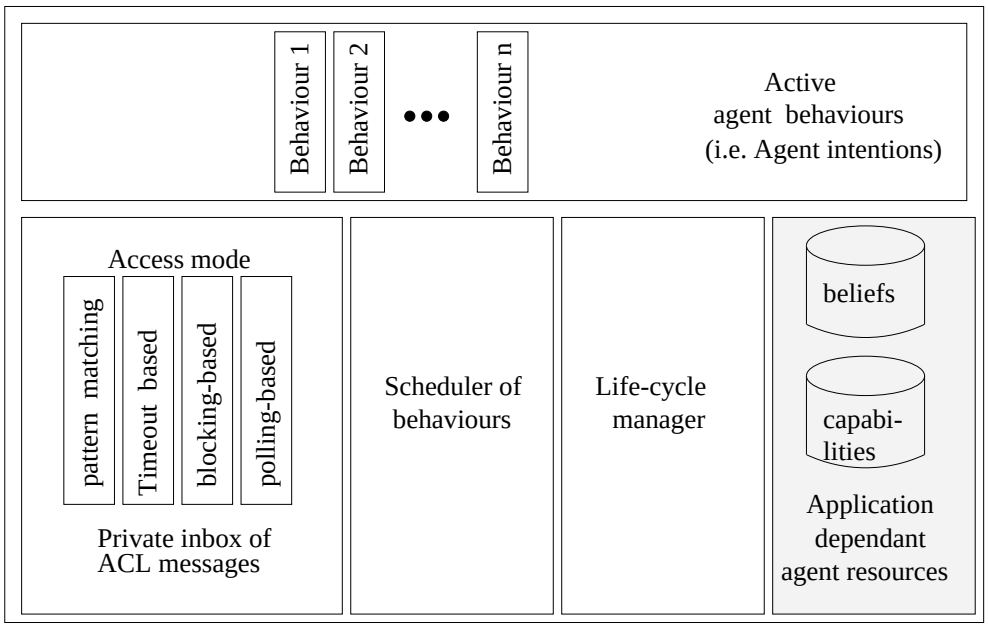

Figura 4.7: Architettura degli agenti JADE.

Life-cycle manager: è la componente che fornisce alla platform i metodi per gestire il ciclo di vita dell'agente.

Active agent behaviour: è l'insieme dei comportamenti dell'agente in esecuzione (schedulati).

Application dependent agent resouce: le componenti fornite dai programmatori degli agenti JADE ad esempio, le strutture dati necessarie agli agenti per svolgere i compiti per i quali sono stati progettati.

Vorremmo che si osservasse che questa architettura è in qualche maniera una meta-archiettura per agenti. Nella progettazione di JADE gli sviluppatori hanno cercato di non fare assunzioni circa l'architettura per agenti da utilizzarsi. Tutte le componenti che sono presenti, sono necessarie per assicurare il corretto funzionamento dell'agente una volta inserito nella platform JADE. In particolare l'organizzazione a Behaviour dei compiti dell'agente è stata introdotta per far fronte al multithreading offerto da Java ed i limiti al numero di agenti supportati. Gli stessi autori del tool fanno osservare in [8] che il modello di agente da loro realizzato è un modello molto generale che può essere facilmente specializzato per implementare architetture reattive o BDI. Questa caratteristica di JADE ci ha fatto riflettere sulla possibilità di sfruttare il tool per sviluppare il nostro D-CaseLP. In D-CaseLP avevamo supposto di lavorare su un modello di architettura per agenti generale e condiviso dalle varie architetture presenti in letteratura. Per questo

motivo, avevamo ipotizzato che gli agenti avessero una mailbox privata al quale accedessero con le primitive di comunicazione da noi individuate. Se confrontiamo tale modello con il modello di architettura adottato in JADE ci rendiamo conto che il secondo mantiene le ipotesi da noi fatte e potrebbe essere visto come una specializzazione della nostra architettura in cui si tiene conto dei vincoli introdotti dai limiti del multithreading offerto da Java.

4.3 Implementazione attraveso JADE di D-CaseLP

In questa sezione cercheremo di analizzare in che modo si possa utilizzare JADE per la realizzazione del tool D-CaseLP. Illustreremo come potrebbero essere ottenuti i punti del sottoinsieme del FRT iniziale presentato in figura 2.3, "arricchito" come illustrato in figura 4.1. Una volta illustrato il supporto offerto da JADE per la realizzazione dei punti funzionali contenuti nel FRT, evidenzieremo le ipotesi sotto le quali sarà possibile l'impiego di JADE ai nostri scopi.

4.3.1 Come realizzare con JADE i punti del FRT

Vediamo nel dettaglio il supporto che JADE ci offre per la realizzazione delle singole funzionalità di D-CaseLP.

Documentazione per l'installazione e utilizzo.

Abbiamo già fatto notare quanto sia importante produrre della documentazione se si vuole che il tool da noi realizzato trovi impiego al di fuori del nostro gruppo di lavoro. L'installazione e l'impiego di JADE è illustrato in due documenti in continuo aggiornamento: un manuale per l'amministratore [5] ed un manule per il programmatore [6]. Nel primo vengono trattate le questioni relative all'installazione, configurazione, ed esecuzione del tool, fornendo una descrizione su come cambiare l'implementazione del trasporto (il FIPA transport) o la codifica dei messaggi nell'esecuzione della platform. Il secondo illustra come programmare agenti in JADE, dettagliando alcuni aspetti descritti nella sezione precedente con esempi di codice. Come avevamo già previsto se l'implementazione fosse stata realizzata in CORBA, si può fare ampio riferimento a questa documentazione per gli aspetti che sono già stati sviluppati: uso tool, start-up, eccetera.

Supporto alla comunicazione.

Analogamente a ciò che abbiamo fatto per illustrare l'ipotesi d'implementazione in CORBA, illustriamo le idee su come sfruttare gli strumenti offerti da JADE per realizzare l'interfaccia di comunicazione presentata in 1.1.

Nel framework D-CaseLP abbiamo previsto due tipologie di agenti: gli Ag-Prolog e gli Ag-Java. Se non ci preoccupiamo delle differenze che l'utilizzo di Java e Prolog introducono nell'implementazione dell'interfaccia di comunicazione non ci sono grandi problemi alla realizzazione dell'interfaccia in termini di funzionalità JADE. Infatti, a parte una conversione fra i parametri, è possibile implementare le primitive D-CaseLP in termini di quelle offerte dalla classe Agent di JADE (si veda la presentazione delle primitive fatta in sezione 4.2.2). La conversione dei parametri consisterà nel tradurre messaggi o pattern di messaggi FIPA ACL espressi come stringhe in oggetti su cui chiamare le primitive JADE. Nelle librerie di JADE esiste una classe ACLParser che permette di fare l'analisi sintattica per messaggi FIPA ACL su una stringa ed ottenere un oggetto di tipo ACLMessage[14], mentre non sembra esistere un metodo che dato un messaggio (pattern) parzialmente istanziato FIPA ACL in forma di stringa restituisca il MessageTemplate con gli stessi campi fissati. Se non si vuole avere un pattern matching troppo elaborato, in cui ad esempio per lo stesso campo si possano esprimere più possibili valori e ci si limita ad un pattern matching in cui lo schema prevede l'indicazione di un solo valore per ogni campo possiamo pensare di implemetare da noi un metodo statico **ACLtoMessageTemplate** di una classe DCaseLPTool e realizzare l'interfaccia come segue:

send(msg). l'agente che vuole spedire il messaggio richiama la primitiva di JADE send. Il parametro msg è una stringa che rappresenta un messaggio FIPA ACL. La stringa viene tradotta in un oggetto di classe ACLMessage pronto per essere passato come argomento alla primitiva send di JADE. Tutto il resto della comunicazione è gestito in modo trasparente dalla platform JADE in aderenza alle specifiche FIPA.

receive_sync(msg). Il parametro msg è una stringa che esprime lo schema sul quale vogliamo applicare il pattern matching. Applicchiamo alla stringa il metodo da noi implementato ACLtoMessageTemplate ottenendo un JADE MessageTemplate e chiamiamo la blocking_receive() su questo oggetto.

receive_async(msg). Stessa implementazione della primitiva bloccante solo che invece che chiamare la primitiva JADE blocking_receive() chiamiamo semplicemente la receive().

Facciamo osservare che requisito a che si possa accedere alle primitive di comunicazione definite nella classe Agent delle API JADE è che gli agenti di D-CaseLP siano estensioni di tale classe.

[14]La conversione da oggetti messaggio FIPA ACL a stringhe non crea problemi in quanto nella classe ACLMessage è stato ridefinito il metodo toString(). Questo metodo permette di ottenere la codifica in stringa dell'oggetto a cui appartiene.

Diamo una definizione più precisa di come si potrebbe realizzare il metodo ACLtoMessageTemplate. La prima cosa che ci serve è un formato in cui esprimere il pattern, ispirandoci alla rappresentazione in forma di stringhe dei messaggi FIPA ACL abbiamo deciso di utilizzare il seguente:

```
(:nomecampo1 valorecampo1,
  ... ,
 :n-esimo-nomecampo n-esimo-valorecampo)
```

Strutturiamo il pattern come una tupla di coppie in cui la prima parte inizia con il carattere ":" ed è seguita dal nome del campo per cui si sta fissando un valore, dopo di che uno spazio con in valore al quale si sta fissando il pattern. Abbiamo convenuto che il simbolo * o l'assenza del relativo campo indichino che non si intende vincolare alcun valore. Le andate a capo nell'esempio non sono significative per il formato che stiamo esprimendo hanno il solo scopo di rendere più leggibile la descrizione. Ad esempio, supponiamo di avere una stringa che contiene la codifica del pattern come il seguente:

```
(:sender pippo, :performative *)
```

Tale stringa esprime un pattern in cui si hanno come selezionabili tutti i messaggi il cui mittente è "pippo", con una performative qualsiasi. Inoltre, visto che non si sono specificati i valori per tutti gli altri nomi campo non si esprime alcun vincolo su di essi. Illustriamo l'algoritmo che si può seguire per implementare il metodo statico **ACLtoMessageTemplate** di conversione da un pattern in forma testuale ad un oggetto di classe MessageTemplate:

1. inizializziamo un oggetto $\mathcal{P}$ di classe MessageTemplate senza alcuni vincolo sui campi di cui si può comporre un messaggio FIPA ACL, equivalente al pattern che è compatibile con tutti i messaggi;

2. consideriamo la stringa in input, se contiene un messaggio vuoto o in cui tutti i campi contengono il carattere * ritorniamo l'oggetto $\mathcal{P}$;

3. fino a quando il messaggio non è vuoto consideriamo il prossimo identificatore del campo (es. :sender) e la costante che rappresenta il valore di quel campo nel pattern di messaggio (es. un AID di un agente);

4. nel caso che il valore per quel campo sia * si va avanti nell'esaminare la stringa;

5. nel caso in cui il valore sia specificato si aggiunge con l'apposito metodo al pattern $\mathcal{P}$ il vincolo sul campo. Nella classe MessageTemplate esiste un metodo che aggiunge per ogni campo di cui si può comporre un messaggio FIPA ACL un vincolo per un determinato valore. Si ritorna al passo due.

Comunicazione via MTS e Directory Name Service (FIPA)

Una volta che si sono implementate le primitive di comunicazione D-CaseLP in termini di quelle offerte da JADE si ha accesso ai servizi di MTS. Ricordiamo infatti che il trasporto sul quale è spedito il messaggio è assolutamente trasparente agli agenti, ed è tutto gestito dalla platform. Poco importa che a fare le chiamate siano gli agenti D-CaseLP "travestiti[15]" da agenti JADE. In maniera analoga possiamo ragionare per i servizi di directory name service o quant'altro. Non sarà necessario che gli agenti D-CaseLP utilizzino particolari stratagemmi per interfacciarsi all'AMS o al directory facilitator, sono infatti in grado di interagire tramite spedizione di messaggi e utilizzare le ontologie standard per effettuare le richieste alle componenti.

Come distribuire gli agenti e dove recuperare il codice degli agenti.

Quello che vorremmo è avere una console che ci permetta di distribuire gli agenti su diverse macchine, indicando per ogni agente quale codice eseguire. Queste funzionalità sono parzialmente supportate da JADE.

Abbiamo spiegato che JADE offre la possibilità di far girare gli agenti su una platform distribuita composta da un main-container e da più container dislocati su macchine differenti. Ciò per cui non vi è supporto sono le operazioni di start-up dei container remoti. Per creare le parti che andranno a comporre la nostra platform distribuita è necessario accedere in locale alla macchina sulla quale si vuol far partire l'istanza di container o main-container ed eseguire da linea di comando l'istruzione relativa. Questo significa che se vogliamo eseguire una platform distribuita su due macchine di nome rispettivamente hostname1 e hostname2, avremo bisogno di collegarci alla prima delle due ed attivare il main-container con un comando del tipo

```
java jade.Boot -gui -host hostname1
```

e successivamente collegarci in remoto sul secondo calcolatore hostname2 ed eseguire il comando

```
java jade.Boot -container  -host hostname1
```

che attiverà il secondo container. Nel primo comando l'opzione -gui attiva l'agente Remote Monitoring Agent (RMA) che offre l'interfaccia grafica per il controllo di tutta la platform, mentre l'opzione -host specifica il nome del host su cui deve

[15] Con questo termine sottointendiamo che gli agenti D-CaseLP implementano l'interfaccia degli agenti JADE e la platform non è in grado di distinguere se sta gestendo un agente JADE o un agente D-CaseLP

essere attivato il main-container[16]. Nel secondo, l'opzione -container specifica che si sta attivando un container mentre -host indica su quale macchina è in esecuzione il main-container al quale unire il container che stiamo attivando. In entrambi i comandi viene eseguita la classe Java "Boot", appartenente al package "jade".

Si osservi quindi che per attivare le platform remotamente si fa affidamento ai servizi offerti dal sistema operativo quali eventuali telnet, ssh e simili. Una volta che i container sono attivati JADE offre tutti gli strumenti per attivare e controllare agenti remoti. Gli agenti si possono far partire sia al momento in cui si attivano le varie componenti della platform attraverso appositi opzioni da linea di comando sia da interfaccia grafica in un secondo momento. A titolo di esempio facciamo vedere come si attiva da linea di comando un agente di nome "pippo" al quale viene associato il codice Java della classe "primo" su un main-container che dovrà partire su "hostname1":

```
java jade.Boot -gui -host hostname1 pippo:primo
```

Visualizzazione delle comunicazioni fra gli agenti.

Per questa funzionalità c'è supporto attraverso quello che è lo Sniffer agent, che come abbiamo brevemente illustrato nella presentazione di JADE offre tutte le funzionalità grafiche per spiare le comunicazione tra gli agenti, anzi, permette di selezionare quali sorvegliare fra tutti gli agenti presenti nel MAS.

4.3.2 Ipotesi per l'utilizzo di JADE in D-CaseLP

Evidenziamo le ipotesi in parte taciute durante la presentazione della realizzazione dei punti del FRT che sono necessarie alla effettiva realizzazione della distribuzione degli agenti Ag-Prolog ed Ag-Java.

Estensione della classe Agent di JADE. La prima ipotesi necessaria per poter integrare gli agenti supportati da D-CaseLP con gli strumenti offerti da JADE è che gli agenti D-CaseLP implementino la stessa interfaccia offerta dagli agenti JADE. In questo modo, se non si contravviene al principio di information hiding e andando a rompere l'oggetto per vedere come si sono implementate le funzionalità, gli agenti Ag-Java o Ag-Prolog risultano in tutto per tutto equivalenti agli agenti JADE.

Architettura degli agenti in termini di Behaviour. La seconda ipotesi consiste nel poter strutturare gli agenti D-CaseLP in Behaviour. Si è già avuto modo di illustrare le motivazioni che stanno dietro la scelta di organizzare gli

[16] Il parametro -host per l'attivazione di main-containere può anche non essere specificato. In tal caso, la platform viene attivata sul host indicato dalla variabile di sistema LocalHost.

agenti JADE in Behaviour. Se si vuole che gli agenti D-CaseLP siano eseguiti concorrentemente, è necessario strutturare anche gli agenti Ag-Prolog e Ag-Java in termini di Behaviour.

Il modo più semplice per realizzare la prima ipotesi è progettare gli agenti D-CaseLP come estensione della classe Agent, riservandosi eventualmente di sovrascrivere l'implementazione di alcune delle operazioni originali della corrispondente iterfaccia. Questa ipotesi evidenzia un primo problema per l'implementazione degli agenti Ag-Prolog. Infatti, mentre è abbastanza semplice progettare degli agenti Ag-Java che siano estensione della classe Agent, non è così nel caso degli agenti Ag-Prolog, se non altro perché il concetto di ereditarietà ed estensione di classe è supportato dal paradigma ad agenti e non dal paradigma logico. Il problema sarà affrontato più dettagliatamente nel prossimo capitolo, il quale tratterà la progettazione dell'integrazione tra agenti logici Ag-Prolog e JADE. Per quanto riguarda la seconda ipotesi vorremmo ribadire ancora una volta che non vi sono grandi alternative per ottenere la concorrenza degli agenti sotto le ipotesi fatte. In uno dei primi articoli che trattano di JADE gli autori segnalano la possibilità di implementare altre politiche per lo scheduling dei Behaviour. In particolare accennano alla possibilità di implemetare uno scheduling di Behaviour interrompibili. Questo implicherebbe rispondere almeno alle seguenti domande.

- Come rappresentare lo stato dei Behaviour?
- Quale politica di schedulazione per l'interruzione?
- È possibile sfruttare alcune implementazioni della JVM che sono pensate per utilizzare i thread di sistema?

A parte la complessità legata alla rappresentazione dello stato dell'esecuzione dei Behaviour che potrebbero essere in parte risolti con l'ausilio della serializzazione, ci sembrava che la risoluzione di queste problematiche affrontassero questioni che vanno ben oltre questa tesi. Non c'è stato il tempo di approfondire questa soluzione, che se percorribile comporta un discreta mole lavoro.

