Integrazione di Agenti Logici in DCaseLP

Anno Accademico 2003/2004

Tesi di Laurea
presentata all'Università degli studi di Genova
Facoltà di Scienze Matematiche Fisiche e Naturali
Dipartimento di Informatica e Scienze dell'Informazione (DISI)

22 Marzo 2005

Candidata: Ivana Gungui

Relatori: Prof. Maurizio Martelli

Dott. Viviana Mascardi

Co-relatrice: Prof. Gianna Reggio

Ringraziamenti

I miei ringraziamenti dovrebbero in realtá occupare pagine e pagine di questa tesi, ma purtroppo devo essere breve, cosa che non mi riesce **mai e poi mai**. Vediamo cosa riesco a fare.....

Le prime persone che voglio ringraziare sono (e poteva essere diversamente?!?) i miei familiari che, nonostante non sapranno mai quanto l'universitá sia molto diversa dal liceo, a loro modo mi hanno sostenuto e mi vogliono molto bene.

Un grazie enorme va soprattutto a mio padre, **Salvatore**, che nel suo modo tutto particolare mi ha aiutato un sacco.

Mia madre, **Sheila**, se ne merita davvero tanti di ringraziamenti e comunque non sarebbero mai abbastanza!

Alle mie sorelle, mia nonna, Stefano, Vale, i miei zii e tutta la famiglia dico "GRAZIE" e li abbraccio tutti.

I miei piú sinceri ringraziamenti vanno ai miei relatori, **Maurizio** e **Viviana**, perché mi hanno dato la massima fiducia e guidata in quest'avventura finale. Un grazie, ovviamente, anche alla mia correlatrice. A Viviana, peró, vanno maggiori ringraziamenti perché anche in momenti critici ha trovato il tempo di pensare alla mia tesi e incoraggiarmi.

A questo punto, viene la parte piú difficile perché non ho idea da chi cominciare e non vorrei MAI dimenticare qualcuno ...
Forse un elenco é davvero la soluzione migliore!
Vorrei dire grazie a:

- **Stefano** perché é stato il mio piú caro amico, siamo cresciuti insieme, ha sempre avuto fiducia in me anche quando io ero la prima a dubitare di me stessa e, infine, é anche grazie a lui se sono arrivata ad Informatica;

- **Cristina** perché abbiamo passato momenti indimenticabili e ci siamo davvero divertite e confortate quando necessario;

- **Diana** e **Franco** per tutte le sere passate insieme;

- **Antonella** perché la nostra é una vera amicizia e, anche se le sembrerá strano, i complimenti piú belli li ho ricevuti proprio da lei;
- **Maura** e **Catuscia** per avermi fatto decidere di rimanere al DISI, ma soprattutto un enorme grazie lo devo a Maura perché si é dimostrata davvero un'amica oltre che ottima insegnante;
- **Monica** perche' e' una carissima amica e studiare con lei é stata una bella esperienza;
- **Stefano**, **Gabriele** e parenti perché mi hanno fatto sentire parte della loro famiglia;
- **Giulio** perché, anche se discutiamo spesso, cerca sempre di sostenermi;
- **Cri** perché é la Chichi e mi fa sorridere sempre;
- **Massimo** e **Chiara** per il passato;
- **Paola** perché se non ci facciamo i dispetti non siamo contente, e in questi ultimi giorni si é persa molto sonno per colpa mia: **GRAZIE**!!!!;
- **Mirko** e **Ombretta** perché sono cari amici e ci sono sempre quando ho bisogno (anche se mipiacerebbe che imparassero ad essere puntuali!);
- **Vale**, **Maury** e **Roby** perché con loro mi sono divertita tanto, insieme anche a Luca e Paola e Daniela;
- i professori al **DISI** e le ragazze delle pulizie perché mi hanno fatto sentire a casa;
- **Claudio** per il sostegno in chat;
- **Luca** ed **Alessio** e tutta la compagnia dei "comici" perché in questo ultimo anno grazie a loro ho riso tanto;
- **Federico** perché é l'unico che non vedeva l'ora di leggere la mia tesi.

Ho senza dubbio dimenticato qualcuno e mi dispiace.

Infine, ci tengo a dimostrare **tantissima gratitudine** a:

Google™ Italia

OneLook®

Dictionary Search

Introduzione

In questa tesi, il termine senza dubbio più citato è *agente*, quindi sembra naturale introdurre il nostro lavoro parlando di ciò che esso indica e del motivo per cui gli "ingegneri del software" si dedicano allo sviluppo di sistemi che sono costituiti da uno o più agenti.

Negli ultimi vent'anni, il concetto di *agente intelligente/autonomo* è stato soggetto a varie interpretazioni nel campo dell'informatica: in questa introduzione ne vedremo solo alcune.

Lo scopo di questa introduzione è di dare a chi la legge un'idea generale di concetti che sono argomenti fondamentali in questa tesi.

Agenti software intelligenti

Fin dagli anni '80, gli *agenti software* ed i *sistemi multi-agente* (spesso abbreviati con MAS da "Multi-Agent System") sono divenuti un'importante area di ricerca nella comunità informatica. Gli agenti ed i MAS sono ormai considerati *una nuova tecnologia software*, la cosiddetta **tecnologia ad agenti**, argomento ormai molto popolare nelle pagine Web riguardanti l'informatica o la robotica.

Il concetto di *agente* (intelligente) è nato nel campo dell'Intelligenza Artificiale, l'area dell'informatica che tratta la progettazione di sistemi software in grado di comportarsi in modo "intelligente".

In altre parole, il termine agente è stato scelto per indicare un programma software od un sistema costruiti in modo tale che il loro comportamento sia simile a quello dell'essere umano.

Inaspettatamente, tale termine ha ottenuto successo anche in altri campi informatici, e può essere incontrato in diversi contesti.

Pur essendo ampiamente usato da molte persone che lavorano in aree strettamente correlate, tale termine non possiede una definizione che sia universalmente accettata. Ancora oggi rimane aperto il dibattito su questa materia,

ma sembra che una caratteristica fondamentale degli agenti sia l'*autonomia*.

Il problema principale è che gli agenti possono essere considerati in molti domini differenti che, a loro volta, possono richiedere ad ogni agente abilità diverse.

Sono in molti a credere che la grande crescita della rete globale Internet abbia aiutato la larga diffusione degli agenti in molti domini di applicazione. Una definizione usuale che si può trovare proprio in Internet è la seguente:

> *"Un agente (intelligente) è un programma che recupera informazioni o effettua altre azioni senza l'intervento di una persona. Tipicamente, un agente usa parametri che gli sono stati precedentemente forniti (manualmente od automaticamente) per ricercare in tutta la rete Internet, od una sua parte, informazioni di interesse per l'utente, e gliele restituisce periodicamente."*. [Wha]

Una tale definizione, però, è parziale, non include altri aspetti su cui, invece, si è concentrata la comunità informatica, e non rispecchia nessuna delle tante definizioni che sono state date nell'ultimo decennio; si può dire che essa descrive semplicemente un tipo di agente.

Una definizione molto famosa e più generale della precedente è quella data da N. R. Jennings, K. Sycara e M. Wooldridge in [JSW98]:

> "*... un sistema computerizzato,* ***situato*** *in un qualche ambiente e capace di azioni* ***autonome*** *e* ***flessibili****, allo scopo di ottenere gli scopi prefissati.*"

Situato vuole indicare che il sistema è in grado di ricevere informazioni da sensori collocati nell'ambiente circostante, e di eseguire azioni che in qualche modo modificano l'ambiente stesso.

Autonomo e flessibile includono nel loro significato la capacità di decidere, in base alle circostanze, quali azioni intraprendere senza che vengano coinvolte persone o altri agenti.

Esistono molte varianti di agente: *agenti autonomi*, *agenti mobili*, *agenti distribuiti*, *agenti che apprendono*, *agenti pianificatori*, *agenti mediatori*, *agenti di visualizzazione personalizzati*, e molti altri ancora, a seconda dell'area in cui sono impiegati.
Alcune persone considerano l'intelligenza come una caratteristica implicita nell'"autonomia"; infatti, i termini "agente *intelligente*", "agente *autonomo*" e "agente *software intelligente*" sono spesso usati come sinonimi.

Gli agenti software possono anche essere caratterizzati attraverso "comportamenti", quali per esempio: *reattivo*, *deliberativo*, *guidato dallo scopo*, *adattivo*, *autonomo*, *comunicativo*, *proattivo*, *competitivo* e *collaborativo*.

Reagire ad eventi che accadono nell'ambiente viene considerato un comportamento *reattivo*. La capacità di intraprendere, nel corso del tempo, azioni coordinate è un comportamento *deliberativo*. Agire, invece, per ottenere un determinato scopo è un comportamento *guidato dallo scopo*.

La capacità di imparare ad adattarsi a nuove circostanze viene reputato un comportamento *adattivo*. Un comportamento *comunicativo* o *sociale*, al contrario, denota la capacità di interagire , quando opportuno, con altri agenti artificiali oppure esseri umani.

Il comportamento *proattivo* corrisponde all'agire in funzione di scopi futuri o, meglio, al pianificare le azioni; in realtà è una combinazione di quello reattivo e quello guidato dallo scopo.

Infine, il comportamento *competitivo* è quello per cui l'agente è in grado di competere con altri agenti, umani o non, per ottenere risorse condivise oppure un proprio scopo; quello *collaborativo*, invece, corrisponde alla capacità di cooperare insieme ad altri agenti per raggiungere un obiettivo comune.

La realizzazione di agenti che posseggano tutti questi comportamenti è, senza dubbio, un compito non facile, soprattutto perchè spesso un comportamento intelligente è in realtà costituito dalla combinazione non precisabile di tali comportamenti. Alcuni ricercatori si sono concentrati nella realizzazione di strutture software caratterizzate solo da alcuni di tali comportamenti, definendo così particolari *architetture di agenti*. Alcuni riferimenti a documenti su architetture per agenti sono forniti alla fine della sezione seguente.

Un altro fattore che va considerato riguardo agli agenti è: *per quanto tempo debbano rimanere in esecuzione*. L'istanza di un agente può rimanere in esecuzione per sempre oppure indefinitamente, cercando di conseguire continuamente i suo obiettivi, ma potrebbe anche terminare la sua esistenza una volta che i suoi scopi sono stati raggiunti oppure quando essi non sono più necessari.

Dopo questa introduzione generale al concetto di agente, passiamo a spiegare cosa intendiamo per comunità di agenti, spesso anche chiamata sistema multi-agente: più agenti indipendenti che cooperano e/o competono per realizzare gli obiettivi comuni e/o quelli individuali.

Un MAS può essere costituito da agenti stabiliti al momento della progettazione oppure può essere "aperto" a nuovi agenti esterni. Un concetto fondamentale del MAS è l'***interazione tra agenti***: essi devono essere in

grado di cooperare, coordinarsi e negoziare quando necessario.

Ingegneria del software basata sugli agenti (AOSE, "Agent-Oriented Software Engineering")

L'ingegneria del software *è* un'area informatica che studia il processo di sviluppo di un prodotto software — soprattutto sistemi *grandi* e *complessi* —, per renderlo economico e breve. Prima tratta la progettazione, poi l'implementazione e, infine, la consegna e la "manutenzione" del prodotto in questione.

In quest'area, gli strumenti (software) CASE ("Computer Aided Software Engineering", ingegneria del software supportata dal computer) sono quelli che cercano di aiutare tale processo con l'ausilio dei calcolatori e, oramai, hanno acquisito una grande rilevanza, avendo automatizzato molte delle attivit*à* coinvolte nelle varie fasi della produzione del sistema, rendendone pi*ù* efficiente lo sviluppo.

Gli strumenti software di sviluppo automatizzati solitamente sono implementati da organizzazioni che vogliono aumentare la produtivit*à*, diminuire i costi, migliorare i controlli del progetto ed aumentare la qualit*à* del prodotto.

Il modello tradizionale del processo di ingegneria del software suggerisce di organizzare lo sviluppo del software in fasi *successive*. Nella prima fase vengono analizzati i requisiti del sistema, poi la struttura fondamentale viene progettata attraverso la *decomposizione in componenti* facili da modellare.

Tali componenti prima di essere integrate e testate in un'unica entit*à*, vengono implementate e testate separatamente, per facilitare il rilevamento di loro comportamenti non voluti o, addirittura, errati. L'ultima fase del processo *è* dedicata alla manutenzione del software installato.