Ricordiamo che abbiamo esplicitamente supposto che il tool D-CaseLP

1. possa dare supporto su diverse piattaforme;
2. non faccia ipotesi troppo restrittive sul tipo di architettura per agente che viene adottata, in particolare non fissi alcuna architettura;
3. offra strumenti per il controllo e il monitoring degli agenti.

Allo stato delle cose l'ipotesi uno giustifica l'impiego di Java come linguaggio d'implementazione del tool, questo significa non poter fare affidamento sui

servizi per il multithreading offerti dal singolo processore ma su quelli forniti da un'astrazione come quella fornita dalla Java Virtual Machine. Osserviamo che le ipotesi due e tre esprimono esigenze che potrebbero considerarsi in contrasto: da una parte abbiamo bisogno di avere una certa libertà nella scelta dell'architettura di cui dotare i nostri agenti, dall'altra ci farebbe comodo sapere come sono fatti dentro per monitorarne e controllarne il comportamento, il tutto per rendere semplici le cose con i limiti imposti dall'ipotesi uno. Non è possibile dire se la soluzione rappresentata dall'utilizzo dei Behaviour sia la soluzione ottima, senza addentrarci troppo in considerazione epistemologiche ci sembra di poter considerare l'architettura a Behaviour un buon compromesso.

4.3.3 Gli agenti Ag-Java

Concentriamoci ora sull'impatto che le due ipotesi precedenti hanno nella definizione degli agenti Ag-Java. Abbiamo capito che dovranno fornire la stessa interfaccia degli agenti JADE ed essere organizzati in Behaviour, a questo punto uno si potrebbe chiedere perché non si facciano coincidere gli agenti Ag-Java con gli agenti nativi di JADE. La questione non è così semplice: bisogna individuare più precisamente quali sono i requisiti necessari per poter effettuare una traduzione semiautomatica da Ag-UML agli agenti Ag-Java.

Lo studio di queste problematiche sarà trattato nei lavori successivi a questo, attualmente per alcuni aspetti sono oggetto dei lavori di Stefano Miglia e Alessandro Rolando. In particolare si sta valutando se la traduzione delle specifiche possa essere effettuata a partire da una libreria di specifiche di componenti basilari (es. interaction protocol, architetture per agenti, ecc), di cui si ha una implementazione. Se così fosse parte della libreria di JADE potrebbe tornare utile per arricchire la libreria di componenti di D-CaseLP, ma prima decidere di far confluire gli agenti Ag-Java negli agenti JADE è necessario valutare attentamente l'impatto che ciò ha sulle ipotesi ed i requisiti di D-CaseLP.

4.3.4 Gli agenti Ag-Prolog

Gli agenti logici soggiaciono agli stessi vincoli degli agenti Ag-Java, tratteremo nel prossimo capitolo lo studio del design di tali agenti dando una descrizione dettagliata di come si possano implementare agenti basati sul paradigma logico. Ciò che anticipiamo fin da ora è la possibilità di far interagire Java e Prolog ed incapsulare del codice Prolog all'interno degli agenti JADE.

4.3.5 Iter dagli agenti prototipali all'esecuzione usando JADE

In figura 4.8 abbiamo riportato lo schema dell'iter che permetterà a partire dagli agenti prototipali di ottenere l'esecuzione su run-time JADE.

Confrontando questo schema con quello fornito in figura 1.8 si può notare che il primo è sostanzialmente un'instanza del secondo. Le modifiche apportate consistono nel aver sostituito ciò che era indicato come il run-time D-CaseLP con il run-time offerto da JADE. In conseguenza di questa sostituzione la fase di preproccessing consisterà nell'includere in scheletri di agenti JADE da noi forniti il codice che si trova negli Ag-Prolog ed Ag-Java.

Allo stato attuale prevediamo che la libreria di scheletri di agenti dovrà contenere uno scheletro per gli agenti Ag-Prolog ed uno per gli agenti Ag-Java, in modo che a partire dalle informazioni sullo stato iniziale, il program e l'engine degli agenti prototipali sia possibile istanziare i rispettivi agenti JADE.

4.4 Confronto tra JADE e CORBA

In figura 4.9, utilizzando una tabella abbiamo riportato le funzionalità appartenenti al function refinement tree che avevamo costruito in fase di analisi per la parte di tool relativa la distribuzione e l'esecuzione degli agenti prototipali. Abbiamo cercato di evidenziare il supporto fornito da JADE e da CORBA per ognuna delle funzionalità che prevediamo di realizzare nella prima implementazione di D-CaseLP.

Abbiamo preparato una seconda tabella (figura 4.10) per le funzionalità che non sono inclusi negli intenti di prima realizzazione di D-CaseLP ma che potrebbero essere incluse nelle versioni successive e per le quali JADE offre già un supporto.

Le due tabelle evidenziano la distanza che intercorre tra CORBA e JADE. Vediamo di illustrare le considerazioni di "progetto" che ci hanno aiutato ad operare la scelta in maniera definitiva.

Considerazioni su JADE.

JADE è un tool più evoluto e completo di quello che potrà mai essere D-CaseLP con l'attuale forza lavoro. Negli scopi di questa tesi e negli obiettivi del progetto in cui è inserita questa tesi, non vi è tanto l'interesse a fornire un nuovo ambiente che competa con i tool esistenti, quanto permettere la distribuzione degli agenti Ag-Prolog e Ag-Java risultanti dall'applicazione della metodologia M-CaseLP che è in via di definizione nel lavoro inerente la prima parte del FRT illustrato in sezione 1.3.4.

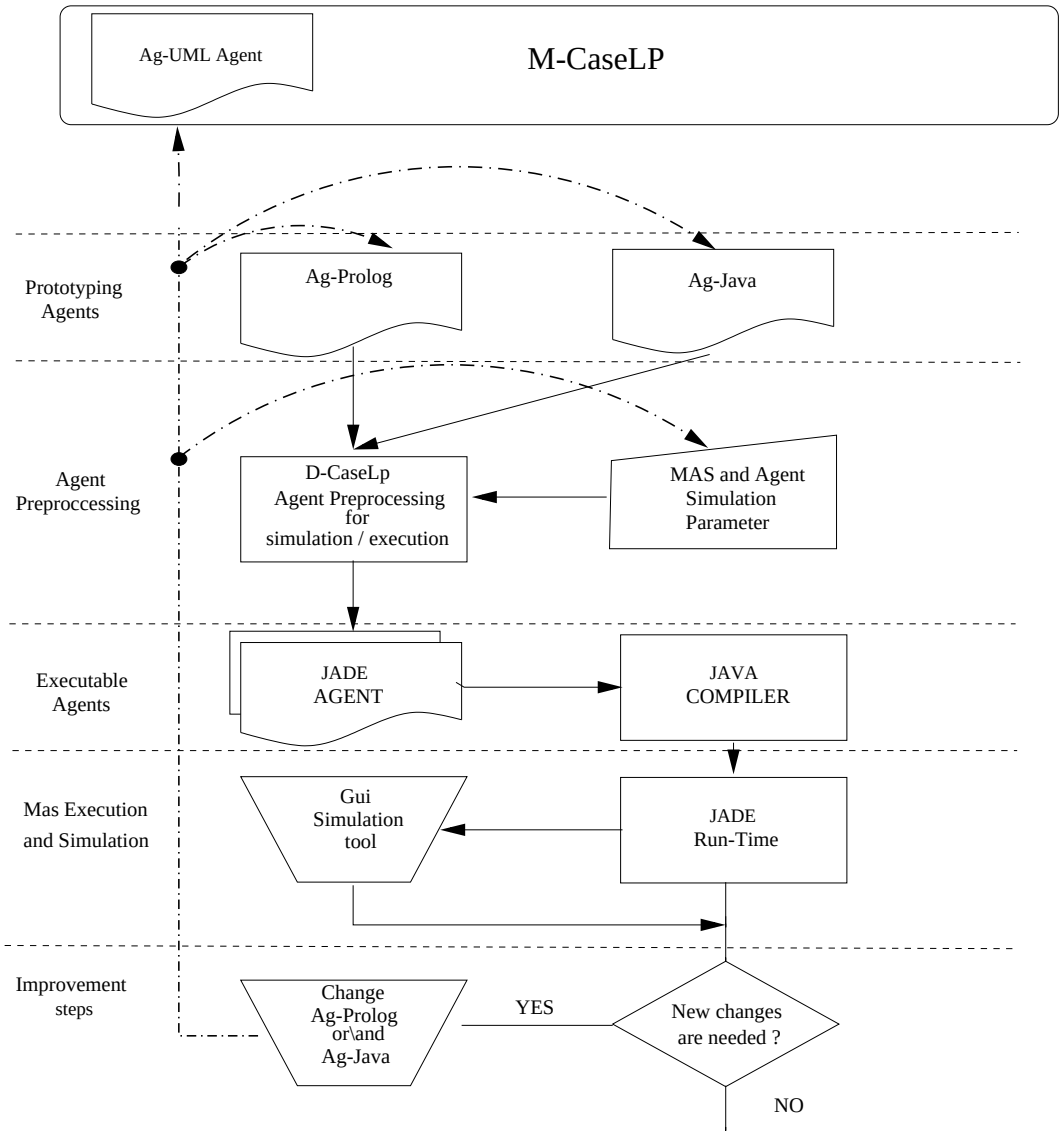

Figura 4.8: Dettaglio sulla l'iter di esecuzione del progetto D-CaseLP con l'ausilio di JADE.

Supporto per l'installazione degli strumenti offerti da D-CaseLP		
Funzionalità	**JADE**	**CORBA**
Documentazione per l'installazione	disponibile per eventuali riferimenti la documentazione d'installazione/utilizzo per il programmatore	disponibile per eventuali riferimenti la documentazione d'installazione/utilizzo per il programmatore
Supporto per l'implementazione del prototipo MAS		
Funzionalità	**JADE**	**CORBA**
Supporto alla comunicazione	peer-to-peer, FIPA compatibile	client-server
Directory Name Service (FIPA)	supportato con AMS secondo le specifiche FIPA	possibile utilizzo dei servizi di CosName
Comunicazione via MTS (FIPA)	implementato, con un certo numero di MTS già realizzati	nessun supporto
Supporto al monitoraggio e controllo degli agenti		
Funzionalità	**JADE**	**CORBA**
Configurazione della distribuzione degli agenti	disponibile, una volta che si sono fatti partire i platform container distribuiti	nessuna
Indicazione su dove recuperare il codice degli agenti	disponibile, con possibilità di associare un nome all'agente (vi è implicitamente il supporto per il concetto di classe di agente)	nessuna
Visualizzazione delle comunicazioni fra gli agenti	fornita attraverso lo Sniffer Agent	nessuna

Figura 4.9: Confronto tra il supporto offerto da JADE e CORBA.

Supporto per l'implementazione del prototipo MAS		
Funzionalità	**JADE**	**CORBA**
Integrazione semantica	supporta le ontologie	nessuna
Supporto alla mobilità	disponibile	nessuna

Figura 4.10: Confronto tra il supporto offerto da JADE e CORBA su funzionalità che protrebbero essere incluse in un futuro prossimo.

Si tenga presente nessuna delle specifiche fino ad ora prodotte dalla FIPA ha raggiunto lo status di "standard", come abbiamo già osservato i documenti che si trovano in fase più avanzata sono nella fase "experimental" quindi ancora soggetti a molte variazioni. Mantenere traccia delle variazioni ed implemetare le modifiche prescritte dai nuovi documenti non è ipotizzabile con l'attuale forza lavoro di cui il gruppo dispone. Adottando gli strumenti da terzi il problema degli aggiornamenti rispetto le specifiche FIPA ricade per la maggior parte sul gruppo che fornisce JADE.

JADE non ha una metodologia propria ed aderendo alle specifiche FIPA va nella direzione in cui si sta muovendo il nostro gruppo, fornendo delle facilitazioni che permettano di integrare facilmente gli agenti prototipali con l'implementazione degli agenti in grado di girare in JADE si può effettivamente avvicinare alla metodologia M-CaseLP ad un pubblico più ampio.

L'adozione di JADE ci manleva da un po' di lavoro. Questo avvicina temporalmente la possibilità di concentrare gli sforzi del gruppo sulla traduzione semiautomatica di specifiche ed aspetti più innovativi del tool relativi alla simulazione.

4.5 Conclusione

Alla luce di ciò che risulta dal confronto tra il supporto fornitoci da JADE rispetto quello offerto da CORBA, e tenuto conto delle motivazioni inerenti al progetto, ci sembra che la scelta migliore sia adottare uno strumento già ampiamente testato. Abbiamo quindi definitivamente optato per JADE sposando condiderazioni pragmatiche volte a non rifare lavoro che è già stato fatto da altri. È evidente che si sono alcune questioni aperte sull'integrazione fra JADE e gli agenti Ag-Prolog ed Ag-Java. Come operare l'integrazione con gli agenti Ag-Prolog sarà specificato nel prossimo capitolo, mentre per poter leggere come saranno integrati gli agenti Ag-Java sarà necessario aspettare la conclusione dei lavori svolti da Stefano Miglia ed Alessandro Rolando.

Capitolo 5

Agenti logici

In questo capitolo illustriamo come sia possibile ottenere agenti logici Ag-Prolog che siano eseguibili dalla platform JADE. Innanzi tutto daremo una breve rassegna sulle implementazioni Prolog che offrono la possibilità d'integrare Java e Prolog, ponendo particolare attenzione al framework tuProlog. Una volta individuata l'implementazione a nostro avviso più adatta, descriveremo attraverso alcuni use case l'analisi dei requisiti di cui dobbiamo fornire l'estensione che permetterà di eseguire gli Ag-Prolog in JADE. Per finire daremo un class diagram con il quale descriveremo le classi necessarie per ottenere gli agenti logici.

5.1 Agenti logici in CaseLP.

Il tool di simulazione CaseLP è interamente realizzato in Prolog ed il paradigma logico è utilizzato sia per programmare l'*engine* che esprimere il *program* degli agenti. CaseLP offre alcuni linguaggi di specifica che si pongono a diversi livelli di astrazione, ogni linguaggio è proposto come strumento per facilitare il lavoro dell'utente nell'applicare un passo della metodologia proposta. Nel seguito diamo una brevissima introduzione ai linguaggi disponibili in CaseLP ed il loro ruolo sulla base delle ultime evoluzioni del lavoro illustrate in [43]. Nel continuare la lettura si tenga presente che la seguente presentazione ha l'obiettivo di dare un'idea del supporto fornito e non ha la pretesa di spiegare in poche righe quello che è un framework molto articolato, per una descrizione completa si faccia riferimento anche a [66].

La metodologia descritta in CaseLP individua un utilizzo del tool che possiamo pensare suddiviso in quattro fasi.

Fase 1: modellazione. In questa fase partendo dalla metodologia illustrata in sezione 1.2.3 vengono definiti:

- i componenti indipendenti dal dominio (Architetture per agenti, Ruoli, Ontologie) attraverso linguaggi di specifica che comprendono $\mathcal{E}_{hhf}$, HEMASL, UML;
- la struttura statica del MAS ovvero come le componenti vengono messe insieme attraverso il linguaggio CaseLP-MASadl;
- la definizione dinamica del MAS attraverso la specifica dello stato e del program di ogni istanza di agente. Per la definizione del program e dello stato degli agenti sono stati definiti diversi linguaggi di specifica.

Fase 2: verifica e validazione. In questa fase è possibile effettuare verifica e validazione di proprietà che devono essere soddisfatte dal MAS attraverso le caratteristiche offerte dal paradigma logico.

Fase 3: implementazione. Dopo che il MAS è stato completamente specificato sia negli aspetti statici che negli aspetti dinamici lo sviluppatore deve implementare un prototipo. La coerenza tra le fasi di specifica ed il prototipo allo stato attuale è responsabilità dello sviluppatore: non sono stati ancora realizzati tool automatici di traduzione delle specifiche.

Fase 4: esecuzione. Il supporto offerto da CaseLP in questa fase è legato ai strumenti di simulazione (CaseLP simulator) e monitoring (CaseLP visualizer) offerti dall'ambiente CaseLP.

Per ognuna delle fasi precedenti viene fornito o è stato studiato un supporto nella forma di linguaggi di specifica, traduttori da un linguaggio ad un altro e tools.

Diamo una brevissima introduzione ai linguaggi che sono stati sviluppati nel contesto del progetto.

5.1.1 ProlAg

Una volta che il MAS è stato specificato è necessario implementare in un linguaggio omogeneo le differenti componenti.

Gli agenti CaseLP, devono in qualche modo andarsi ad interfacciare con l'ambiente di simulazione. È quindi necessario prevedere un linguaggio di implementazione in cui tradurre omogeneamente almeno le specifiche relative allo stato ed il program degli agenti ed implementare il relativo engine.

L'ambiente di simulazione ed esecuzione di CaseLP è stato sviluppato utilizzando il paradigma logico inizialmente dandone un'implementazione nel linguaggio ECLiPSE [21], e successivamente con SICStus Prolog [55].

ProlAg è definito come lo standard Prolog esteso con predicati che permettono aggiornamento dello stato dell'agente in maniera *safe* e predicati per la comunicazione tra agenti. L'impiego di ProlAg permette di astrarre da quelle che sono

le estensioni per la comunicazione proprietarie di SICStus Prolog ed ECLiPSE, permettendo riutilizzare engine stato e program degli agenti specificati indipendentemente dall'implementazione Prolog che si è utilizzata.

5.1.1.1 Predicati per modificare lo stato

assert_state(Fact) asserisce *Fact* nello stato dell'agente. L'implementazione è *Safe*, nel senso che qualora il predicato in cui viene effettuata la assert fallisse, il fatto viene rimosso dallo stato.

retract_state(Fact) rimuove il primo fatto che unifica con il *Fact*. Anche questo predicato è *Safe*, nel caso in cui il predicato in cui viene effettuata la retract fallisse, il fatto tolto verrebbe riasserito.