Questo modello, il cosiddetto *modello a cascata*, proposto nel 1970 da W.W.Royce [Roy70], presenta alcuni difetti, il pi*ù* importante dei quali *è* la mancanza di interazione con le fasi precedenti.

Durante l'analisi o la progettazione lo sviluppatore pu*ò* introdurre errori che possono non essere rilevati fino alla fase di testing ed, in tal caso, la loro correzione risulta essere molto costosa.

Gi*à* diversi anni fa sono stati proposti nuovi modelli migliori, tra i quali meritano di essere menzionati la *prototipazione* ed il *modello a spirale* — quest'ultimo ideato da B.Boehm nel 1986 [Boe86] — che incoraggiano un *approccio incrementale*: creare un prototipo "leggero" ma eseguibile, in grado di rilevare malfunzionamenti gi*à* in fase di progettazione, e da cui sia possibile ricavare informazioni utili al suo miglioramento per poi testarlo nuovamente, e cos*ì* via.

Questo innovativo *ciclo di prototipazione* viene ripetuto fino a quando il software *è* terminato oppure la poca disponibilit*à* di tempo/soldi permette solo di consegnare una versione con limitate capacit*à*.

Molto noto nella comunit*à* informatica e quella industriale *è* l'approccio orientato agli oggetti dell'ingegneria del software, che ha portato alla OOSE ("Object-Oriented Software Engineering",ingegneria del software orientata agli oggetti). L'approccio orientato agli oggetti consiste nel modellare un sistema tramite oggetti che interagiscono e che possono avere determinati schemi di comportamento: persone, oggetti reali e non, possono essere tutti modellati come oggetti.

Gli oggetti sono dotati sia di struttura sia di comportamento: la struttura rappresenta le informazioni sull'oggetto e le sue propriet*à* statiche, mentre il comportamento rappresenta le sue propriet*à* dinamiche.

La propriet*à* di cui si parla di pi*ù*, ma non per questo la pi*ù* importante, riguardo agli oggetti *è l'ereditarietà*: nell'informatica funziona pi*ù* o meno come l'ereditarieta*à* che esiste in natura. Come i bambini ereditano la struttura (ad es. lo scheletro, i muscoli, ecc.) ed il comportamento (abitudini, preferenze, ecc.) dai loro genitori, allo stesso modo gli oggetti ereditano la struttura (metodi, attributi, ecc.) e comportamento (un diagramma di un automa a stati, per es.) dagli oggetti di cui sono eredi. Il meccanismo dell'ereditarietà permette di risparmiare tempo e denaro durante lo sviluppo di prodotti nuovi od aggiornati.

Gli agenti software sono il passo evolutivo successivo agli oggetti ed ai programmi. Le ragioni per l'attuale grande interesse nei confronti degli agenti sono molte, ma certamente una delle pi*ù* importanti *è* che il concetto di agente come sistema autonomo, capace di interagire con altri agenti allo scopo di soddisfare i suoi obiettivi, risulta essere un concetto naturale ai progettisti di software.

L'ingegneria del software orientata agli agenti (AOSE, "Agent-Oriented Software Engineering") *è* uno dei campi pi*ù* recenti dell'ingegneria del software: risulta molto stimolante per gli ingegneri del software il compito di analizzare, progettare ed implementare software che sia costituito da un insieme di agenti autonomi che interagiscono.

Questo approccio ha diversi vantaggi rispetto ad altri approcci di sviluppo del software esistenti, in primo luogo la capacit*à* degli agenti di rappresentare astrazioni ad alto livello di entit*à* attive in un sistema.

Gli oggetti tendono ad essere passivi, essendo attivo un solo oggetto alla volta, e "nascosto" in programmi. I programmi, a loro volta, tendono ad essere relativamente statici, quindi diventa difficile progettare computazioni

distribuite con un numero indeterminato di programmi collaboranti.

Gli agenti, invece, sono dinamici perchè mostrano un certo grado di attività piuttosto che la fornitura passiva di servizi. Gli agenti software offrono opportunità sia per il parallelismo sia per la collaborazione tra programmi su larga scala in un modo facile, robusto, affidabile, flessibile e sicuro.

Un'altra differenza rilevante tra gli oggetti e gli agenti è che i primi sono controllati dall'esterno (come una scatola trasparente), mentre i secondi hanno un comportamento autonomo che non può essere direttamente controllato dall'esterno (come una scatola nera).

I linguaggi basati sulla logica sono più adatti, rispetto a quelli orientati agli oggetti, a modellare gli agenti non solo perchè possono esprimere il comportamento attivo degli agenti in una maniera concisa e semplice, ma anche perchè riescono a modellare agenti che reagiscono ad un evento — a questo scopo possono essere usate implicazioni logiche della forma "se accade l'evento E, allora qualcosa diventa vero" — e, infine, possono anche "ragionare" su sofisticate conversazioni.

La tecnologia ad agenti gioca un ruolo centrale nello sviluppo di sistemi complessi distribuiti e sistemi informatici in rete.

Un MAS, formato da più entità autonome che lavorano (o competono) per raggiungere un dato scopo, rappresenta un modo naturale per modellare correttamente la decentralizzazione, le prospettive multiple e gli interessi competitivi dei vari agenti nel MAS, ma anche per modellare sistemi informativi che sono distribuiti, aperti ed eterogenei.

Così, lo scopo principale dell'AOSE è quello di creare metodologie e strumenti che permettano, in modo economico, lo sviluppo ed il mantenimento di software basato sugli agenti. Il software, inoltre, deve essere flessibile, facile da usare, scalabile e di alta qualità; in altre parole, lo scopo di AOSE è simile a quello degli altri rami dell'ingegneria del software, ma si adopera per migliorare la coordinazione, la cooperazione, la comunicazione e l'intelligenza negli agenti e nei MAS.

Linguaggi di Comunicazione per Agenti

Un altro aspetto, non meno importante, è la comunicazione tra gli agenti; due standard che sono stati definiti per lo scambio di messaggi tra agenti sono: **KQML** [FLM95] e l'**ACL** di FIPA ("Foundation for Intelligent Physical Agents") [ACL].

FIPA è un'organizzazione internazionale non a scopo di lucro che, dal 1996, ha prodotto standard per l'interazione di agenti software eterogenei ed ha

promosso generiche tecnologie ad agenti.