5.1.1.2 Predicati per la comunicazione

sync_receive(Message) blocca l'agente che ha effettuato la chiamata di questo predicato fino a quando non arriva nella mailbox dell'agente un messaggio che unifica con *Message*.

async_receive_one(Msg_input,Msg_output) ricerca nella mailbox dell'agente che richiama questo predicato un messaggio che unifichi con *Msg_input* , se un cosiffatto messaggio è presente nella mailbox, viene unificato con *Msg_output*, altrimenti *Msg_output* viene unificato con **no_message**.

async_receive_all(List_of_message) prende tutti i messaggi, nella mailbox dell'agente chiamante, e le unifica nella lista *List_of_Message*.

send(Receiver,Message) spedisce il *Message* dall'agente chiamante all'agente *Receiver*. In CaseLP questo era realizzato utilizzando le primitive *Safe* per aggiornare lo stato asserendo il *Message* nella mailbox del *Receiver*.

Il linguaggio "target" del processo di traduzione da specifiche ad alto livello in specifiche eseguibili è ProlAg, con esso è possibile esprimere l'engine degli agenti.

In figura 5.1 sono schematizzati i linguaggi per il quale si fornisce una traduzione o un interprete verso ProlAg.

5.1.2 AgentRules

Questo linguaggio permette di specificare il comportamento di una classe di agenti con regole del tipo *event-condition-action* per il comportamento reattivo dell'agente o *condition-action* per esprimere il comportamento proattivo dell'agente. Combinando l'utilizzo dei due tipi di regole è possibile esprimere agenti ibridi.

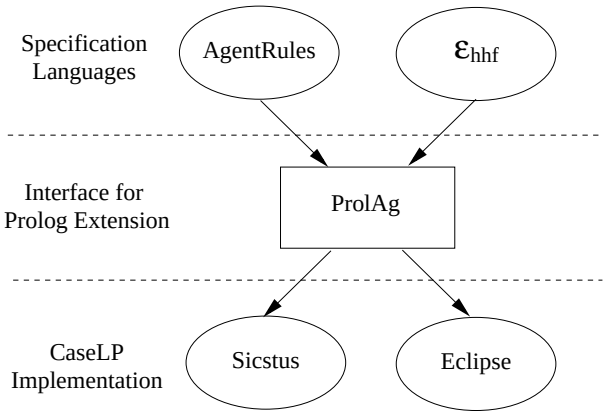

Figura 5.1: Uno schema per la traduzione in codice eseguibile da CaseLP.

AgentRules essendo basato sulla definizione di regole è un linguaggio che ci permette di specificare con facilità lo stato ed il program di un agente con una sintassi user-friendly. Forniamo in figura 5.2 la sintassi formale affinché sia possibile capire come scrivere le specifiche utilizzando AgentRules. Non è difficile implementare una traduzione delle specifiche AgentRules verso ProlAg: si tratta essenzialmente di riscrivere le regole che abbiamo espresso con la sintassi presentata prima in predicati ProlAg. Ciò che otterremo sarà qualcosa nella forma illustrata in figura 5.3.

Facciamo notare il risultato della traduzione è suddivisa in tre sezioni:

Initial state: stato iniziale definito tramite fatti ground del tipo **fact(B)**, in cui il termine B esprime una credenza *belief* dell'agente.

Program: programma dell'agente definito come un insieme di regole *event-condition-action* o *condition-action* espresse rispettivamente con in predicati **reactive_rule** e **proactive_rule**. Le regole vengono tradotte secondo le associazioni indicate in tabella 5.4.

Auxiliary predicates: sono clausole corrispondenti a procedure che possono essere utilizzate per verificare condizioni ausiliarie.

Una volta tradotte le regole è necessario che ci sia un programma che le interpreti.

Successivamente la traduzione delle regole i predicati potranno essere interpretati da un meta-interprete simile a quello definito di seguito:

```
engine :-
async_receive_one(Msg_input, Msg_output),
{Msg_output== no_message -> actions1=[]
```

InitialState ::= `nil` | BeliefsList

BeliefsList ::= `Belief` | `Belief`; BeliefsList

Program ::= **program** InitialState ReactiveRulesList ProactiveRulesList **endprogram**

ReactiveRulesList ::= `nil` | ReactiveRule | ReactiveRule; ReactiveRulesList

ReactiveRule ::= **on message** Msg **check** Condition **do** ActionsList

ProactiveRulesList ::= `nil` | ProactiveRule | ProactiveRule; ProactiveRulesList

Msg ::= Performative { **content**: Content; **sender**: Sender; **receiver**:Receiver; }

Condition ::= StateCondition **and** AuxiliaryCondition

StateCondition ::= `true` | Goal

AuxiliaryCondition ::= `true` | Goal

Goal ::= A | A **and** Goal | A **or** Goal

ProactiveRule ::= **check** Condition **do** ActionsList

Action ::= `assert_state(Belief)` | `retract_state(Belief)` | `send`(Msg, `Receiver`)

Figura 5.2: Sintassi formale per AgentRules

```
%%%%%%%%%%%%%%% Initial State %%%%%%%%%%%%%%%%%%
fact(B_1).
fact(B_2).
... ... ...
fact(B_n).
%%%%%%%%%%%%%%%%%%% Program %%%%%%%%%%%%%%%%%%%
reactive_rule(Message_1, Condition_1, Action_1).
... ... ...
reactive_rule(Message_m, Condition_m, Action_m).
proactive_rule(Condition_m+1, Action_m+1).
... ...
proactive_rule(Condition_m+k, Action_m+k).
%%%%%%%%%%%% Auxiliary predicates %%%%%%%%%%%%%%
p_1(..):-
... ... ...
p_s(..):-
```

Figura 5.3: La traduzione delle istruzioni AgentRules.

Regola AgentRules	Predicato
on message Message **check** Condition **do** Actions	reactive_rule(Message, Condition, Actions)
check Condition **do** Actions	proactive_rule(Condition, Actions)

Figura 5.4: Associazione tra regole AgentRules e predicati

```
;
select_reactive_rule(Msg_output, Actions1)
),
select_proactive_rule(Actions2),
choose(Actions1, Actions2, Actions),
execute(Actions).
```

Facciamo osservare che scrivere lo stato e il program in AgentRules, definirne la traduzione in ProlAg e fornire il meta-interprete, equivale a definire un'architettura per agenti logici.

5.1.3 CaseLP-MASadl

È un linguaggio che permette di specificare l'architettura del MAS, in particolare in CaseLP erano state individuate due tipologie di agenti:

Logical Agent forniscono il controllo e la coordinazione dei componenti del MAS, grazie alle loro complesse capacità di ragionamento,

Interface Agent forniscono l'intefaccia fra i moduli esterni e gli agenti del MAS.

La sintassi del linguaggio permetteva di specificare a quale delle due classi ogni agente appartenesse, le classi di agenti presenti nel MAS e le relazioni tra di esse, i servizi richiesti, forniti, il tipo di comportamento associata alla classe di agenti. Per una trattazione più completa si consiglia di leggere [16], nel quale viene illustrata anche la sintassi del linguaggio formalizzata in una BNF.

5.1.4 $\mathcal{E}_{hhf}$

Il linguaggio logico lineare $\mathcal{E}_{hhf}$ [19] è un linguaggio eseguibile per modellare sistemi concorrenti con gestione di risorse finite basato sul linguaggio Forum [44]. $\mathcal{E}_{hhf}$ è un linguaggio basato su logica multiset che combina caratteristiche di linguaggi logici estesi come λProlog, cioè goals con implicazione e quantificazione universale con la nozione di formule come risorse che è alla base della logica lineare.

5.1.5 HEMASL

HEMASL è un linguaggio di specifica procedurale per agenti che può essere utilizzato per definire un agente ad ogni livello di astrazione. La scelta di un linguaggio imperativo è stata fatta poiché esso è maggiormente vicino alle conoscenze dello sviluppatore industriale, solitamente poco familiare con il paradigma logico. HEMASL si pone su livelli similari di $\mathcal{E}_{hhf}$ fornendo un sistema completo per la specifica di un agente.

5.1.6 Il CaseLP Visualizer: un'interfaccia grafica per la simulazione

Il CaseLP Visualizer [50] fornisce una documentazione sugli eventi accaduti a livello di agente durante l'esecuzione del MAS. In accordo coi bisogni dello sviluppatore, il codice di un agente può essere automaticamente "instrumentato" dopo che è stato caricato. La "Instrumentazione" aggiunge delle sonde nel codice dell'agente; gli eventi correlati al cambiamento di stato e/o ai messaggi scambiati possono essere raccolti e registrati per un esame on-line o off-line. Sono fornite finestre grafiche per inizializzare la mailbox degli agenti o per fissare dei parametri della simulazione (lunghezza, granularità della visualizzazione on line).

La figura 5.5 descrive la finestra che consente l'inizializzazione della mailbox di un agente; la traccia on-line della esecuzione è mostrata in figura 5.6 dove gli eventi e lo stato relativi ad ogni agente vengono mostrati. La figura 5.7 mostra la traccia della esecuzione off-line, con eventi e messaggi mostrati sulle linee orizzontali (agenti), in corrispondenza dell'istante in cui accadono (dimensione verticale, tempo). I dettagli di ogni evento possono essere ottenuti cliccandovi sopra. La finestra che si apre è mostrata in figura 5.8. Durante la simulazione vengono mostrati dati riguardanti gli agenti "Instrumentati". Al termine della simulazione un traccia completa di tutti gli eventi che sono stati selezionati può essere visualizzata. La "Instrumentazione" è completamente indipendente dall'esecuzione, e quindi non influenza futuri cambiamenti del supporto di esecuzione.

5.1.7 Comandi di shell per l'esecuzione della simulazione

Se lo sviluppatore preferisce utilizzare comandi testuali, egli può utilizzare un insieme di primitive per eseguire la simulazione. Le primitive messe a disposizione sono

- `compile_agent(List_modules_dynamic_predicates_files, List_state_files, List_behaviour_files, Agent_name, Compiled_file_name)`:

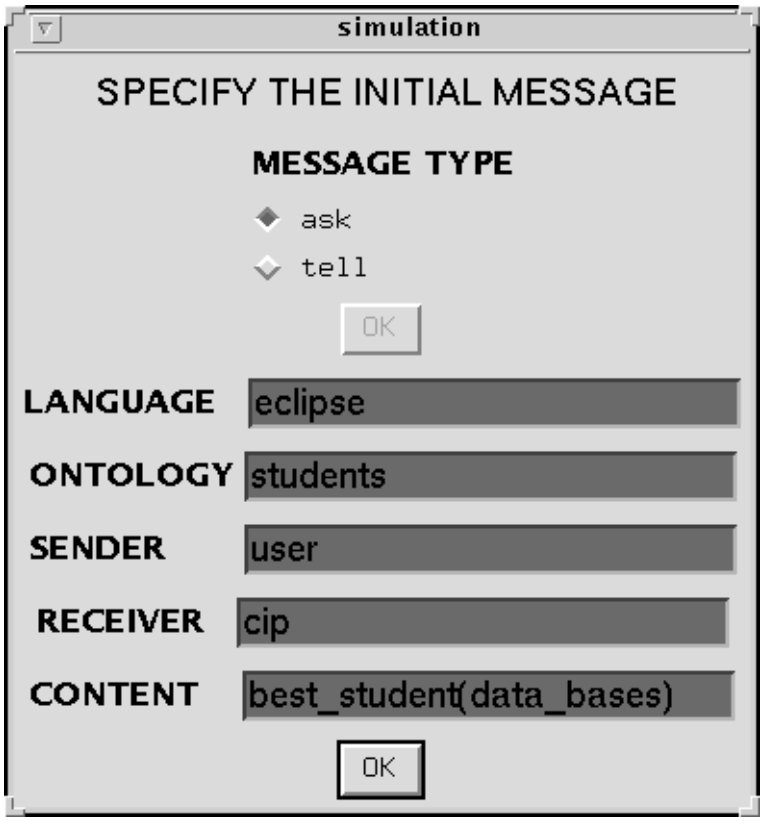

Figura 5.5: Finestra per l'inizializzazione delle mailboxes.

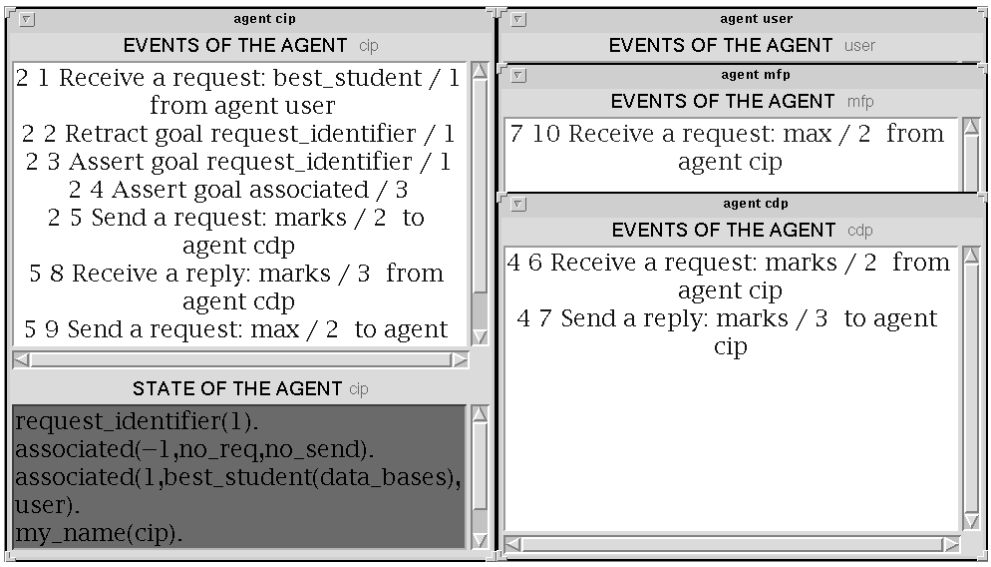

Figura 5.6: finestra che descrive l'esecuzione on line.

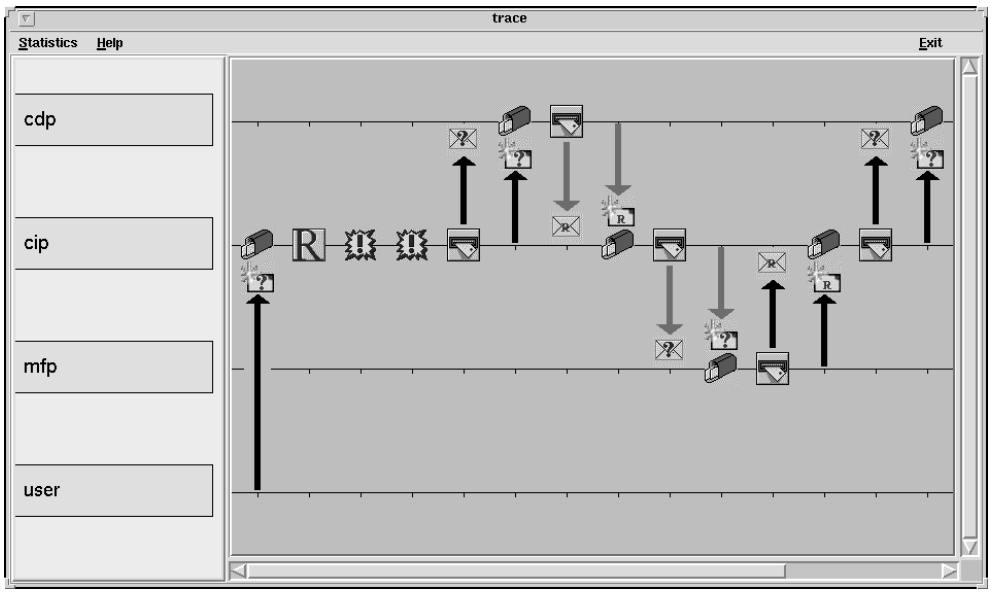

Figura 5.7: Traccia off-line delle esecuzioni.

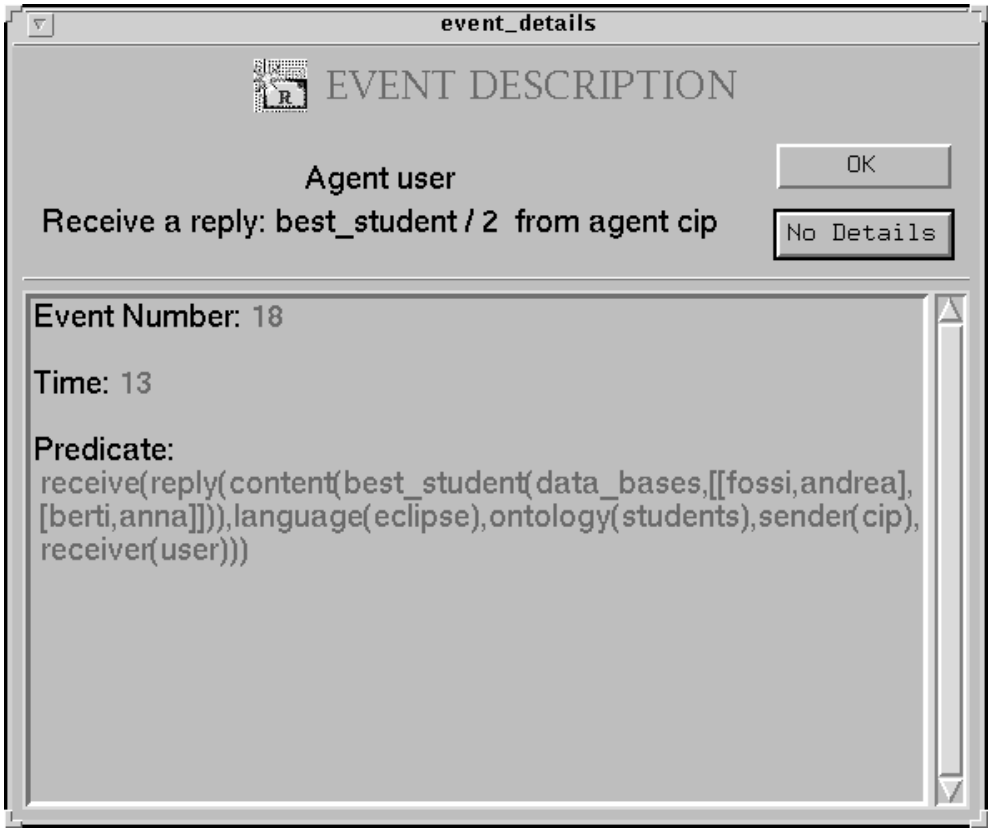

Figura 5.8: Dettagli di un evento tracciato.