Sia KQML sia l'ACL di FIPA permettono lo scambio di messaggi che hanno un contenuto dichiarativo e non considerano la struttura interna dell'agente mittente/destinatario.

L'*ontologia* di un agente è una specifica formale che definisce il vocabolario dei termini conosciuti dall'agente ed, eventualmente, una specifica del significato di tali termini. I linguaggi per agenti implementano protocolli di comunicazione che non conoscono nè il linguaggio usato per esprimere il contenuto, nè il meccanismo per specificare l'ontologia.

Anche se non esiste un linguaggio standard di comunicazione per agenti universalmente accettato, è interessante notare che la maggior parte di quelli proposti condividono l'assunzione che la comunicazione tra agenti debba essere trattata in termini di *speech acts* (atti comunicativi).

La nozione di speech act proviene dalla filosofia del linguaggio [Aus62, Sea69] ed è stata adottata dagli autori di KQML e da FIPA come base per la comunicazione tra agenti.

La teoria degli atti comunicativi è una teoria sul significato del linguaggio molto potente e vasta; considerando la comunicazione tra agenti come una forma di azione, rende facile e naturale inserire una componente comunicativa nell'architettura di un agente.

Un linguaggio di comunicazione ad agenti specifica solo l'*intenzione* alla base di un messaggio ed include il contenuto del messaggio come un elemento separato: tale intenzione è specificata esplicitamente nel messaggio scambiato, indicando un atto comunicativo standard scelto da un insieme predefinito.

In altre parole, questi linguaggi derivano le loro primitive di comunicazione dalla teoria linguistica e distinguono varie tipologie di atti comunicativi, a seconda dell'intento dell'agente che effettua la comunicazione.

L'insieme di standard che questi linguaggi permettono, costituisce le basi per ogni significativo tipo di conversazione.

Il contenuto di un messaggio ACL (cioè appartenente ad uno dei linguaggi per la comunicazione degli agenti) ricevuto da un agente può essere capito da quest'ultimo solo se condivide una comune *ontologia* con l'agente mittente.

A causa della mancanza di standard universalmente accettati, uno strumento di sviluppo che supporti AOSE deve offrire dei "mattoni" per alcune architetture di agenti e il supporto per uno dei linguaggi di comunicazione per agenti disponibili.

Un altro argomento di molto interesse nella comunità che studia gli agenti è senza dubbio l'*architettura di agenti*.

L'architettura di un agente è la particolare metodologia usata per creare

l'agente e specifica come esso possa essere decomposto in un insieme di moduli componenti e come questi moduli dovrebbero interagire. Il lettore interessato ad architetture di agenti — reattive, BDI, a livelli e cosı' via — può, per esempio visitare le seguenti pagine Web:

- "*A Survey of Cognitive and Agent Architectures*" (completato nel 1994 alla University of Michigan, Department of Electrical Engineering and Computer Science) - *http://ai.eecs.umich.edu/cogarch0/* ;
- "*AGENT ARCHITECTURES*" contiene molti riferimenti - *http://www.ecs.soton.ac.uk/~mml/absd/chap2.html*;
- "*Agent Architectures*" - *http://www.csc.liv.ac.uk/ mjw/pubs/ker95/section3_3.html*;
- "*Roadmap - Where we go next!*" - *http://www7.in.tum.de/~weissg/Lehre/AosmSS02/aosm-ss02-2.slides.pdf* .

Lo scopo della tesi

Lo scopo di questa tesi è:

> *Compiere un passo avanti verso la* ***coesistenza di agenti eterogenei*** *nell'implementazione di un prototipo di un sistema multi-agente sviluppato attraverso lo strumento software DCaseLP, integrando in questo stesso ambiente software un nuovo tipo di agenti, chiamati "****agenti logici****".*

DCaseLP (***D****istributed* ***CaseLP***) [AMMM02] e' uno strumento CASE che è nato come successore di un altro ambiente software, CaseLP (***C****omplex* ***a****pplication* ***s****pecification* ***e****nvironment based on* ***L****ogic* ***P****rogramming*) [MMZ99, BDM+99], entrambi sviluppati dal *Gruppo di Programmazione Logica* del Dipartimento di Informatica e Scienze dell'Informazione, all' Università di Genova.

CaseLP fornisce un metodo di prototipazione ed un insieme di strumenti e linguaggi che supportano la realizzazione di prototipi di applicazioni complesse modellate come MAS.

Consente una specifica ed un'architettura *multilinguali* degli agenti appartenenti al MAS sviluppato, ma non supporta un'implementazione multilinguale di tali agenti; in altre parole, lo sviluppatore può usare linguaggi diversi per la specifica e l'architettura di ciascun agente, mentre *solo un* linguaggio è supportato per programmare gli agenti.

DCaseLP *è* nato allo scopo di eliminare i limiti di CaseLP e diventare uno *strumento di prototipazione multilinguale AOSE completo.* La prima versione di DCaseLP ha fatto un primo passo verso il raggiungimento di questo obiettivo proponendo *due linguaggi per l'implementazione degli agenti*: il linguaggio **Java**^{TM} [Mic] ed il linguaggio per sistemi esperti **Jess** [FH].

Sfortunatamente, la multilingualità di questa versione era ancora troppo limitata e le funzionalità offerte da CaseLP erano ancora non utilizzabili da esso, perchè DCaseLP non comprendeva il codice basato su Prolog e gli strumenti già sviluppati per CaseLP.

Quello di cui avevamo bisogno erano "**agenti logici**" che soddisfacessero le seguenti proprietà:

- il loro comportamento *è* specificato interamente tramite un linguaggio Prolog;
- sono in grado di "ragionare", nel senso che possono eseguire quella che in programmazione logica viene definita una "*dimostrazione*";
- possono comunicare con agenti sviluppati con DCaseLP.

CaseLP *è* implementato in SICStus Prolog [SIC] ed i suoi agenti sono fondamentalmente codice SICStus Prolog esteso con primitive di comunicazione ad–hoc.

Una volta che gli agenti logici saranno disponibili in DCaseLP, quelli precedentemente sviluppati con CaseLP potranno essere eseguiti in un prototipo sviluppato con DCaseLP: la sola azione necessaria sarà la traduzione degli "agenti CaseLP" da SICStus Prolog al linguaggio Prolog usato per creare gli agenti logici.

Riteniamo che tale traduzione sarà facilmente automatizzabile permettendo, così, allo sviluppatore di usare gli strumenti sviluppati per CaseLP.