Il primo argomento è una lista di files nei quali sono contenute le dichiarazioni dei moduli esterni che l'agente dovrà usare e la lista dei predicati dinamici.
Il secondo argomento è la lista dei files nei quali sono contenuti i fatti che rappresentano lo stato iniziale dell'agente.
Il terzo argomento è la lista dei files che contengono le regole che governano l'esecuzione dell'agente (è necessario definire il predicato "interpret" per ogni agente, e tutti i predicati ausiliari).
Il quarto argomento è il nome dell'agente (le primitive per inviare messaggi hanno un argomento che è il nome dell'agente a cui mandare il messaggio; tale nome deve essere unico).
Il quinto argomento, infine, è il nome del file che verrà generato a seguito di questo processo di compilazione. Tale file contiene il codice completo dell'agente, e verrà usato dal predicato **load** quando l'agente verrà caricato nel sistema.

- `load(File_name, Agent_name)`: carica l'agente specificato dal file File_name, assegnandogli il nome Agent_name

- `initialize(List_of_initial_messages)`: inizializza le mailbox degli agenti del sistema.

- `global_parameters(Iter, Min_time_unit, Out, Verbose)`: specifica i parametri della simulazione; Iter specifica il numero di iterazioni da eseguire nella simulazione, Min_time_unit specifica l'unità di tempo base della simulazione ossia la durata di ciascuna iterazione, Out è il nome del file in cui tutti i messaggi scambiati vengono registrati, Verbose è un flag che serve per decidere se tutti i messaggi scambiati nel sistema devono essere mostrati a video durante la simulazione.

- `net_parameters(From_ag, To_ag, Fail_rate, Min_late, Max_late, Unit)`: seleziona i parametri di comunicazione tra ogni coppia di agenti presenti nel sistema; From_ag e To_ag specifica la coppia di agenti, Fail_rate specifica quanti messaggi su 100.000.000 sono persi per problemi di comunicazione, Min_late e Max_late e Unit specificano il minimo ed il massimo tempo di ritardo sul canale espresso in Unit. Se per una coppia di agenti non sono specificati parametri la comunicazione è immediata e totalmente affidabile.

- `start_simulation`: è il comando che viene utilizzato per far partire la simulazione. La simulazione è eseguita tramite uno scheduler round-robin che chiama il predicato cycle di tutti gli agenti.

- `get_statistics`: è utilizzato per raccogliere tutte le informazioni sulla simulazione stessa.

5.2 Implementazioni Prolog

Prima di presentare la progettazione degli agenti logici analizziamo alcune delle implementazioni Prolog che si trovano in rete allo scopo di capire quali sono le possibilità di integrazioni tra Prolog e Java. Nonostante Prolog sia uno dei linguaggi tradizionali dell'intelligenza artificiale, dopo la popolarità acquisita durante gli anni '80 con i progetti correlati alle architetture di quinta generazione non ha trovato molto spazio fuori dagli ambienti accademici. Navigando sul World Wide Web sembra che negli ultimi anni ci sia un certo interesse a fornire interpreti e compilatori Prolog in grado di integrare funzionalità logiche con codice Java. In tabella[1] 5.1 abbiamo raccolto un certo numero d'implementazioni Prolog cercando di evidenziare, quali di essi sono compilatori e quali interpreti, in che forma sono forniti i motori inferenziali su cui si basano (binario, libreria), il tipo di licenza con cui è distribuito il software, l'esistenza o meno di una interfaccia verso Java.

Name		**Type**	**Licence**	**Java interface**
SICStus Prolog	[55]	comp/bin	commercial	Jasper
SWI-Prolog	[56]	comp/bin	free software	JPL
binprolog	[9]	comp	commercial	bidirectional via library
tuProlog*	[57]	int/lib	free for no profit	bidirectional via library
Jinni*	[36]	interpreter	commercial	buit-in (bidirectional)
jprolog*	[37]	interpreter	free software	Java calls Prolog
GNU PROLOG	[33]	comp	GPL	none, (C only)
MINERVA*	[45]	comp	commercial	built-in (bidirectional)

Tabella 5.1: Confronto implementazioni di Prolog.

Per operare una scelta in un insieme piuttosto eterogeneo di implementazioni di Prolog, discutiamo quali condizioni debba offrire l'implementazione candidata all'impiego in D-CaseLP.

Disponibilià della implementazione per diversi sistemi operativi: visto che una delle motivazioni che ci aveva orientato alla scelta di Java come linguaggio d'implementazione dei tool da noi progettati era la possibilità di fornire

[1] Abbreviamo compilatore con comp, binario con bin, interprete con int, libreria con lib, traduttore con trad, l'asterisco indica che il linguaggio d'implementazione è Java.

strumenti indipendenti dai particolari sistemi operativi, vorremmo evitare d'introdurre vicoli sul tipo di macchina o sul sistema operativo utilizzato nell'integrazione fra Java e Prolog.

Una interfaccia Java-Prolog bidirezionale: per eseguire gli agenti logici in JADE avremo bisogno di fare riferimento a oggetti Java all'interno di codice Prolog e dimostrare predicati Prolog all'interno del codice Java. In particolare, necessiteremo di un'interfaccia da Java a Prolog per includere gli agenti logici negli agenti di JADE e dell'interfaccia da Prolog a Java per richiamare le primitive di comunicazione offerte da JADE all'interno degli agenti logici. Questo implica che gli strumenti con cui integreremo agenti logici e agenti JADE dovranno offrire un'interfaccia bidirezionale Prolog-Java.

Software con licenza di utilizzo: essendo un progetto in ambito universitario è nostro interesse che il software, e le estensioni da noi prodotte siano il più possibile utilizzate e provate da altri. Prerequisito affinché questo sia possibile, è che il software abbia una licenza che permetta di utilizzarlo senza troppi oneri, almeno nei casi in cui il suo impiego non sia a fini di lucro. Questo requisito al contrario dei precedenti non è determinante nella scelta dell'implementazione Prolog ed è da considerarsi solo come una preferenza.

5.2.1 Tipi di interfacce Prolog-Java

Abbiamo già evidenziato l'esigenza di avere un'interfaccia bidirezionale Prolog-Java, in questa sezione confrontando la tecnologia offerta dalle diverse implementazioni vorremmo entrare un po' più nel dettaglio in quelle che sono le modalità con cui l'integrazione viene effettuata. Analizzando le implementazioni che abbiamo elencato in tabella 5.1 è possibile rendersi conto che l'integrazione di PrologJava avviene sostanzialmente per due strade:

- attraverso la Foreign Language Interface (FLI);
- utilizzando interpreti scritti in Java.

Iniziamo con illustrare le caratteristiche della FLI. Le interfacce verso altri linguaggi in Prolog sono virtualmente disponibili in tutte le implementazioni commerciali, esse pur differendo nei particolari si basano sull'idea di associare durante la fase di linking un file oggetto del linguaggio che si vuole integrare all'eseguibile del motore Prolog. È generalmente possibile anche il viceversa, quasi tutte le implementazioni forniscono una libreria con la quale è possibile accedere alle funzionalità del motore inferenziale della particolare implementazione Prolog.

Su questo tipo di integrazione sono basate le interfacce verso Java di SICStus Prolog (Jasper) e di SWI-Prolog (JPL), ed in generale di tutte le implementazioni che forniscono un compilatore per il linguaggio. Sempre all'interno di quella che potrebbe essere la filosofia vicina alle FLI vi sono alcune implementazioni che per integrare Prolog in Java utilizzano la la Java Native Interface (JNI) la quale permette d'includere del codice nativo (C, C++). Tale espediente permette di recuperare librerie di integrazione verso i motori Prolog che erano state pensate per il linguaggio C all'interno di Java. L'integrazione con queste modalità, passando per il codice oggetto pone dei limiti alla compatibilità con sistemi operativi diversi da quelli in cui il codice è stato compilato, perdendo di fatto i vantaggi di portabilità di Java. Inoltre, eseguendo alcuni test in SICStus Prolog utilizzando Jasper, ci siamo accorti che le integrazioni di questo tipo hanno dei limiti sul numero di motori Prolog attivabili per processo. Sia Jasper che JPL permettono di includere un solo "motore logico" per JVM, mentre noi utilizzando JADE avremmo bisogno di un motore logico per ogni thread che rappresenta un agente.

La seconda modalità d'integrazione vede l'impiego di interpreti al posto di compilatori. In generale questa strada risulta più adatta ai nostri scopi in quanto offre soluzioni portabili per l'integrazione JavaProlog e non soffre dei limiti sul numero di istanze di motori logici attivabili per processo o thread.

Ci sono svariati interpreti Prolog scritti in Java alcuni di essi offrono funzionalità d'integrazione attraverso un'interfaccia bidirezionale assolutamente portabile; noi ne abbiamo individuato uno che sembra particolarmente versatile e leggero: tuProlog. Le funzionalità offerte da tuProlog per l'integrazione sono discusse nella sezione 5.2.2.

5.2.2 tu**Prolog.**

Descriviamo in questa sottosezione sulla base delle informazioni contenute in [20, 2] le caratteristiche principali di tuProlog. tuProlog è stato progettato con tre principali obiettivi:

- offrire un motore Prolog minimale, efficiente e aderente alle specifiche ISO in un oggetto Java accessibile con una semplice interfaccia;
- dare la possibilità di estendere il nucleo minimale del linguaggio con librerie dinamiche contenenti nuovi predicati;
- offrire gli strumenti per poter interagire e controllare l'interazione con componenti Internet.

tuProlog offre un interprete Prolog realizzato per intero in Java che è stato ideato appositamente per costruire componenti "intelligenti" che siano facili da utilizzare, leggeri, scalabili, staticamente e dinamicamente configurabili ed interoperanti.

L'interprete viene fornito sotto forma di un motore inferenziale Prolog che chiameremo tuProlog Virtual Machine (VM). Ogni motore è istanza di una classe Java. Il fatto che il motore consista in un oggetto Java permette di avere un numero potenzialmente illimitato di istanze che possono essere attivate da uno stesso processo eventualmente su più thread.

Chiameremo il motore con caratteristiche minimali core tuProlog, ogni motore è configurato indipendentemente dagli altri ed a seconda delle particolari esigenze può essere integrato con componenti non Prolog sotto forma di oggetto Java. Le componenti tuProlog possono interagire con altre componenti non necessariamente logiche attraverso il consueto schema comunicativo client-server utilizzando strumenti come RMI e CORBA. In alternativa, è possibile utilizzare approcci di comunicazione peer-to-peer utilizzando un modello di comunicazione a blackboard[2] in cui le componenti comunicano aggiungendo e togliendo tuple in uno spazio comune. Questo tipo di comunicazione può risultare particolarmente versatile per coordinare e la favorire la cooperazione tra componenti, siano esse oggetti Java o VM tuProlog.

Integrazione Prolog- Java.

tuProlog offre il supporto per integrare codice Prolog all'interno di applicazioni Java ed oggetti Java in programmi Prolog.

Questa integrazione è essenzialmente basata sul fatto che i motori inferenziali di tuProlog sono oggetti Java, non facendo riferimento ad alcun codice oggetto ne alle funzionalità della Java Native Interface offre una soluzione portabile.

Tramite l'integrazione offerta le applicazioni Java possono sfruttare il potere espressivo del paradigma logico per realizzare le parti "intelligenti" in modo più naturale rispetto che con la sola programmazione orientata agli oggetti.

Includere Prolog in Java.

Per abilitare l'accesso a tuProlog da Java vengono fornite un insieme di classi, in modo che ve né sia una per tutte le principali entità Prolog, ad esempio i tipi di dato standard Prolog vengono associati ad oggetti di classe **Term**, le variabili in oggetti di classe **Var**, i termini composti (funtori con argomenti) in oggetti di classe **Compound**, le stringhe vengono associate a **Compound** senza argomenti.

[2]Tale modello di comunicazione era offerto in Linda, una famosa estensione per la comunicazione fornita in alcune implementazioni di Prolog.

```
Theory theory1 = new Theory(new FileInputStream("test.pl"));
String text = "append([],X,X). \n \n" +
  "append([X|L1],L2,[X|L3]):-append(L1,L2,L3). \n \n";
Theory theory2 = new Theory(new ByteArrayInputStream(text));
```

Figura 5.9: Istanziamo due teorie in tuProlog.

```
public interface Prolog {
void setTheory(Theory th)
          throws InvalidTheoryException;
Theory getTheory();
void loadLibrary(String libname)
          throws InvalidLibraryException;
void unloadLibrary(String libname)
                   throws UnknownLibrary;
SolveInfo solve(Term t);
SolveInfo solve(String term)
           throws MalformedGoalException;
SolveInfo solveNext()
          throws NoMoreSolutionException;
}
```

Figura 5.10: Interfaccia per accedere agli oggetti rappresentanti il motore inferenziale Prolog.

È possibile rappresentare differenti teorie come oggetti di classe **Theory**, questi oggetti possono essere costruiti a partire da un file di testo contenente la definizione dei predicati che sono nella teoria. Nell'esempio illustrato in figura 5.9 definiamo con alcune istruzioni Java due teorie: "theory1" a partire da un file di testo "test.pl" il quale conterrà la definizione di alcuni predicati non specificati nell'esempio, e "theory2" a partire dalla stringa "text" contenente la definizione del predicato di append. Le teorie che definiamo andranno associate ad un motore Prolog attraverso l'interfaccia di figura 5.10, che fornisce le operazioni necessarie a registrare una teoria ad un motore, caricare una nuove libreria, risolvere un termine (goal) nel motore ed accedere alle soluzioni successive alla prima.

Le soluzioni calcolate dal motore logico vengono restituite come oggetti che implementano l'interfaccia **SolveInfo** illustrata in figura 5.11. Una discussione più dettagliata delle classi e dei metodi disponibili si possono trovare consultando la documentazione delle API tuProlog [58].

```
public interface SolveInfo {
boolean success();
boolean hasOpenAlternatives();
Substitution getSubstitution()
              throws NoSolutionException;
Term getSolution()
              throws NoSolutionException;
boolean halt();
int getHaltCode()
              throws NoHaltException;
}
```

Figura 5.11: Interfaccia **SolveInfo** per accedere alle soluzioni.

Includere oggetti Java nel codice Prolog.

La possibilità di includere componenti Java all'interno del codice Prolog viene fornita dalla JavaLibrary, attraverso di essa è possibile includere in un programma Prolog oggetti Java già istanziati, istanziarne di nuovi ed accedere ai servizi da essi forniti. Vediamo i principali predicati definiti nella libreria e la rispettiva arietà:

java_object/3 crea un nuovo oggetto Java istanziandolo da una classe specificata, è possibile indicare gli argomenti del costruttore. Il riferimento al nuovo oggetto viene associato ad un termine ground attraverso il quale si potrà interagire con l'oggetto.

<-/2 invoca un metodo indicato a destra del simbolo di predicato sull'oggetto Java rappresentato dall'identificatore a sinistra del simbolo di predicato. Con esso è possibile accedere anche a attributi pubblici dell'oggetto.

returns/2 unifica il valore ritornato dall'invocazione di un metodo Java con tipo di ritorno differente da "void" che si trova sinistra del simbolo di predicato con il termine Prolog posizionato a destra dello stesso.

In figura 5.12 vi è un semplicissimo esempio di un programma Prolog che istanzia un oggetto dalla classe "Counter" associandone il riferimento ad un termine "myCounter", quindi viene richiamato il metodo "set" e poi un metodo "get" il cui parametro di ritorno viene unificato con "X", alla fine "X" viene stampato a video.

Nell'esempio precedente l'oggetto Java al quale accediamo è stato istanziato all'interno del programma Prolog, invece, in alcuni casi può essere utile accedere ad oggetti Java che sono stati precedentemente istanziati. Ciò è possibile

```
java_object('tuprolog.demo.Counter',[],myCounter),
myCounter <- set(5),
myCounter <- get returns X,
write(X).
```

Figura 5.12: Esempio di programma Prolog che istanzia ed utilizza un oggetto Java.

grazie alla funzione statica **register**. Tale funzione è richiamabile dal lato Java attraverso la seguente sintassi:

```
JavaLibrary.register(Struct name, Object obj)
```

L'effetto di tale funzione è registrare un oggetto "obj" all'interno del motore Prolog e renderlo accessibile attraverso il riferimento "name". Nell'utilizzo della **JavaLibrary** si faccia attenzione in quanto non è possibile registrare lo stesso oggetto due volte anche se con nomi differenti. Gli oggetti out, in, error appartenenti alla libreria statica java.lang.System vengono registrati automaticamente rispettivamente con nomi stdout, stdin, stderr.

Estensibilità attraverso librerie.

Una caratteristica importante di tuProlog è la capacità di gestire estensioni attraverso l'utilizzo delle librerie. Le librerie devono essere tutte derivate dalla classe definita nel framework tuProlog **Library**, questo per assicurare che la libreria abbia le caratteristiche che le assicurino l'interazione con il core tuProlog. Tra le librerie che sono attualmente fornite troviamo

MetaLibrary: fornisce le primitive per consultare le teorie, caricare e scaricare le libreria, generare nuovi processi Prolog.

JavaLibrary: fornisce un'interfaccia per accedere, creare, e gestire gli oggetti Java da codice Prolog.

IsoLibrary: fornisce i predicati Prolog standard ISO.

Vi sono altra librerie che permettono di accedere a spazi di tuple, e infrastrutture Internet. Le tre librerie che abbiamo citato vengono caricate automaticamente ogni volta che si istanzia un motore tuProlog.

tuProlog rispetto le altre implementazioni.

Ciò che ci ha spinto a scegliere tuProlog come motore logico per gli agenti sono essenzialmente due fattori:

- l'integrazione bidirezionale e portabile con Java;
- un motore inferenziale scritto in Java, minimale ed estensibile attraverso le librerie;

tuProlog è liberamente scaricabile all'indirizzo [57] in forma di classi Java compilate, a detta degli autori presto sarà reso pubblico il codice sorgente.

Nel manuale di utilizzo [2] sono presentate alcune prove di prestazione, dove sono confrontati tuProlog ed altre implementazioni tra cui Jinni e jprolog. Per quanto lo studio sia piuttosto limitato, le performance ottenute da tuProlog sembrano comparabili con le altre implementazioni, anzi, in alcuni casi tuProlog offre delle prestazioni addirittura migliori.

Effettuate alcune prove per prendere confidenza con il tool abbiamo avuto modo di renderci conto che il progetto è ancora in fase di sviluppo: la stessa documentazione in rete è work in progress. Nonostante ciò, ci sembra che tuProlog sia il tool che offre un buon supporto ai nostri obiettivi ed abbiamo deciso d'impiegarlo per realizzare l'integrazione JADE - Prolog.

5.3 Utilizzo degli agenti logici nel modello di sviluppo D-CaseLP

Prima di passare all'analisi dell'estensione che ci permetterà di far eseguire alla platform JADE agenti logici, vorremmo fornire alcune idee sul ruolo che avranno gli agenti logici nell'attività di sviluppo del MAS. Abbiamo ipotizzato che lo sviluppo avvenga seguendo un ciclo di vita a spirale, ciò implica la realizzazione di un certo numero di prototipi allo scopo di raffinare i requisiti e la progettazione del sistema sotto sviluppo. Supponiamo di suddividere i requisiti in diversi insiemi a seconda di quanto questi requisiti siano in relazione e per ogni insieme di sviluppare un prototipo, definiamo alcune delle relazioni che possono intercorrere tra i prototipi che verranno sviluppati.

Definizione 5.1 *Presi due prototipi $\mathcal{A}$ e $\mathcal{B}$ appartenenti al processo di sviluppo di un sistema, diciamo*

$\mathcal{B}$ **è raffinamento di** $\mathcal{A}$ *se $\mathcal{B}$ è un prototipo per lo stesso sottoinsieme di requisiti del sistema di cui si occupa $\mathcal{A}$, ed alcune delle soluzione implementate in $\mathcal{B}$ sono ad un livello di dettaglio superiore rispetto alle soluzioni implementate in $\mathcal{A}$;*

$\mathcal{B}$ **ha intersezione con** $\mathcal{A}$ *se i due hanno almeno un requisito in comune;*

$\mathcal{B}$ **è indipendente da** $\mathcal{A}$ *se l'intersezione degli insiemi dei requisiti di* $\mathcal{A}$ *e* $\mathcal{B}$ *è l'insieme vuoto;*

Ci sembra ragionevole fare i seguenti assunti.

Assunto 5.1 *Ogni singolo prototipo si occuperà di un sottoinsieme dei requisiti del MAS, eccetto i prototipi ottenuti verso la fine del ciclo a spirale, i quali visto che si vorrebbe arrivare ad una implementazione o almeno ad un unico prototipo del sistema è plausibile che implementino tutti i requisiti.*

Assunto 5.2 *Ogni prototipo realizzerà un sottoinsieme degli agenti del MAS finale.*

Assunto 5.3 *Ogni agente in un MAS unito ad altri implementa le funzionalità che servono per ottenere uno o più dei requisiti di sistema.*

Assunto 5.4 *Ogni requisito sarà implementato da un insieme di agenti in forte coesione.*

Facciamo osservare che l'assunto 5.1 sottointende che il sistema venga sviluppato tramite un insieme di prototipi e durante il processo di raffinamento sia possibile fare confluire due prototipi in uno solo in modo che alla fine si abbia un unico prototipo o realizzazione del sistema.

Con gli assunti 5.3 e 5.4 mettiamo in relazione i prototipi e gli agenti. In figura 5.14 abbiamo illustrato alcuni prototipi che hanno una intersezione evidenziando gli agenti in essi contenuti.

La figura 5.14 dovrebbe aiutarci a comprendere che se si vuole costruire un sistema, non tutti gli agenti dell'architettura MAS saranno sotto sviluppo contemporaneamente. Ve ne saranno alcuni le cui mansioni risulteranno in qualche modo marginali rispetto al sottoinsieme di requisiti che si sta analizzando nel prototipo. Durante l'esecuzione del prototipo è facile che alcuni degli agenti sotto sviluppo necessitino di riferirsi o interagire con alcuni fra quelli al di fuori del prototipo. Ad esempio, in figura 5.14, è schematizzato un MAS con tre agenti (cerchi) da cui partono delle frecce che individuano le interazioni fra gli agenti. Si è suddiviso il MAS (rettangolo più grande), in due prototipi (rettangoli piccoli che si intersecano). Gli agenti che stanno entro il confine del prototipo sono gli agenti sotto sviluppo per quel prototipo, quindi gli agenti che sono oggetto della prototipazione. Quando si simulerà il prototipo A, è probabile che l'agente A, per completare un suo compito abbia bisogno di interagire con l'agente C che non era sotto sviluppo. Un'idea potrebbe essere utilizzare la programmazione logica per scrivere velocemente un agente che a determinate richieste dell'agente A risponda come potrebbe rispondere l'agente C. Potremmo così individuare alcune tipologie di agenti:

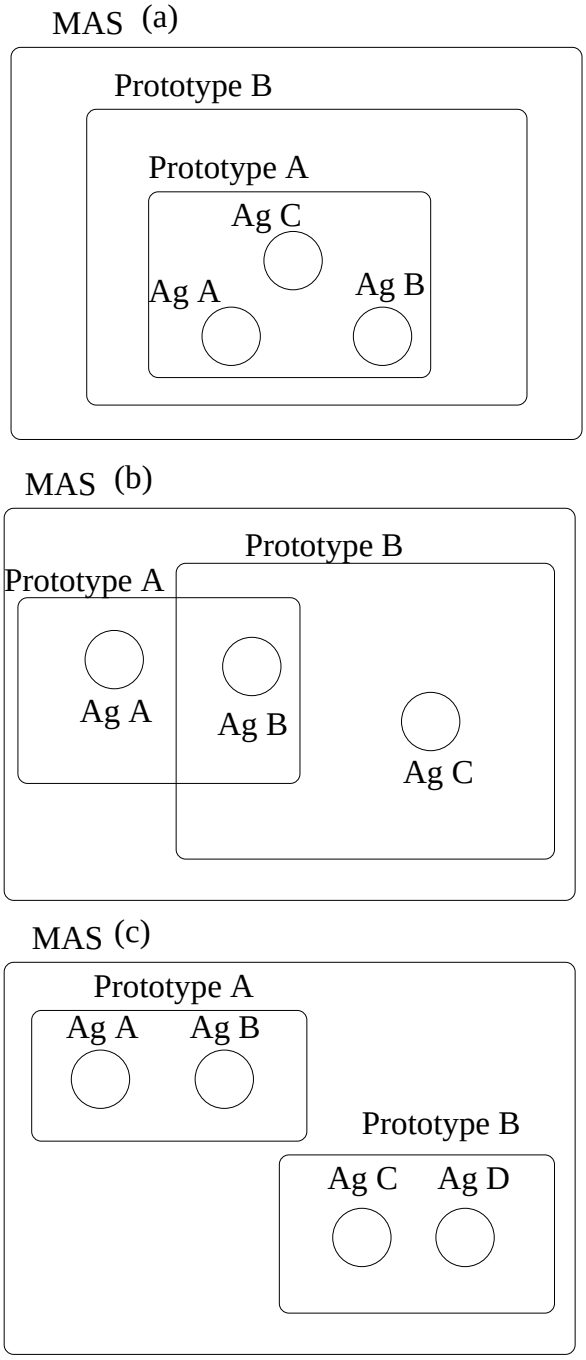

Figura 5.13: Schematizzazione delle relazioni tra prototipi: (a) raffinamento, (b) intersezione, (c) indipendenza.

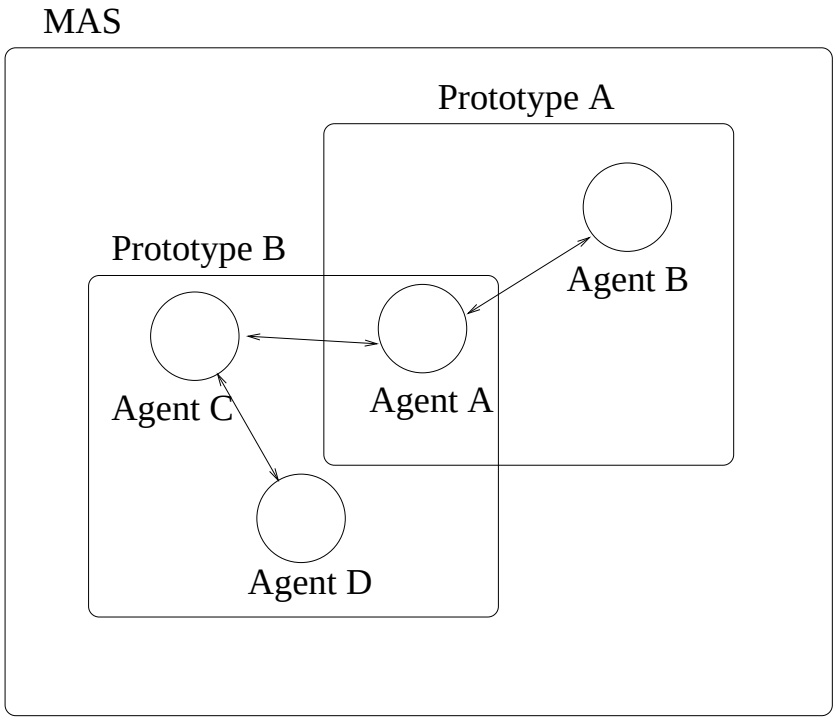

Figura 5.14: Esempio di come potrebbero essere coinvolti gli agenti in diversi prototipi.

Agenti dummy: agente che serve solo in fase di sviluppo per far funzionare il prototipo.

Agenti sotto sviluppo: un Ag-Prolog o un Ag-Java che è in fase di sviluppo e alla fine del ciclo di sviluppo farà parte del MAS.

Si osservi come ci si limiti ad associare un agente dummy all'agente C e non al D, questo perché essendo C un agente dummy non sfrutterà nessuna interazione con l'agente D per fornire le risposte a A. Gli agenti dummy potrebbero essere dei semplici agenti logici specificati ad esempio tramite AgentRules per i quali si fornisce un comportamento reattivo. L'esempio precedente ci suggerisce due modi di utilizzare gli agenti logici:

- come agenti provvisori che aiutano a sviluppare rapidamente prototipi;
- come agenti a tutti gli effetti che verranno mantenuti nella realizzazione finali.

5.4 Analisi

Illustriamo finalmente la progettazione dell'estensione che ci permetterà di eseguire agenti logici in JADE.

5.4.1 Obiettivi dell'integrazione

Considerato che allo stato attuale la trattazione delle problematiche relative alla mobilità degli agenti è al di fuori degli obiettivi di D-CaseLP, le funzionalità che l'integrazione dovrebbe rendere accessibili sono:

Esecuzione di agenti Ag-Prolog. Si vuole fornire la possibilità di eseguire indistintamente agenti Ag-Prolog o Ag-Java che grazie alla condivisione di uno stesso modello di comunicazione basato sullo scambio di messaggi FIPA ACL siano in grado di interagire tra loro alla pari.

Controllo e monitoring su agenti Ag-Prolog. Ci aspettiamo che l'integrazione offra agenti logici che siano a tutti gli effetti agenti della platform, sui quali sia quindi possibile utilizzare gli stessi strumenti di monitoring e controllo (RMA, Sniffer Agent, Introspector Agent, etc) che si applicano per il monitoring e il controllo di agenti JADE anche agli agenti Ag-Prolog.

Al fine di poter utilizzare gli strumenti di JADE sui nuovi agenti Ag-Prolog, bisogna che gli agenti Ag-Prolog siano nascosti in oggetti della classe Agent di JADE.

Una prima questione è decidere a che livello introdurre le componenti logiche negli agenti JADE. Gli agenti sono istanze della classe Agent, il cui comportamento è suddiviso in diversi oggetti istanze della classe Behaviour, che vengono schedulati secondo una qualche politica. È chiaro che per effettuare l'integrazione sarà necessario operare delle scelte circa le seguenti questioni:

- Introduciamo una classe agente logico o un comportamento logico o entrambi?
- Nel caso in cui ci avvalessimo di Behaviour, quale tipo di Behaviour è più adatto ai nostri scopi?

Risponderemo alle precedenti problematiche quando illustreremo il design.

5.4.2 $_N$ProlAg: linguaggio d'implementazione

Fino ad ora abbiamo parlato di agenti logici specificando che saranno scritti in Prolog[3], in realtà tale linguaggio non nasce per scrivere agenti intelligenti e non offre quindi costrutti nativi utili per essi. Il Prolog di per sé non dispone di primitive per la comunicazione, infatti, molti dei predicati che il linguaggio offre per la programmazione concorrente, i thread, e l'esecuzione distribuita sono estensioni successive, spesso fornite attraverso soluzioni proprietarie differenti da una implementazione all'altra del linguaggio. È evidente che gli agenti Ag-Prolog non possono essere semplicemente agenti il cui linguaggio di programmazione è Prolog, ma abbiamo bisogno di estendere il linguaggio ed aggiungere alcuni predicati che permettano di utilizzare il paradigma logico per la programmazione degli agenti.

Il problema circa quale estensione di Prolog sia la più adatta al fine di utilizzarlo come linguaggio per programmare agenti è stato ampiamente discusso e trattato, tant'è vero che ci sono varie proposte. Noi partiremo da *Prolog for Agents* (ProlAg) [42] che è la soluzione proposta in CaseLP.

La prima idea che verrebbe in mente, è scegliere ProlAg come linguaggio di programmazione per gli agenti Ag-Prolog. Considerato che gli agenti CaseLP erano implementati in ProlAg questa scelta darebbe una certa continuità, permettendo di recuperare il lavoro già svolto sui linguaggi di descrizione del MAS. È anche vero che l'impostazione metodologica di D-CaseLP è differente da quella di CaseLP, in particolare si sta lavorando per offrire un supporto più completo dei ruoli per gli agenti ed un'integrazione del framework concettuale FIPA, per questo parte della metodologia e del lavoro precedente dovrà essere aggiornato.

[3] Più precisamente si pensa di utilizzare l'implementazione di Prolog fornita in tuProlog

Definiamo una versione modificata di ProlAg che chiameremo per distinguerla dalla precedente $_N$ProlAg. Una delle prime modifiche rispetto ad ProlAg così come era in CaseLP cosiste nel permettere alle primitive di comunicazione di trattare messaggi espressi in un qualche linguaggio di comunicazione per agenti (FIPA ACL, KQML). Questo implica ad esempio che il destinatario di un messaggio spedito attraverso una send non sia più espresso come termine del predicato ma si trovi all'interno del messaggio stesso. In generale i parametri delle primitive saranno codifiche o schemi di messaggi FIPA ACL. In $_N$ProlAg abbiamo eliminato la primitiva *async_receive_all()* in quanto può essere ottenuta a partire dalla async_receive().

Non si forniscono le versioni "safe" dell'assert e della retract che possono essere ottenute come spiegato in [15], mentre per ciò riguarda i predicati di comunicazione abbiamo cambiato sostanzialmente solo la sintassi.

Predicati per la comunicazione.

I predicati di comunicazioni offerti da $_N$ProlAg sono l'implementazione delle primitive di comunicazione che appartengono all'interfaccia astratta presentata in 1.3.5 con alcune modifiche dovute al fatto che l'interfaccia è realizzata in un paradigma logico.

Si tenga presente che il paradigma logico ed in particolare Prolog basa le sue "computazioni" sul principio di risoluzione di Robinson e l'unificazione. Il primo permette di combinare delle clausole in modo da ottenere altre clausole conseguenze logiche delle precedenti, mentre il secondo permette di capire se due termini sono "compatibili" fornendo una relazione fra termini.

L'unificazione dei termini potrebbe essere anche vista come una forma bilaterale di pattern matching, in cui invece che confrontare uno schema con qualcosa di completamente istanziato, si confronta uno schema con uno schema individuando un insieme di elementi che appartengono ad entrambi.

Dato che i presupposti della programmazione logica sono fortemente differenti dal paradigma ad oggetti o imperativo, bisogna fare attenzione al modo in cui codifichiamo i messaggi FIPA ACL in Prolog. La cosa migliore sarebbe sfruttare le peculiarità del paradigma logico fornendo una codifica dei messaggi FIPA ACL più maneggevole rispetto a quella basata sulle stringhe che si era pensata per gli agenti Ag-Java.

Chiameremo la nuova codifica dei messaggi **PrologACLCode** , in essa dovremo definire un insieme di termini per esprimere schemi e messaggi FIPA ACL.

In prima approssimazione dovremo aggiungere un simbolo di funzione unaria per ogni chiave che può essere specificata in un messaggio FIPA ACL, cosicché le coppia chiave-valore del tipo

:chiave valore

sia esprimibile nella codifica come

chiave(valore)

Un messaggio FIPA ACL sarà una tupla contenente un certo numero di coppie chiave valore, nella codifica un messaggio sarà rappresentato come una lista di termini simili a quelli descritti precedentemente.