Struttura della tesi

Questa tesi é strutturata come segue. Il Capitolo 1 fornisce una descrizione degli approcci seguiti dai ricercatori che implementano ambienti software per lo sviluppo e l'esecuzione di sistemi multi-agente. Include anche l'analisi di tre piattaforme esistenti che possono essere usate per creare MAS.

Il Capitolo 2 all'inizio contiene la descrizione del progetto generale, chiamato ARPEGGIO, di cui DCaseLP puó essere considerato un'istanza, successivamente viene spiegato il motivo per cui é nato DCaseLP; viene anche trattato lo strumento software CaseLP, predecessore di DCaseLP.

La piattaforma ad agenti JADE usata nell'ambiente DCaseLP per eseguire il prototitpo di MAS sviluppato, insieme al sistema tuProlog, é descritto nel Capitolo 3. Il sistema tuProlog é brevemente introdotto perché system solo il suo linguaggio é stato integrato nella versione attuale di DCaseLP e lo sviluppatore non é tenuto a conoscere come tale sistema funzioni per poter programmare gli agenti logici.

Nel Capitolo 4 spieghiamo come é stato possibile integrare gli agenti logici in DCaseLP. Piú precisamente, dettagliamo il package Java che abbiamo chiamato tuPInJADE e aggiunto all'ambiente per permettere all'utente di eseguire quelli che definiamo agenti tuProlog, in altre parole agenti che in DCaseLP sono totalmente programmati in Prolog.

Oltre a dimostrare che gli agenti tuProlog sono veramente capaci di ragionare e comunicare tra di loro, nel Capitolo 5 diamo i dettagli di un semplice MAS — contenente agenti tuProlog ed un agente Java — che abbiamo creato per testare la compatibilitá tra gli agenti Java e tuProlog in uno stesso prototipo.

Infine, nelle Conclusioni riassumiamo il lavoro fatto in questa tesi e suggeriamo alcune direzioni per lavori futuri. Prima della bibliografia abbiamo aggiunto tre Appendici.

L'Appendice A contiene il codice dei file che definiscono le classi Java che costituiscono il package risultante dall'integrazione degli agenti logici in DCaseLP.

L'Appendice B contiene l'elenco degli atti comunicativi disponibili e che possono essere inseriti nei messaggi scambiati dagli agenti in JADE.

L'Appendice C contiene la serie di messaggi di errore che possono comparire quando lo sviluppatore fa un uso non corretto dei predicati definiti nella libreria TuJadeLibrary.

Integrating Logical Agents Into DCaseLP

Submitted by Ivana Gungui to
the Department of Computer Science,
University of Genova,
Italy,
as a MSc Thesis.

22nd March 2005

Principal supervisors: Prof. Maurizio Martelli
Doct. Viviana Mascardi

Co-supervisor: Prof. Gianna Reggio

To Gran and Rosy

Contents

Introduction **i**

1 Environments for developing MASs **1**
1.1 Introduction 1
1.2 MASs and concurrency 2
1.3 Mobile agents 3
1.3.1 Intelligent networks 5
1.3.2 Agent-based management of wireless/mobile networks . . 8
1.4 The MadKit Project 9
1.4.1 The Agent/Group/Role model 11
1.5 The Mozart Programming System 14
1.6 The Jack Platform 19
1.7 Conclusions 23

2 The DCaseLP environment **25**
2.1 Introduction 25
2.2 ARPEGGIO 26
2.2.1 The general architecture 28
2.2.2 The proposed methodology 29
2.2.3 The ARPEGGIO partners 30
2.3 DCaseLP 32
2.3.1 DCaseLP's focuses 32
2.3.2 CaseLP 34
2.3.3 DCaseLP's first release 36
2.4 DCaseLP's current release 44
2.5 Conclusions 47

3 The tuProlog and JADE systems **49**
3.1 Introduction 49

3.2 The tuProlog system 50
3.3 The JADE platform 54
3.3.1 FIPA's standards 57
3.3.2 A FIPA-compliant architecture 58
3.3.3 Communication in a FIPA-compliant platform 60
3.3.4 JADE's components 62
3.3.5 Running the JADE platform 63
3.3.6 The main container 65
3.3.7 Communication in JADE 67
3.3.8 Agents behaviours 69
3.4 Conclusions 70

4 The interface between Prolog and DCaseLP 73
4.1 Introduction 73
4.2 The tuPInJADE package 74
4.2.1 The ErrorMsg class 74
4.2.2 The JadeShell42P class 79
4.2.3 The JadeShell42PGui class 83
4.2.4 The TuJadeLibrary class 86
4.3 Conclusions 90

5 Java and tuProlog agents interacting - An example 93
5.1 Introduction 93
5.2 A distributed marketplace scenario 94
5.2.1 The buyer agents 95
5.2.2 The seller agents 106
5.3 Executing the prototype 116
5.3.1 The results 120
5.4 Conclusions 123

Conclusions 125

Appendix A: The tuPInJADE Java package 128

Appendix B: Available performatives in JADE 163

Appendix C: Error messages caused by misuse of predicates defined in TuJadeLibrary 165

Bibliography 170

Introduction

The most mentioned term in this thesis is, without a doubt, the *agent*, therefore it seemed natural to introduce our work with an overview of what this term indicates and why software engineers concentrate on developing software systems composed of one or more of such entities.

Since an *intelligent/autonomous agent* has been, during the last two decades, subject to various interpretations of its meaning in the computing field, it has not been possible to state them all in this introduction.

With this introduction we wish to give the reader a general idea of the subject of this thesis.

Intelligent software agents

Since the 1980s, *software agents* and *multi-agent systems* have grown into what is now one of the most interesting research fields in the computer science community. Agents and agent-based systems are considered as *a new software technology*, the so-called **agent technology**, and can be found as a main topic on the Web.

> *"... agents are here to stay, not least because of their diversity, their wide range of applicability and the broad spectrum of companies investing in them. As we move further and further into the information age, any information-based organisation which does not invest in agent technology may be committing commercial hara-kiri."*
> [Nwa96]

The concept of an (intelligent) *agent* originated from the area of Artificial Intelligence, a very popular field in Computer Science that deals with designing computer systems to behave intelligently. In other words, agent was the term chosen to indicate a program or system constructed to act as similarly as a

human, whose behaviour is considered as intelligent. Nevertheless, the term also gained popularity in other fields of Computer Science, being referred to in different contexts.