Descriviamo nel seguito le primitive incluse in $_N$ProlAg.

sync_receive(Message) nel caso in cui la mailbox dell'agente contenga un messaggio che unifica con lo schema di messaggio indicato come argomento, attraverso l'unificazione vengono riempiti i campi liberi dello schema. Se non vi è alcun messaggio che unifica con lo schema l'agente che ha effettuato la chiamata viene bloccato fino a quando non arriva un messaggio che unifica con lo schema.

async_receive(Msg_input,Msg_output) controlla nella mailbox dell'agente che richiama questo predicato se c'è un messaggio che unifica con *Msg_input*, se un cosiffatto messaggio è presente nella mailbox, viene unificato con *Msg_output* e tolto dalla mailbox, altrimenti *Msg_output* viene unificato con il termine **no_message**.

send(Message) spedisce il *Message* al destinatario o ai destinatari indicati con gli appositi termini nel messaggio.

In analogia con ciò che era ProlAg in CaseLP, $_N$ProlAg si pone come linguaggio in cui tradurre i linguaggi di specifica omogeneamente. Si consideri come esempio la specifica AgentRules di un agente, ciò di cui dovremmo aver cura per esegure tale specifica come agente logico in D-CaseLP, è fornire un meta-interprete scritto in $_N$ProlAg che fornisca l'engine allo stato ed il program tradotti dalla specifica.

L'astrazione delle classi di agenti verrà fornito dai linguaggi di specifica presenti in CaseLP, che si prevede di rendere disponibili anche nel framework D-CaseLP[4].

Allo stato delle cose gli agenti logici che ci proponiamo di fornire in D-CaseLP potrebbero essere caratterizzati dal punto di vista concreto dalla seguente equazione:

JADE + $_N$ProlAg = Ag-Prolog

Dove $_N$ProlAg a sua volta potrebbe essere visto come

Prolog + Predicati per la comunicazione = $_N$ProlAg

[4]Ciò che prevediamo di fornire, è un'implementazione dei predicati di ProlAg in $_N$ProlAg, in modo che sia possibile continuare a utilizzare linguaggi e traduttori offerti in CaseLP.

5.4.3 Use Case Diagram per l'estensione di JADE

Analizziamo in questa sezione quali sono i requisiti che deve avere l'estensione alle librerie JADE, affinché sia possibile scrivere agenti logici. Nello use case diagram in figura 5.15 sono illustrati gli use case che descrivono l'iterazione degli utenti con il package che forniremo. Abbiamo individuato due attori coinvolti dello use case:

Logic Agent Developer: che si occupa di progettare e programmare gli agenti logici;

MAS Configurator: che si occupa di configurarli ed esegurli.

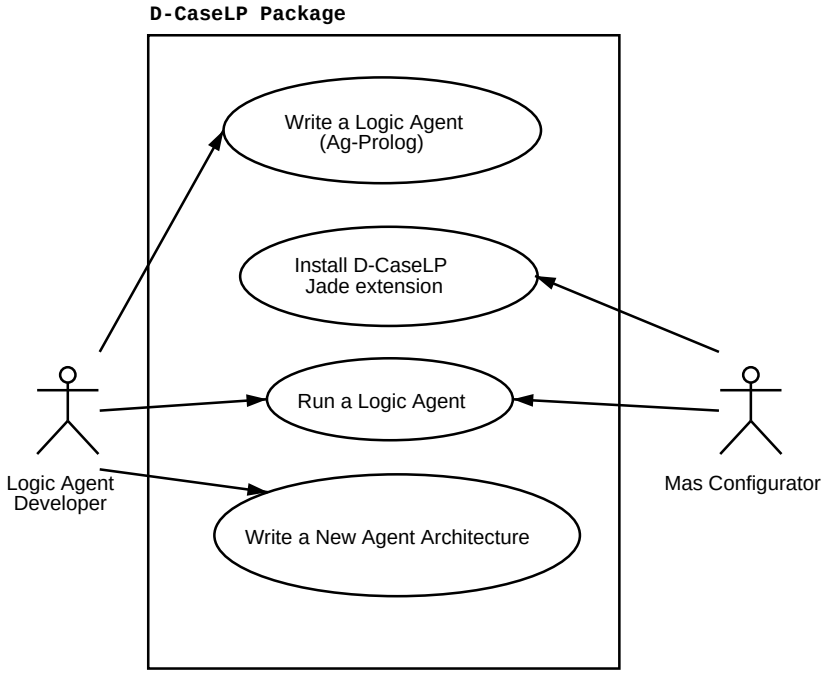

Figura 5.15: Use case per D-CaseLP forniamo.

Descriviamo gli use case in modo da dare un'idea di cosa rappresentano e quali funzionalità presuppongano.

Write a Logic Agent: il "Logic Agent Developer" specifica un agente logico, sceglie un'architettura per agenti implementata in $_N$ProlAg e specifica in un linguaggio dipendente dall'architettura scelta lo stato iniziale e il Behaviour dell'agente.

Write a New Agent Architecture: il "Logic Agent Developer" non trova fra le architetture per agenti logici implementate quella che fa al caso suo, decide quindi di scriverne una nuova.

Run a logic agent: una volta che gli agenti sono stati scritti il "Logic Agent Developer" passa al "Mas Configurator" le componenti che sono necessarie all'esecuzione degli agenti logici, in modo che quest'ultimo possa configurare l'esecuzione ed eseguire il MAS.

Install D-CaseLP JADE extension: in questo use case viene coinvolto solo il "Mas Configurator" che ha il compito di installare e configurare l'estensioni che permette di utilizzare gli agenti logici in JADE prima di eseguire gli agenti forniti dal "Logic Agent Developer". Questa fase sarà eseguita una sola volta durante installazione di JADE dal momento che l'estensione consisterà in un certo insieme di package Java da aggiungere a quelli JADE questo use case consisterà nel seguire delle istruzioni di installazione e configurazione fornite da D-CaseLP.

Concludiamo quindi la fase di analisi fornendo un maggior livello di dettaglio nello spiegare gli use-case **Write a Logic Agent** e **Write a New Agent Architecture**.

5.4.3.1 Write a Logic Agent

Seguendo la concettualizzazione di agente in termini di *State-Engine-Program* descritta in sezione 1.2.1, raffiniamo lo use case Write a Logic Agent nello use case diagram in figura 5.16. In esso si possono individuare due nuovi use case necessari al completamento dello use case precedentemente descritto:

Write a Logic Agent: è lo use case principale in cui l'utente (attore Logic Agent Developer) definerà un agente Ag-Prolog utilizzando l'estensione alla libreria JADE fornita in D-CaseLP.

Choose an Agent Engine: l'utente sceglierà quale engine è più adatto per l'agente che sta progettando.

Write State and Behaviours: in dipendenza del tipo di engine che ha scelto, il Logic Agent Developer scriverà lo stato iniziale e il comportamento dell'agente.

5.4.3.2 Write a New Agent Architecture

Vediamo cosa succede quando il Logic Agent Developer non trova un'architettura per agente già sviluppata che soddisfa i requisiti da lui individuati per esprimere i

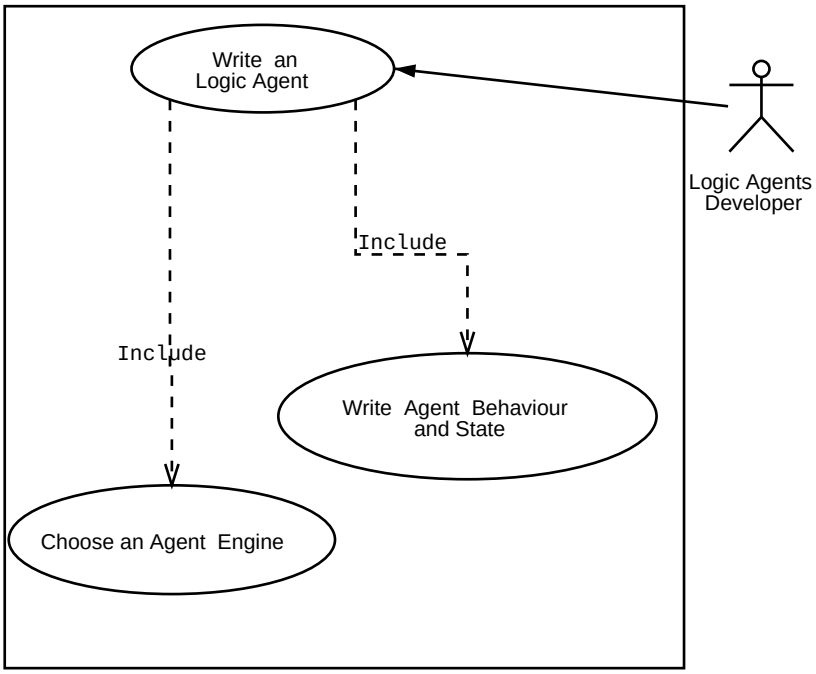

Figura 5.16: Use case per scrivere agenti logici.

suoi agenti[5]. Come illustrato nel diagramma 5.17, lo use case in questione include altre tre attività.

Write a New Agent Architecture: l'utente decide di progettare una nuova architettura per agente.

Decide a Behavior and State Format: l'utente progetta un linguaggio in cui esprimere il comportamento e lo stato per gli agenti. Qualora non sia necessaria l'introduzione di un linguaggio più astratto di quello che è $_N$ProlAg, si possono esprimere stato e program in forma di predicati, altrimenti si definirà una nuova sintassi attraverso una BNF e la relativa semantica.

Write a Behavior & State Translator to $_N$ProlAg: si indica una procedura per la traduzione degli eventuali linguaggi definiti per esprimere lo stato e il program che l'utente ha progettato. Si fornisce un traduttore che data la specifica del linguaggio per lo stato ed il program restituisca dei predicati $_N$ProlAg che saranno utilizzati dall'engine durante l'esecuzione[6].

[5]Si ricorda una delle ipotesi progettuali di D-CaseLP consiste nel non vincolare l'utente ad utilizzare una prescelta architettura per agenti, anzi si vorrebbe lasciare la possibilità di impiegarne differenti in uno stesso MAS.

[6]Si può anche decidere di includere il traduttore nel engine, in tal caso non è necessario dare un traduttore.

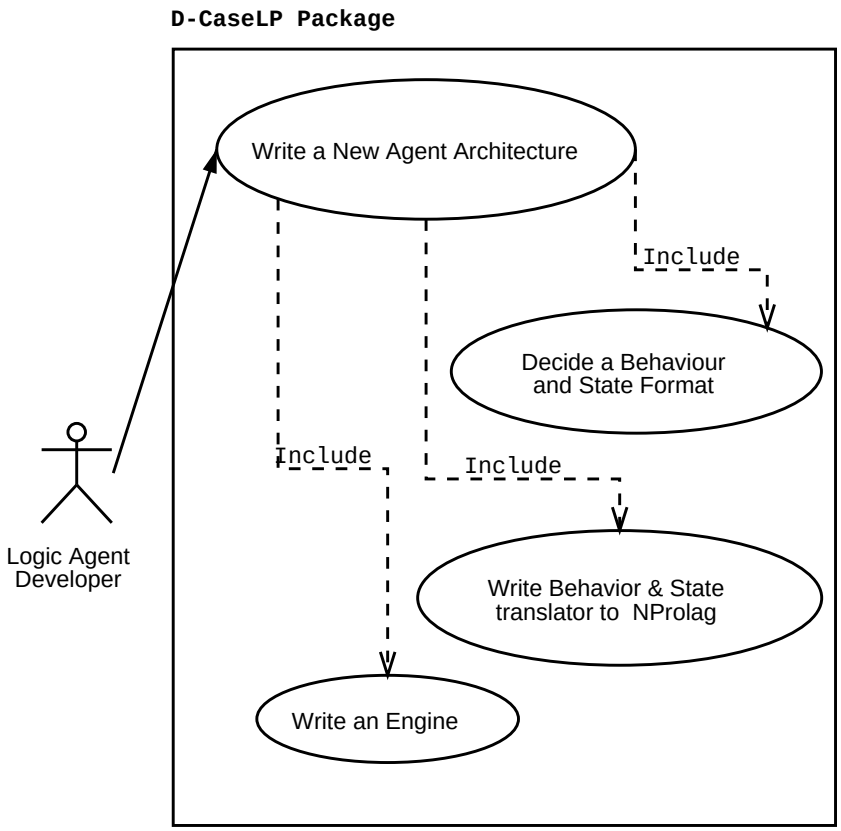

Figura 5.17: Dettaglio sullo use case WriteArchitecture.

Write an Engine: si progetta ed implementa in $_N$ProlAg un engine, che presa la codifica di stato e comportamento dell'agente si comporta coerentemente con il modello di architettura pensata dall'utente.

Da questo punto in poi daremo per scontato che qualora l'utente si definisca una nuova architettura metta a disposizione la documentazione e gli strumenti necessari alla realizzazione dello use case "Write a New Agent Architecture";

Visto che è altamente probabile che l'utente dopo aver definito un'architettura la utilizzi, avremmo potuto includere nello use case "Write a New Agent Architecture" lo use case "Write a Logic Agent". Tuttavia ci sembra inutile inserire un ulteriore vincolo: se qualcuno dovesse definire architetture per agenti logici ad uso di altri deve essere libero di poterlo fare.

Visto che se si sta scrivendo un agente logico (use case Write a Logic Agent) si deve avere a disposizione un'architettura per agente, ovvero un engine espresso in $_N$ProlAg e degli strumenti per tradurre lo stato ed il program in predicati $_N$ProlAg. Assumeremo nel resto di questa sezione che il program e lo stato siano sempre stati tradotti dall'utente in $_N$ProlAg e siano stati memorizzati in dei

file. Nella prossima sezione illustreremo il design dell'estensione a JADE, per il momento ci concentreremo soprattutto sugli use case che coinvolgono le fasi di sviluppo degli agenti logici e non illustreremo in dettaglio quelle di esecuzione e configurazione del tool.

5.5 Design dell'estensione

In questa sezione presentiamo la descrizione di come pensiamo di ottere l'integrazione gli agenti logici Ag-Prolog a partire dagli agenti JADE e tuProlog. È chiaro che per mettere in grado gli utenti di scrivere un'architettura per agenti logici che sia eseguibile dalla nostra estensione dobbiamo specificare alcune questioni:

- quale predicato individua l'engine;
- ipotesi che deve soddisfare un engine affinché non si perda la concorrenza tra agenti.

Per la prima delle due questioni, la cosa più semplice è prevedere che tutti gli engine delle architetture siano specificati da uno stesso predicato. Usando tutta la fantasia di cui disponiamo imporremo che l'engine sia indicato dal predicato zero-ario "engine". Per le problematiche relative alla concorrenza, il discorso è un poco più complicato, si tratta a grandi linee di individuare come frammentare l'esecuzione dell'engine in differenti Behaviour di JADE.

5.5.1 La concorrenza degli agenti in JADE

Come abbiamo illustrato in sezione 4.2.2.1, in JADE gli agenti corrispondono a thread eseguiti su una Java Virtual Machine. Il multithreading offerto da Java è del tipo *pre-emptive multitasking* senza *time-sharing*, in altre parole lo scheduling dei thread è interrompibile ma non essendoci un tempo limite di occupazione della risorsa (*time-slice*) non è prevista alcuna interruzione se un thread occupa il processore troppo a lungo.

In questo modello di concorrenza è responsabilità del programmatore non scrivere thread che monopolizzino la risorsa processore. Si pensi ad un agente logico caratterizzato, ad esempio, dalla architettura proposta in [39] la quale prevede un engine che consta di una serie di attività svolte una dopo l'altra in un ciclo come quello che segue

- Inizio ciclo al tempo T;
    1. Osservare gli input al tempo T;

2. Pensare;
3. Scegliere una o più azioni da eseguire;
4. Eseguire le azioni;

- Fine ciclo: T=T+1.

Innanzi tutto tale architettura contiene un ciclo e non è pensabile implementare l'agente logico fornendo un engine che consista in un ciclo potenzialmente infinito di attività. Una scelta di questo tipo comporterebbe che l'agente non rilasci mai la CPU con la conseguente perdita della concorrenza tra gli agenti del MAS. A questo problema sopperiamo non includendo il ciclo all'interno dell'engine, cosicché l'agente rilasci il processore dopo che ha eseguito una volta tutte le attività precedenti. Nonostante il fatto che si tolga il ciclo dall'engine l'esecuzione delle attività rimane ciclica per effetto delle chiamate successive al predicato che rappresenterà l'engine effettuate dallo schedulatore.

Questo tipo di soluzione per la concorrenza fra gli engine di agenti concorrenti risulta efficace qualora sia rispettata la seguente ipotesi:

Ipotesi 5.1 *Le attività contenute in un singolo ciclo dell'engine terminano e sono dal punto di vista computazionale "sufficientemente leggere".*

Quanto debbano essere leggere le computazioni dipende in sostanza dal tempo di reazione che si vuole dagli agenti, maggiore è la reattività pretesa minore deve essere il tempo necessario ad eseguire un "ciclo" di istruzioni.

L'ipotesi 5.1 è in generale ragionevole, e non crea problemi fino a quando si vogliono utilizzare particolari tipi di agenti. Ad esempio soddisfano tale ipotesi gli agenti *dummy* che abbiamo individuato come utili durante lo sviluppo dei prototipi del MAS. Questi agenti, infatti, saranno principalmente agenti reattivi che reagiscono agli stimoli/richieste di altri agenti con risposte "preconfezionate".

Qualche difficoltà nel caso in cui si vogliano agenti logici da utilizzare come "ragionatori". Questo tipo di agenti richiede in genere la dimostrazione predicati piuttosto complessi e non è semplice capire dove possa essere interrotta la dimostrazione senza avere informazioni sulle dimostrazioni stesse.

Per il momento ci concentreremo sugli agenti che soddisfano l'ipotesi illustrando come integrarli in JADE attraverso tuProlog.

5.5.2 Design di agenti logici con tuProlog e JADE

Spieghiamo a partire da tuProlog e JADE come abbiamo intenzione di realizzare gli agenti logici illustrando

- come viene incluso il codice $_N$ProlAg negli agenti JADE;

- come estendiamo tuProlog con i predicati $_N$ProlAg al fine di permettere di comunicare con la platform JADE.

Per rispondere a queste domande ci serviremo del class diagram in figura 5.18, del quale descriveremo prima le classi e poi gli oggetti che vi compaiono.

Le Classi.

Prima di descrivere le singole classi, vorremmo chiarire che per ognuna di esse abbiamo evidenziato solo i metodi e gli attributi che in qualche maniera vengono coinvolti nella definizione ed esecuzione degli agenti logici. L'obiettivo di questa descrizione è dare un'idea precisa di come si possa organizzare l'integrazione dei vari strumenti adottati rimanendo ad un livello progettuale. Descriviamo le classi che compaiono nel class diagram ricordando velocemente le funzionalità associate ai metodi.

Agent: è la classe che tutti gli agenti JADE devono estendere. Come abbiamo già avuto modo di illustrare nella sezione dedicata a tale tool, in essa sono definiti i metodi che permettono d'interfacciare gli agenti alla platform.

Si ricorda che ogni agente JADE viene eseguito dalla platform attraverso il metodo *run*, tale metodo non può essere sovrascritto dalle sottoclassi.

Il metodo *setup* permette di preparare l'agente all'esecuzione, nella sua definizione è possibile aggiungere i comportamenti (behaviour) iniziali all'agente attraverso il metodo addBehaviour.

Con i metodi *send*, *receive*, *blockingReceive* si accede ai servizi di comunicazione degli agenti.

Infine, il metodo *getArguments* permette di recuperare i parametri dell'agente che sono stati specificati da linea di comando o nell'interfaccia di start-up.

Agent1: è una generica classe di agenti definita dall'utente, l'abbiamo inclusa nel class diagram al solo scopo di ricordare che tutti gli agenti JADE sono istanze di classi che estendono la classe Agent.

AgProlog: è la classe principale della nostra estensione, dalla quale istanzieremo agenti logici sulla platform JADE. In tale classe si ridefinisce il metodo *setup* in modo da effettuare le seguenti operazioni:

- leggere i parametri associati all'agente (tramite getParameters) e venire a conoscenza del nome dei file in cui sono specificati il program, engine e stato dell'agente;

- creare un motore inferenziale Prolog a partire da una istanza dell'interprete tuProlog, a cui si aggiungono le teorie contenenti i predicati che rappresentano l'estensione per la comunicazione, ed i predicati letti da file che definiscono stato, program, ed engine dell'agente;
- importare attraverso le funzionalità offerte da tuProlog il riferimento (This) all'oggetto di cui stiamo definendo il metodo setup, in modo da potersi riferire ad esso tramite il temine "me" all'interno del codice $_N$ProlAg;
- aggiungere all'agente attraverso il metodo *addbehaviour* un'istanza della classe ResolveEngine, passandogli il motore inferenziale precedentemente creato.

CyclicBehaviour: è la classe definita nel framework JADE che definisce un comportamento semplice che viene ripetuto ad ogni attivazione dell'agente. Le istruzioni che caratterizzano il comportamento devono essere contenute nel metodo *action*.

ResolveEngine: è classe appartenente all'estensione D-CaseLP che specializza il comportamento del generico Behaviour ciclico in un behaviour nel quale viene risolto il goal "?-engine". Al costruttore di questa viene passato il riferimento al motore logico (parametro prolog_engine) che viene custodito in un attributo della classe in modo da poter invocare all'interno del metodo *action* il metodo *solve* del goal.

Prolog: è una classe che appartiene al framework tuProlog, ogni sua istanza è un motore inferenziale differente in cui sono state caricate automaticamente le librerie ISOLibrary, MetaLibrary e JavaLibrary. Il metodo *addtheory* permette di aggiungere al motore inferenziale delle nuove teorie, mentre i metodi *solve* permettono di dimostrare un certo goal espresso nella forma di un oggetto di classe Term (termine) o una stringa.

Theory: una classe appartenente al framework tuProlog che mantiene la codifica di una teoria ovvero un insieme di predicati Prolog. Il costruttore di questa classe prende come argomento una codifica in stringhe del programma Prolog. Le teorie possono essere aggiunte ad un motore logico dinamicamente.

Gli oggetti.

Descriviamo gli oggetti che compaiono nel diagramma 5.18 [7], sono tutti istanze della classe Theory e rappresentano insiemi di predicati Prolog. Gli oggetti Engine, State, Program rappresentano rispettivamente le teorie che contengono i predicati che definiscono il codice dell'engine, lo stato ed il program dell'agente logico sotto sviluppo. Invece, l'oggetto NProlag rappresenta i predicati che definiscono l'estensione $_N$ProlAg del Prolog standard. Tutte queste teorie vengono aggiunte al motore logico di ogni istanza di agente della classe AgProlog.

Le relazioni contenute nel diagramma.

Nel diagramma ci sono tre tipi di relazioni usate coerentemente con la notazione UML:

Generalization: individuata dalla freccia continua, indica che la classe dalla quale parte estende la classe in cui arriva.

Use: relazione di dipendenza con stereotipo use standard UML, indica che la semantica della classe di partenza dipende dall parte pubblica della classe in cui la freccia arriva.

IstanceOf: relazione di dipendenza con stereotipo IstanceOf standard UML, indica che gli oggetti da cui parte la freccia sono istanze della classe in cui arriva.

Rimarchiamo ulteriormente che la classe che implementa gli agenti logici semplici Ag-Prolog estende l'Agent di JADE, quindi è un agente JADE a tutti gli effetti ed avrà dei comportamenti espressi in termini di Behaviour. Per la precisione ogni agente istanza di AgProlog avrà un unico behaviour ResolveEngine, il quale ridefinisce il metodo action() in modo che il behaviour possa risolvere il goal "?-engine". Si noti che nel costruttore viene passato il riferimento al motore d'inferenza attivato nell'agente, questo affinché sia possibile invocare su tale oggetto il metodo solve.

Implementazione dell'oggetto NProlag.

Nonostante tuProlog permetta di estendere l'insieme dei predicati built-in attraverso l'utilizzo delle librerie, abbiamo scelto di definire i predicati dell'estensione $_N$ProlAg in una teoria da aggiungere al motore inferenziale Prolog.

[7] Per il lettore nuovo alla notazione UML, ricordiamo che gli oggetti vengono rappresentati nel class diagram dai rettangoli con etichetta sottolineata.

Prima di illustrare come vengono implementati i singoli predicati di comunicazione introdotti da $_N$ProlAg supponiamo di aver definito due predicati di servizio che ci permettano di trasformare la codifica PrologACLCode per esprimere i messaggi FIPA ACL o pattern di messaggi, rispettivamente in oggetti di classe ACLMessage o MessageTemplate.

PrologACLMsg(X,Y) predicato che è vero se X è una codifica PrologACLCode del messaggio ACL contenuto nell'oggetto rappresentato dal temine Y.

PrologACLTemplate(X,Y) predicato che è vero se X è la codifica PrologACLCode del pattern che esprime gli stessi vincoli dell'oggetto MessageTemplate rappresentato dal temine Y.

Nonostante sia evidente che i predicati appena definiti debbano al loro interno fare uso di funzionalità extralogiche se non altro per interfacciarsi con gli oggetti Java, agli scopi di questa presentazione ipotizziamo che siano entrambi invertibili. Con ciò intendiamo ad esempio che PrologACLMsg(X,Y) "costruisce" l'oggetto ACLMessage e lo unifica con Y se istanziamo X con un termine ground, mentre costruisce il termine rappresentante il messaggio in codifica PrologACLCode unificandolo con X se istanziamo la Y con un termine oggetto di classe ACLMessage.

Definiamo ora il codice tuProlog con cui si possono implementare i predicati di comunicazione offerti da $_N$ProlAg.

```
sync_receive(M):- PrologACLTemplate(M,T),
            me <- blockingReceive(T) return M1,
            PrologACLMsg(M,M1).

async_receive(M_in,M_out):- PrologACLTemplate(M_in,T),
            me <- receive(T) return M1,
            what(M1,M_in, M_out).
what(M1, M_in, M_out):- isnull(M1), !,
            M_out= no_message.
what(M1, M_in, M_out):- PrologACLMsg(M_in,M1).

send(M):- PrologACLMsg(M,ACL), me <- send(ACL).
```

L'idea è che ogni predicato di comunicazione venga trasformato nel rispettivo metodo JADE per la comunicazione. Si noti le primitive che permettono agli agenti JADE di comunicare vengono offerte nella forma di metodi della classe Agent. Per questo nel metodo setup avremo cura di rendere disponibile al motore

inferenziale il riferimento all'oggetto istanza di AgProlog attraverso l'operazione tuProlog

JavaLibrary.register(me, this)

Questa operazione ci permetterà attraverso il termine "me" associato al riferimento dell'agente di accedere a tutti i metodi di comunicazione.

5.6 Considerazioni conclusive

La progettazione che abbiamo appena descritto dovrebbe aver convinto il lettore che è possibile ottenere agenti Ag-Prolog, eseguibili sulla platform JADE. Ci siamo serviti della notazione UML non tanto con la pretesa di scrivere una vera e propria specifica quanto per esprimere in una forma più immediata le idee che dovranno guidare la realizzazione. A causa delle limitate risorse tempo e uomo di cui disponevamo abbiamo effettuato solo la "progettazione" dell'integrazione fra JADE e tuProlog. Si prevede di realizzare l'implementazione in un lavoro di tesi successivo. Non appena si disporrà di un supporto funzionante per gli agenti logici sarebbe interessante validare anche le idee (che trovano qui una forma esclusivamente embrionale) circa l'utilizzo degli agenti "dummy" per facilitare il testing di prototipi parziali del sistema che si sta sviluppando.

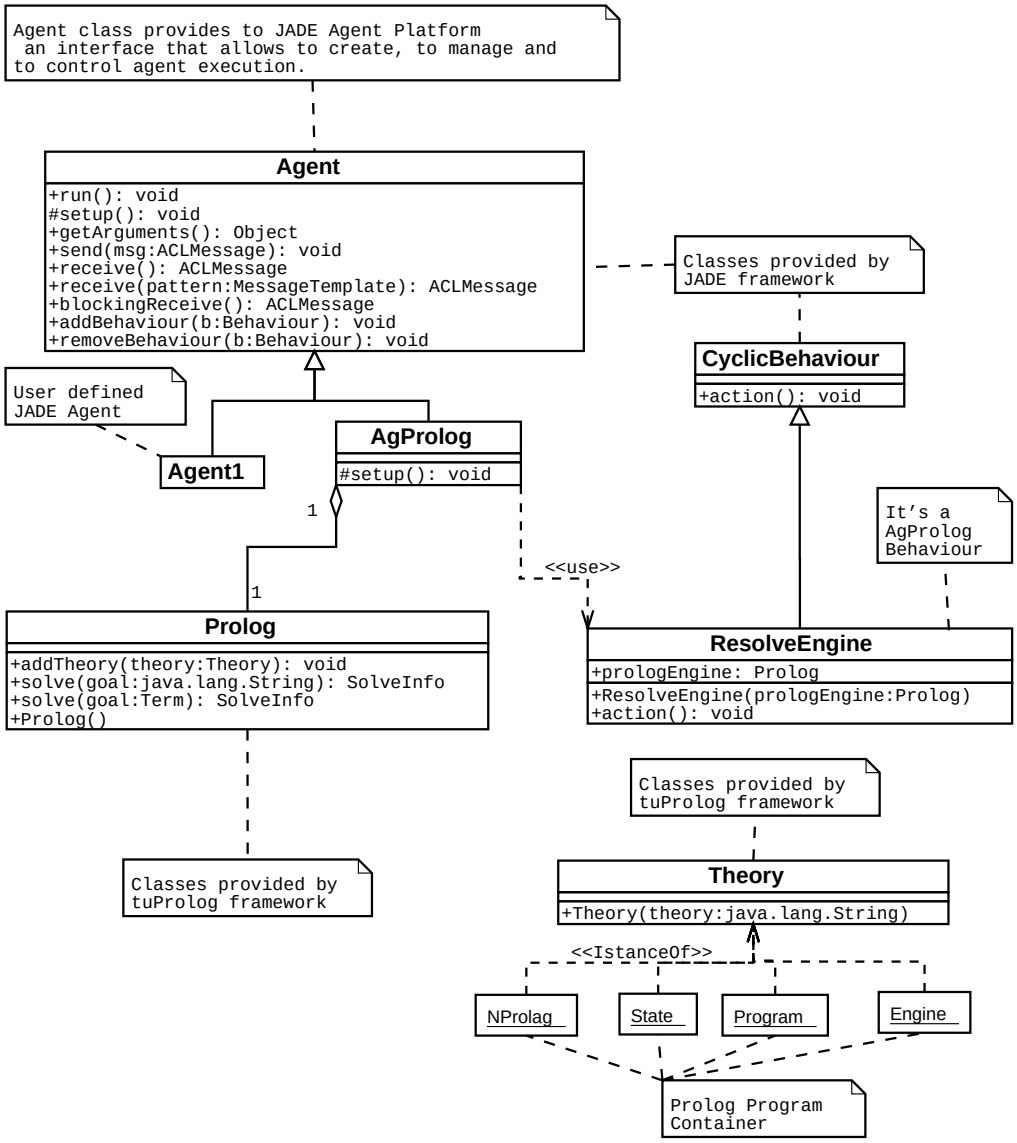

Figura 5.18: Il class diagram che spiega come integrare le differenti componenti per ottenere gli agenti Ag-Prolog.

Conclusioni e sviluppi futuri

Questo lavoro di tesi nasce dall'esigenza di estendere CaseLP: un ambiente per la prototipazione rapida di sistemi multi-agente basato sulla programmazione logica.

Una prima parte del lavoro effettuato è consistito nell'individuare i requisiti dell'estensione applicando le linee guida per il design proposte da Wieringa. In questa fase il candidato, la Dott. Viviana Mascardi e Stefano Miglia hanno definito un Functional Refinament Tree contenete le funzionalità che si prevedeva di prendere in considerazione.

Il Functional Refinament Tree delle estensioni è risultato abbastanza articolato da essere suddiviso in due progetti: M-CaseLP e D-CaseLP. Il primo racchiude le problematiche inerenti gli aspetti di progettazione di sistemi multi-agente e definizione di prototipi a partire da una libreria di componenti riutilizzabili. Il secondo tratta degli strumenti adatti ad ottenere un ambiente distribuito che permetta l'esecuzione, la simulazione, e monitoring di prototipi di agenti Ag-Prolog ed Ag-Java.

Vista l'ambizione del progetto, questo lavoro di tesi si concentra sugli aspetti relativi all'esecuzione ed il monitoring degli agenti Ag-Prolog e Ag-Java lasciando fuori gli aspetti di simulazione. Il contributo di questa tesi al progetto D-CaseLP sono stati:

- la valutazione di CORBA come tecnologia per ottenere la distribuzione degli agenti;
- lo studio e la valutazione delle specifiche FIPA per il design del tool D-CaseLP;
- lo studio e la valutazione del framework JADE;
- la progettazione dell'integrazione fra agenti logici (Ag-Prolog) e strumenti offerti dal run-time JADE.

Valutazione di CORBA.

La comunicazione fra agenti coinvolge tre livelli di astrazione: trasporto (protocollo di comunicazione), linguaggio di comunicazione per agenti ed ontologie;

l'interazione fra gli agenti si basa sul modello peer-to-peer nel quale ogni entità coinvolta ha gli stessi doveri e diritti di ogni altra.

Queste caratteristiche non sono supportate da CORBA il quale è un middleware per oggetti fortemente basato sul modello di comunicazione client-server. Inoltre, essendo progettato per distribuire componenti software non intelligenti, CORBA non prevede una comunicazione strutturata su diversi livelli di astrazione e fornisce supporto solo per il livello più basso della comunicazione per agenti: il trasporto.

I limiti riscontrati in CORBA hanno messo in luce l'inadeguatezza rispetto agli obiettivi di D-CaseLP. In particolare, l'impiego di CORBA per effettuare la distribuzione avrebbe comportato un onere eccessivo rispetto la scarsa forza lavoro di cui dispone il progetto.

Studio delle specifiche FIPA.

Il secondo contributo è consistito nello studio della proposta di standardizzazione effettuata dalla FIPA per gli aspetti attinenti a D-CaseLP.

La FIPA si prefigge di standardizzare l'intelligenza esterna dei MAS lasciando ampia libertà su quali architetture per agenti lo sviluppatore può utilizzare nella progettazione del suo sistema.

Il framework concettuale della FIPA offre un design in cui è fornita la soluzione ad alcune delle problematiche progettuali di D-CaseLP. In particolare è fornito attraverso il concetto di platform il livello di astrazione sufficiente a permettere

- l'aggiunta e l'eliminazione dinamica di istanze per le classi di agenti appartenenti al MAS;
- la sostituzione dei protocolli con cui gli agenti comunicano attraverso i servizi offerti dal Message Transport Service (MTS).

Lo studio delle specifiche FIPA ha portato a riflettere sulla possibilità di prendere in considerazione un tool FIPA compliant per la realizzazione del run-time in cui eseguire gli agenti Ag-Prolog ed Ag-Java. Con questo obiettivo si sono valutate le funzionalità offerte da JADE.

Valutazione di JADE.