Being widely used by many people working in closely related areas, does not automatically mean that there is a single universally accepted definition of the term. There still is an ongoing debate about this subject, but the notion of *autonomy* seems to be considered as a key notion of agents. The main problem is that agents can be applied to a lot of different domains which can require different abilities for each agent.

Many people are of the opinion that the growth of the Internet has certainly helped the wide spread of agents in many application domains. A common definition that one can find on the Internet is:

> *"An (intelligent) agent is a program that gathers information or performs other actions without the immediate presence of a human being. Typically, an agent program — using parameters previously provided to it manually or automatically — searches all or some part of the Internet, collects information the user is interested in, and presents it on a daily or other periodic basis. This is why it is sometimes called a bot".* [Wha]

Such a definition, though, is partial and does not include other aspects on which the agent community has focused attention, nor does it fully adhere to one of the many definitions given in the last decade; it merely describes a typology of agent.

A classical definition of intelligent agent, given by N. R. Jennings, K. Sycara and M. Wooldridge in [JSW98], is:

> "*... a computer system,* ***situated*** *in some environment, that is capable of* ***flexible autonomous*** *actions in order to meet its design objectives.*"

Situatedness means the capability to receive input from sensors in the surrounding environment and to perform actions which change the environment itself in some way.

Flexible autonomy includes the ability to make decisions about what actions to take, according to the circumstances, without any human or other agent getting involved.

There are, indeed, many variants of agent: *autonomous agents*, *mobile agents*, *distributed agents*, *learning agents*, *planning agents*, *data mediation agents*, *personalised visualisation agents*, and many others depending on the application in which they are involved.

In 1997, S. Franklin and A. Graesser proposed the following categorisation of the major classes of autonomous agents, that is the known families of examples, viewed as a natural kinds taxonomy (see Figure 1).

At the root of the tree one can find autonomous agents, while at the level below there are the biological, robotic, and computational agents since it is easy to distinguish between animated organisms, artifacts and abstract concepts. Computational agents are divided into software and artificial life agents — the latter are systems frequently composed of many similar, simple organisms that collectively exhibit life-like behaviours. Software agents can be subclassified into task-specific agents (like email filtering agents), entertainment agents and computer viruses.

Franklin and Graesser also proposed to further classify software agents by considering different subclasses of agents, characterised by certain properties such as, for instance, the ability to communicate, to learn, to plan and so forth; this gives an idea of how many agents there can be.

Figure 1: A Taxonomy for Autonomous Agents [FG97]

As we can see in Figure 1, they considered intelligence implicit in the concept of autonomy. Actually, the terms "*intelligent* agents", "*autonomous* agents" and "*software intelligent* agents" are often interchangeable.

Software agents can also be characterised by behaviours: *reactive*, *deliberative*, *goal-driven*, *adaptive*, *autonomous*, *learning*, *communicative*, *proactive*, *competitive*, and *collaborative* are some examples.

Reacting, in a timely fashion, to events happening in the environment is considered as a *reactive* behaviour. Being able to carry out coordinated actions over time is a *deliberative* behaviour. Acting to obtain a certain target is a *goal-driven* (or *goal-directed*) behaviour.

The ability to respond to changes in the environment is an *adaptive* behaviour. A *learning* behaviour is distinguished by being able to learn from one's own experience and, possibly, from others. A *communicative* or *social* behaviour denotes the capacity to interact, when appropriate, with other artificial agents or humans.

A *proactive* behaviour corresponds to acting in anticipation of future goals (i.e., making plans); it is, really, a mix of goal-driven and reactive behaviours. Being able to compete with other agents, human or not, to obtain shared resources, or an own aim, is a *competitive* behaviour. Finally, being able to act together with other agents, human or not, is a *collaborative* behaviour.

Realising agents possessing all of these behaviours is not a trivial task at all. Some researchers have concentrated their work on realising software structures in which only some of such behaviours appear, leading to particular agent architectures.

Another issue to consider about a software agent is *how long it should run for*. It could run forever or indefinitely, continuously pursuing its goals, but it is also possible for an instance of an agent to terminate once its goals have been met or are no longer needed.

Now, having in mind what we consider an agent, we can speak of a community of software agents or a multi-agent system, in which many independent agents cooperate and/or compete in achieving common goals or, only their own. The community may be limited to pre-planned members, or it could be open-ended and allow arbitrary, external agents to "join". A key concept in a multi-agent system is the ***interaction*** between agents: they must be able to communicate in order to cooperate, coordinate and negotiate when necessary.

Agent-Oriented Software Engineering (AOSE)

Software Engineering is a field of Computer Science that deals with the process of developing a software product — especially large and complex systems

— in an economic and timely manner. It starts from the inception to subsequently pass to the design, the implementation, and finally to the delivery and maintenance of the software product.

In this area, the CASE (**C**omputer-**A**ided **S**oftware **E**ngineering) tools have gained great relevance, being a class of software that automates many of the activities involved in the various cycle phases of the system being built, therefore making a more efficient development. Automated development tools are generally implemented by organisations that want to increase productivity, decrease costs, enhance project controls, and increase product quality.

The traditional software engineering process model suggests organising the software development in successive phases. In the first phase, the system's requirements are analysed, then the fundamental structure is designed by decomposition into components that are easy to model. Such components are implemented and tested individually before being integrated and tested as a whole. The last phase is dedicated to the maintenance of the installed software.

This model — the so-called *waterfall model* proposed in 1970 by W. W. Royce [Roy70] — has some drawbacks, above all a missing feedback to prior phases. During the analysis or the design the developer can introduce errors that may not be discovered until the testing phase, in such a case correcting them would be at a high cost.

Improved or new models have already emerged, many years ago, among them *prototyping* and the *spiral model* — the latter proposed by B.Boehm in 1986 [Boe86] —, they both foster an incremental approach: creating a thin but executable prototype that reveals deficiencies at an early stage, and from which it is possible to gather information to improve the prototype, verify it again, and so on.

This evolutionary *prototyping cycle* is repeated until the software is finished or a shortage in time or money allows delivery of at least one version with limited capabilities.

Very well known, not only in the Computing but also in the Business community, is the Object-Oriented approach to software engineering that has led to the OOSE (**O**bject-**O**riented **S**oftware **E**ngineering) subfield.