JADE si pone come un vero e proprio middleware per agenti, e si basa su ipotesi sufficientemente minimali da essere utilizzato come run-time per l'esecuzione degli agenti previsti in D-CaseLP. Gli agenti JADE sono agenti Java sviluppati a partire da una meta-architettura a Behaviour. Tale meta-architettura per agenti offre un buon compromesso tra le esigenze di controllo descritte in FIPA ed i vincoli

imposti dal multithreading Java, lasciando all'utente la possibilità di istanziare la meta-architettura sulle architetture per agenti descritte in letteratura. Al fine di poter utilizzare JADE come middleware per l'esecuzione degli agenti previsti in D-CaseLP, è necessario supportare gli agenti Ag-Prolog nella stessa maniera in cui sono supportati gli agenti nativi JADE per permettere di applicare gli strumenti di monitoring e controllo anche sugli agenti logici. Tale esigenza ha portato alla progettazione dell'integrazione di agenti Ag-Prolog in JADE.

Gli agenti Ag-Prolog in JADE.

Prerequisito affinché sia possibile eseguire gli agenti Ag-Prolog in JADE è l'integrazione di Prolog in Java. Si sono analizzate diverse implementazioni Prolog allo scopo di individuarne una che offrisse un'interfaccia verso Java bidirezionale e sufficientemente astratta da non pregiudicare la portabilità degli agenti. Tali requisiti sono soddisfatti da tuProlog: un'interprete Prolog completamente realizzato in Java originariamente nato per scrivere componenti intelligenti da impiegarsi in applicazioni che fanno utilizzo della rete.

Attraverso l'analisi di alcuni use case e sfruttando le funzionalità di tuProlog si è progettata una particolare classe di agenti JADE in grado di incapsulare agenti logici definiti in termini di stato iniziale, program ed engine. Tale agenti permettono di eseguire agenti logici nella platform JADE usufruendo degli strumenti di monitoring e controllo che si hanno a disposizione per gli agenti nativi del tool.

Lavoro e sviluppi futuri.

Il progetto è stato suddiviso in due parti ed attualmente vi sono due tesi i cui obiettivi constano nel

- fornire un supporto metodologico alla progettazione di agenti Ag-UML (agenti specificati in UML);
- definire una libreria di componenti con l'ausilio delle quali fornire una traduzione semiautomatica in Ag-Prolog o Ag-Java;
- arricchire la libreria di componenti di alcune architetture per agenti ottenute partire dalla meta-architettura JADE.

Al di là di queste due tesi il lavoro da fare prima di ottenere un tool con i requisiti che ci siamo preposti è ancora lungo, indichiamo alcune delle attività che si possono prevedere:

- implementazione dell'integrazione fra Ag-Prolog e platform JADE secondo la progettazione illustrata in questa tesi;

- studio di strumenti per la simulazione, in particolare valutazione di strumenti che permettano di simulare canali di comunicazione inaffidabili utilizzando le caratteristiche dei Message Transport FIPA;
- studio di linee guida metodologiche ed eventuali strumenti per l'integrazione del software legacy sulla base delle specifiche prodotte dalla FIPA sull'argomento.

Elenco delle figure

1.1 Architettura generale. 9
1.2 La gerarchia architetturale in CaseLP. 10
1.3 Modello a spirale proposto da Bohem. 16
1.4 Tre modi per agentificare legacy software. 22
1.5 Schematizzazione dei concetti con cui viene caratterizzato il software da Wieringa. 25
1.6 Un prima suddivisione del progetto D-CaseLP. 29
1.7 Iter seguito in M-CaseLP. 33
1.8 Iter seguito in D-CaseLP. 34

2.1 Il Reference Model dell'OMA. 39
2.2 Componenti coinvolti in una comunicazione client-server CORBA. 41
2.3 FRT con evidenziate le funzionalità che saranno supportate dal primo prototipo di D-CaseLP. 44
2.4 Un modo possibile di per far comunicare gli agenti con l'impiego di placeholder. 46
2.5 Integrazione delle primitiva di comunicazione negli agenti prototipali di D-CaseLP. 49

3.1 Mission statement costitutivo della FIPA. 57
3.2 L'architettura astratta e le sue realizzazioni (specifiche di realizzazione). 60
3.3 La struttura della FIPA. 62
3.4 Il lyfe-cycle delle specifiche FIPA. 64
3.5 Le classi di specifiche della FIPA. 65
3.6 Relazione tra architettura astratta e concreta. 67
3.7 Coppie chiave-valore contenute nella directory entry dell'agente $\mathcal{A}$. 71
3.8 Chiave-valore di ricerca. 71
3.9 La struttura per un messaggio FIPA. 73
3.10 Trasformazione di un messaggio in un Transport-Message. 74
3.11 (a) Transport-Message per la comunicazione tra "123" e "ABC". (b) Lo stesso Transport-Message criptato. 75

3.12 Il modello di riferimento FIPA per la gestione degli agenti. 76
3.13 AID tipo del Directory Facilitator. 79
3.14 AID tipo del Agent Management System. 80
3.15 Il ciclo di vita degli agenti FIPA. 81

4.1 Aggiornamento dell'albero delle funzionalità D-CaseLP. 91
4.2 Progetti Open Source di tool per la realizzazione di MAS aderenti alle specifiche d'interoperabilità FIPA. 92
4.3 L'interfaccia grafica con cui si accedono le funzionalità del RMA. 96
4.4 L'interfaccia grafica dello tool Sniffer. 96
4.5 L'interfaccia grafica del tool Introspector. 97
4.6 La gerarchia dei Behaviour in JADE. 102
4.7 Architettura degli agenti JADE. 105
4.8 Dettaglio sulla l'iter di esecuzione del progetto D-CaseLP con l'ausilio di JADE. 114
4.9 Confronto tra il supporto offerto da JADE e CORBA. 115
4.10 Confronto tra il supporto offerto da JADE e CORBA su funzionalità che protrebbero essere incluse in un futuro prossimo. 116

5.1 Uno schema per la traduzione in codice eseguibile da CaseLP. . . 120
5.2 Sintassi formale per AgentRules 121
5.3 La traduzione delle istruzioni AgentRules. 121
5.4 Associazione tra regole AgentRules e predicati 122
5.5 Finestra per l'inizializzazione delle mailboxes. 124
5.6 finestra che descrive l'esecuzione on line. 124
5.7 Traccia off-line delle esecuzioni. 125
5.8 Dettagli di un evento tracciato. 125
5.9 Istanziamo due teorie in tuProlog. 131
5.10 Interfaccia per accedere agli oggetti rappresentanti il motore inferenziale Prolog. 131
5.11 Interfaccia **SolveInfo** per accedere alle soluzioni. 132
5.12 Esempio di programma Prolog che istanzia ed utilizza un oggetto Java. 133
5.13 Schematizzazione delle relazioni tra prototipi: (a) raffinamento, (b) intersezione, (c) indipendenza. 136
5.14 Esempio di come potrebbero essere coinvolti gli agenti in diversi prototipi. 137
5.15 Use case per D-CaseLP forniamo. 142
5.16 Use case per scrivere agenti logici. 144
5.17 Dettaglio sullo use case WriteArchitecture. 145

5.18 Il class diagram che spiega come integrare le differenti componenti per ottenere gli agenti Ag-Prolog. 153

Indice analitico

CORBA
- activation, 40
- active object map, 40
- deactivation, 40
- etherealization, 40
- incarnation, 40
- object, 38
- object Id, 39
- servant, 39
- skeleton, 39
- stub, 39

FIPA, 56
- FIPA ACL, 71
- Abstract Architecture
  - Agent Communication Language, 60
  - Directory Service, 60
  - Message Transport Service, 60
- Agent Identifier, 77
- Agent Management System, 79
- Agent Platform, 77
- AID, 77
- AMS, 79
- DF, 78
- Directory entry, 70
- Directory Facilitator, 78
- Directory Services, 70
- HAP, 78
- platform, 66
- principi
  - minimalità, 58
  - openness, 58
- specifiche
  - life cycle, 64
  - organizzazione, 65
- tipo opaco, 67

JADE, 92
- Architecture, 104
- Behaviour, 102
- Dummy Agent, 95
- Introspector Agent, 97
- Sniffer Agent, 95

MAS, 6

$_N$ProlAg, 141

ProlAg, 118

Prolog
- tuProlog, 129
- FLI, 128

CORBA
- OMA Model, 37

Agente, 6

agenti deliberativi, 16

agenti reattivi, 16

Context Boundary, 24

DCaseLPTool
- ACLtoMessageTemplate, 107

Distribuzione effettiva, 18

Environment, 23

green thread, 101

multitasking, 99

multithreading, 100

OMG, 37

openness, 58

Pattern Matching, 32
peer-to-peer, 47
PrologACLCode, 140

Sistema di Agenti Multipli, 6
Subject Domain, 23
System Boundary, 24

thread, 100
Traciability, 26

Bibliografia

[1] AgentBuilder. Reticular Systems, Inc. http://www.agentbuilder.com/.

[2] The aliCE research team — DEIS - University of Bologna/Cesena, Italy. tu*Prolog Documentation*, October 2001. work in progress.

[3] E. Appiani, M. Martelli, and V. Mascardi. A Multi-Agent Approach to Vehicle Monitoring in Motorway. Technical Report DISI TR-00-13, 2000.

[4] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi agent systems with JADE, 2001.

[5] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, and Giovanni Rimassa. *Jade Administrator'S Guide*. CSELT S.p.a, July 2001. Copyright © 2000 CSELT S.p.A, © TILab 2001 S.p.A.

[6] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, and Giovanni Rimassa. *Jade Programmer'S Guide*. CSELT S.p.a, September 2001. Copyright © 2000 CSELT S.p.A, © TILab 2001 S.p.A.

[7] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE — A FIPA-compliant agent framework. In *Proceedings of the 4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99)*, pages 97–108, 1999.

[8] Fabio Bellifemine, Agostino Poggi, Giovanni Rimassa, and Paola Turci. An object-oriented framwork to realize agent system.

[9] binprolog. BinNet Corporation. http://www.binnetcorp.com/BinProlog/.

[10] Rodney A. Brooks. Intelligence without reason. In Ray Myopoulos, John; Reiter, editor, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann.

[11] Rodney A. Brooks. Intelligence without representation. Number 47 in Artificial Intelligence, pages 139–159. 1991.

[12] J. Brustoloni. Autonomous agents: Charaterization and requirements. Technical report, Pittsburgh, 1991.

[13] Frank Mc Cabe. Sociological aspect of the internet revolution. *FIPA inform!*, 1(3):3, 2000. FIPA newsletters.

[14] P. Charlton, R. Cattoni, A. Potrich, and E. Mamdani. Evaluating the FIPA standards and its role in achieving cooperation in multiagent system. IEEE Computer Society, January 2000. 33rd Annual Hawaii International Conference on System Sciences (HICSS-33).

[15] L. Console, E. Lamma, and M. Milano. *Programmazione Logica e Prolog*. UTET, 1997.

[16] A. Cuppari, P. L. Guida, M. Martelli, V. Mascardi, and F. Zini. An Agent-Based Prototype for Freight Trains Traffic Management. In P. G. Larsen, editor, *Proc. of FMERail Workshop 5*, Toulouse, France, September 1999. Springer-Verlag. A satellite workshop of FM'99.

[17] A. Cuppari, P. L. Guida, M. Martelli, V. Mascardi, and F. Zini. Prototyping Freight Trains Traffic Management Using Multi-Agent Systems. In *Proc. of IEEE International Conference on Information, Intelligence and Systems*, Washington, DC, November 1999. IEEE.

[18] R. Degl'Innocenti. UML come linguaggio per specificare agenti: analisi, estensione ed integrazione in CaseLP. Master's thesis, DISI – Università di Genova, Genova, Italy, 2000. In Italian.

[19] G. Delzanno. *Logic & Object-Orinted Programming in Linear Logic*. PhD thesis, Universita of Pisa, Dipartimento di Informatica, December 1996.

[20] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A lightweight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer-Verlag, 2001. 3rd International Symposium (PADL 2001), Las Vegas (NV), 11–12 March 2001.

[21] ECLiPSE. © IC-Parc 1997-2001, http://www.icparc.ic.ac.uk/eclipse/index.html.

[22] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, Maryland, 1994. ACM Press.

[23] FIPA Home Page. Foundation for Intelligent Physical Agents. http://www.fipa.org/.

[24] Foundation for Intelligent Physical Agents. *FIPA ACL Message Structure Specification*, August 2000. FIPA, Approved For Experimental 01-08-2000.

[25] Foundation for Intelligent Physical Agents. *FIPA Agent Software Integration Specification*, August 2000. FIPA, Approved For Experimental 14-06-00.

[26] Foundation for Intelligent Physical Agents. *FIPA Ontology Service Specification*, June 2000. FIPA, Approved For Experimental 15-06-2000.

[27] Foundation for Intelligent Physical Agents. *FIPA Abstract Architecture Specification*, February 2001. FIPA, Approved For Experimental 29-01-01.

[28] Foundation for Intelligent Physical Agents. *FIPA Agent Management Specification*, August 2001. FIPA, Approved For Experimental 15-08-01.

[29] Foundation for Intelligent Physical Agents. *FIPA Agent Message Transport Service Specification*, August 2001. FIPA, Approved For Experimental 15-08-01.

[30] Foundation for Intelligent Physical Agents. *FIPA Communicative Act Library Specification*, October 2001. FIPA, Approved For Experimental 29-01-2001.

[31] Graesser Franklin. Is it an agent, or just a program?: A taxonomy for autonomous agents. Technical report, 1996.

[32] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):49–53, July 1994.

[33] GNU PROLOG. ftp://ftp.inria.fr/INRIA/Projects/loco/gnu-prolog/.

[34] O. Inc. Orbacus for C++ and Java, 1998.

[35] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[36] Jinni. BinNet Corporation. http://www.binnetcorp.com/Jinni/.

[37] jprolog. http://www.cs.kuleuven.ac.be/ bmd/PrologInJava/.

[38] David Kinny, Michael Georgeff, and Anand Rao. A methodology and modelling technique for systems of BDI agents. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[39] Robert A. Kowalski and Fariba Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):391–419, 1999.

[40] M. Martelli, V. Mascardi, and F. Zini. CaseLP: a Complex Application Specification Environment Based on Logic Programming. In *Proc. of ICLP'97 Post Conference Workshop on Logic Programming and Multi–Agents*, pages 35–50, Leuven, Belgium, July 1997.

[41] M. Martelli, V. Mascardi, and F. Zini. A Logic Programming Framework for Component-Based Software Prototyping. In A. Brogi and P. Hill, editors, *Proc. of 2nd International Workshop on Component-based Software Development in Computational Logic (COCL'99)*, Paris, France, September 1999.

[42] M. Martelli, V. Mascardi, and F. Zini. Specification and Simulation of Multi-Agent Systems in CaseLP. In M. C. Meo and M. Vilares-Ferro, editors, *Proc. of Appia–Gulp–Prode 1999: Joint Conference on Declarative Programming*, pages 13–28, L'Aquila, Italy, September 1999.

[43] V. Mascardi. *CaseLP, Logic-Based Specification Enviroments For Multi-Agent Systems*. PhD thesis, Università of Genova, Dipartimento di Informatica e Scienze dell'Informazione, December 2001.

[44] D. Miller. Forum: A multiple-conclusion specification logic, 1996.

[45] MINERVA. http://www.ifcomputer.co.jp/MINERVA/home_en.html.

[46] MokeByte. *The Moka Book*, 2001. MokaBook © 2001 - MokaByte s.r.l. `http://www.mokabyte.it/mokabook/`.

[47] Hyacinth S. Nwana and Divine T. Ndumu. A perspective on software agents research. *The Knowledge Engineering Review*, 14(2):1–18, 1999.

[48] Object Management Group. *Unified Modelling Language Specification, version 1.3*, March 2000. OMG document formal/00-03-01. URL: `http://cgi.omg.org/cgi-bin/doc?formal/00-03-01.ps.gz`.

[49] Object Oriented Concept Inc. *The Orbacus Homepage*, 2000. http://www.ooc.com/ob/.

[50] M. De Pedrini. CaseLP Visualizer: un tool di visualizzazione per sistemi Multi-agente. Master's thesis, DISI – Università di Genova, Genova, Italy, 1998. In Italian.

[51] J. Pitt, A. Mamdani, and P. Charlton. The open agent society and its enemies: a position statement and research programme. In *Telematics and Informatics*, volume 18, pages 67–87. PERGAMON, January 2001. © 2001 Elsevier Science Ltd.

[52] Stefan Poslad and Patricia Charlton. Standardizing agent interoperability: The FIPA approach. In M. Luck et al, editor, *Multi-Agent Systems and Applications*, volume 2086, pages 98–117, Prague, Czech Republic, July 2001. Springer-Verlag. 9th ECCAI Advanced Course ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School, EASSS 2001.

[53] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[54] D. Schmidt and S. Vinoski. Object interconnections: Concepts and terminology, Oct 1997.

[55] SICStus Prolog. SICS AB. http://www.sics.se/sicstus.html.

[56] SWI-Prolog. http://www.swi.psy.uva.nl/projects/SWI-Prolog/.

[57] tuProlog. http://lia.deis.unibo.it/research/2P/.

[58] tuProlog JavaDoc API Documentation. http://lia.deis.unibo.it/research/2P/javadoc/index.html.

[59] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[60] Steve Vinoski. New features for CORBA 3.0. *Communications of the ACM*, 41(10), 1998.

[61] Jan Wielemaker. *A CORBA IDL compiler for SWI-Prolog*. University of Amsterdam, The Netherlands. http://www.cs.vu.nl/ eliens/documents/corba/pl/plcorba.html.

[62] R.J. Wieringa. *Design Method for Reactive Systems*. University of Twente, 2000.

[63] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[64] Word Wide Web Consortium. Extensible Markup Language (XML). W3C Working Draft available at `http://www.w3.org/TR/xml`.

[65] The Zeus Agent Building Toolkit. British Telecommunications. http://www.labs.bt.com/projects/agents/zeus/.

[66] F. Zini. *CaseLP, a Rapid Prototyping Environment for Agent-Based Software*. PhD thesis, Università of Genova, Dipartimento di Informatica e Scienze dell'Informazione, December 2000.