Object-orientation models a system as objects that interact and thus trigger certain patterns of behaviour: persons, real world subjects, or imaginary things may all be objects. Objects incorporate both structure and behaviour: the structure represents the object's data and the static properties of an object, while behaviour represents the dynamic properties of an object.

The most discussed but not most important property of objects is inher-

itance: in computer science it works similar to that in nature. As children inherit structure (e.g. skeleton, muscles, ...) and behaviour (e.g. habits, preferences, ...) from their parents, in the same way objects inherit structure (e.g. methods, attributes, ...) and behaviour (e.g. a state machine diagram) from their parent objects. The inheritance mechanism allows money and time saving when developing new or evolved products.

Software agents are the next evolutionary step beyond objects and programs. There are many reasons for the current intensity of interest, but certainly one of the most important is that the concept of an agent as an autonomous system, capable of interacting with other agents in order to satisfy its design objectives, is a natural one for software designers.

AOSE (**A**gent-**O**riented **S**oftware **E**ngineering) is one of the most recent contributions to the field of Software Engineering: analysing, designing and implementing software as a collection of interacting and autonomous agents represents a challenging task for software engineering.

It has several benefits compared to existing software development approaches, especially the ability to let agents represent high-level abstractions of active entities in a system. *Software Engineering with Agents*, *Agent-Based Software Engineering*, ***M**ulti-**a**gent **S**ystems **E**ngineering* (**MaSE**) and AOSE can be considered semantically equivalent terms.

Objects tend to be passive, with only one object active at a time, and "hidden" inside programs. Programs tend to be relatively static and designing a distributed computation with an indeterminate number of collaborating programs is not an easy task. Agents are dynamic because they can exercise some degree of activity, rather than passively providing services.

Software agents offer opportunities for parallelism and inter-program collaboration not only on a large scale, but also in an easy, robust, reliable, flexible, and secure manner. Another relevant difference between objects and agents is: the former are controlled from the outside (*whitebox control*), while the latter have autonomous behaviour which cannot be directly controlled from the outside (*blackbox control*).

Logic-based languages are more suitable for model agents than object-oriented languages because: not only can they express the active behaviour of agents in a concise and simple way, but model agents that react to an event — logical implications of the form "if the event E took place then something becomes true " can also be used for this purpose — and to reason about sophisticated conversations.

The agent technology plays a central role in the development of complex

distributed systems, networked information systems, and computer interfaces. A MAS, formed by several autonomous acting entities working together (or competing) to reach a given goal, can naturally represent not only decentralisation, multiple perspectives and competing interests, but also information systems that are distributed, open and heterogeneous.

So, the main purpose of Agent-Oriented Software Engineering is to create methodologies and tools that enable inexpensive development and maintenance of agent-based software. In addition, the software should be flexible, easy-to-use, scalable and of high quality. In other words, its research issue is quite similar to the one of other branches of software engineering, as for example object-oriented software engineering, but it focuses on improving coordination, cooperation, communication and artificial intelligence in agents and agent systems.

Agent Communication Languages

A side issue, nevertheless not less important, is the communication among agents: two well-known standards, called "agent communication languages" or **ACLs**, that have been defined for message-passing between agents are **KQML** (**K**nowledge **Q**uery and **M**anipulation **L**anguage [FLM95]) and FIPA's **ACL** [ACL].

FIPA is an international non-profit organisation that, since 1996, has produced standards for the interaction of heterogeneous software agents and has promoted generic agent technologies. Both KQML and FIPA's ACL allow the exchanging of messages that have a declarative content and do not take into account the internal structure of the agent sender/receiver.

An agent ontology is a formal specification that defines the vocabulary of terms known by the agent and, possibly, a specification of the meaning of such terms. ACLs implement a communication protocol that is unaware both of the content language used and of the ontology specification mechanism. Even though there is not a universally accepted standard ACL, it is interesting to note that all major proposals share the assumption that agent communication should be dealt with in terms of *speech acts*.

The notion of a speech act comes from the philosophy of language [Aus62, Sea69]. Speech acts have been adopted as the base of agent communication by the proposers of KQML as well as by FIPA.

The speech act theory is a powerful, deep and remarkably comprehensive theory of language semantics so, regarding agent communication as a form of

action, makes it easy and natural to embed a communicative component in an agents architecture.

An ACL only specifies the basic intention of a message and includes the content as a separate element: such intention is explicitly specified in the message exchanged, stating one of a small set of standard *speech acts*. In other words, these languages derive their primitives from the linguistic theory of *speech acts* and distinguish various "performatives", depending on the intent of the agent that performes the so-called speech act.

The list of standard performatives that they allow gives the agents the chance to carry out every significant kind of conversation.

The content of an ACL message received by an agent can be understood only if it shares a common *ontology* with the agent that sent the message.

Due to the lack of universally accepted standards, a development tool for supporting AOSE is a software providing "building blocks" for some agent architectures and support for one of the basic ACLs. There is another subject of interest in the agent community: *agent architectures*. An agent architecture is the particular methodology used to build the agent and specifies how it can be decomposed into a set of component modules and how these modules should interact. Any reader interested in agent architectures — reactive, BDI, layered and so forth — can visit, for example, the following links:

- "*A Survey of Cognitive and Agent Architectures*" (a work carried out in 1994 at the University of Michigan, Department of Electrical Engineering and Computer Science) - *http://ai.eecs.umich.edu/cogarch0/* ;
- "*AGENT ARCHITECTURES*" contains many references - *http://www.ecs.soton.ac.uk/~mml/absd/chap2.html*;
- "*Agent Architectures*" - *http://www.csc.liv.ac.uk/ mjw/pubs/ker95/section3_3.html*;
- "*Roadmap - Where we go next!*" - *http://www7.in.tum.de/~weissg/Lehre/AosmSS02/aosm-ss02-2.slides.pdf* .

The aim of the thesis

The aim of this thesis is:

> *To make a step forward towards the* ***engineering of heterogeneous agents and multilingual agent systems*** *by integrating "****logical agents****" into the existing prototyping environment DCaseLP.*

DCaseLP (***D****istributed* ***CaseLP***) [AMMM02] is a CASE tool that originated as the successor to CaseLP (***C****omplex* ***a****pplication* ***s****pecification* ***e****nvironment based on* ***L****ogic* ***P****rogramming*) [MMZ99, BDM+99], both developed by the *Logic Programming Group* at the Department of Computer Science of the University of Genova in Italy.

CaseLP offers a prototyping method and a set of tools and languages to support the realisation of prototypes of complex applications modelled as MASs.

By using CaseLP, the developer has the possibility to choose the most suitable language for specifying different aspects of the MAS. The set of available languages includes $\mathcal{E}_{hhf}$ [Del97] and **HEMASL** (**HE**terogeneous **M**ulti-**A**gent **S**ystems **L**anguage) [Mar99, MMMZ00a, MMMZ00b].

$\mathcal{E}_{hhf}$ is an executable linear logic programming language very useful not only for describing the overall system at an abstract level, but also for describing, at a lower level, the internal components of the agents and their interactions.

HEMASL, on the other hand, is an imperative meta-language for specifying different agent architectures and the configuration of the MAS.

$\mathcal{E}_{hhf}$ specifications can be tested and/or verified, and HEMASL descriptions can be translated in sets of clauses.

Semi-automatic compilers that translate the specifications given in these languages into SICStus Prolog [SIC] are part of the environment. SICStus Prolog (extended with ad-hoc communication primitives) is, on the contrary, the unique implementation language available.

So, choosing SICStus Prolog as the implementation language for prototypes, presented many advantages that were, briefly speaking, due to the following reasons:

- *logic-based languages are very suitable for MAS prototyping*, since interactive development and testing of agent-based applications are easier to

carry out. The specifications can be straightforwardly executed, thus directly offering the prototype, and properties of the code can be formally verified.

- the linear logic-based and the rule-based specification languages provided to describe the MAS are *easy to translate into a logic programming language*.

On the other hand, though, such choice has caused some drawbacks: the most immediate is a low degree of portability, since logic programming languages are not much supported in commercial applications, and not less relevant is that most of the programmers find the logic paradigm not easy to use and lacking concurrency support.

DCaseLP came into being with the aim of extending the set of implementation languages of CaseLP and becoming a *complete multilingual AOSE prototyping tool*. The first release of DCaseLP began the process of achieving this aim by proposing *two implementation languages*: the **Java**^{TM} [Mic] language and the **Jess** [FH] expert system language.

Although the first release of DCaseLP overcame many limitations of CaseLP, it could not offer all the functionalities existing in its predecessor, because it did not include the *Prolog-based code and instruments already developed for CaseLP*.

At this point, the need of integrating "**logical agents**" into DCaseLP arose, in order for the environment to provide the MAS developer with the logic-based tools that were, indeed, available in CaseLP and not in the first version of DCaseLP. The logical agents must show the following features:

- their behaviour is fully specified by a Prolog-like language;
- they are able to "reason", in the sense that they can perform what is called a *resolution* in logic programming;
- they can communicate with other agents developed with DCaseLP, namely those implemented in Java and Jess.

The aim of this thesis is thus to integrate these logical agents into DCaseLP and test their *interoperability* with other type of agents developed with DCaseLP.

Structure of the thesis

This thesis is structured as follows. Chapter 1 gives an overview of approaches followed by researchers that implement software environments for developing and executing multi-agent systems. It includes the analysis of three available platforms that can be used to build MASs.

Chapter 2 initially deals with the description of the general project, ARPEGGIO, of which DCaseLP can be viewed as an instance, then it explains the reason why DCaseLP has come into being; CaseLP, the software tool that is the predecessor of DCaseLP, is also overviewed.

The agent platform JADE used in the DCaseLP environment in order to execute the developed MAS prototype, together with the tuProlog system, is described in Chapter 3. The tuProlog system is briefly introduced since only its language has been integrated into the current version of DCaseLP and it is not necessary for the developer to be aware of the functioning of that system in order to program the logical agents.

In Chapter 4 we explain how we have carried out the integration of the logical agents into DCaseLP. More precisely, we detail the Java package tuPInJADE that we have added to the environment in order to allow the user to run what we call tuProlog agents, namely agents that are totally programmed in Prolog, in DCaseLP.

Besides demonstrating that the tuProlog agents are really able to reason and communicate with each other, in Chapter 5 we give details of the simple MAS — containing both tuProlog agents and a Java agent — that we have created to test the compatibility in the same prototype of Java agents with tuProlog agents.

Finally, in Conclusions we summarise the work of this thesis and suggests some directions of future work. Before the Bibliography we have added three Appendices.

Appendix A contains the code of the files that define the Java classes that constitute the package resulting from the integration of logical agents into DCaseLP.

Appendix B contains the list of available performatives that can be inserted into messages exchanged by agents in JADE.

Appendix C contains the series of error messages that can be displayed when the predicates defined in the TuJadeLibrary are misused by the developer.

Chapter 1

Environments for developing MASs

1.1 Introduction

The real strength of agents today is based not only on them as being part of a community, a ***multi-agent system (MAS)***, but also on the negotiation mechanisms and coordination facilities that they imply. A multi-agent system is a dynamic federation of software agents, coupled by common environments, goals or plans, which cooperate with each other or coordinate their actions.

Dividing functionality among many agents provides modularity, flexibility, modifiability, and extensibility.

MASs provide a better *support to applications requiring distributed computing*, since agents can be designed as fine-grained autonomous components acting in parallel.

However, in order to provide a MAS-building toolkit, an appropriate environment must be available: an infrastructure that, besides specifying communication and interaction protocols, is open and not centralised, and can contain autonomous, adaptive and cooperative agents.

Before describing our software environment for prototyping Multi-Agent Systems, DCaseLP [Mig02, AMMM02], we want to give general insights into other relevant issues that have attracted the agent community in the last years, and briefly present three software tools for creating MASs and characterise their main features. Those that we present in this chapter have been chosen among many available in the Internet.

These software tools are more likely to be considered ***Agent Frameworks***

than real AOSE (**A**gent-**O**riented **S**oftware **E**ngineering) "development tools", since they are programming tools for constructing agents or MASs, and do not present a real AOSE methodology to guide the developer during the different phases that lead to the final software system.

As we will stress in the next chapters, we have concentrated our interest on tools that support ***rapid prototyping*** since we believe it offers a means by which to emphasise and show the essential features of a proposed system and, among all, really facilitate the work of software engineers.

Rapid prototyping can be summarised as promoting *early experimentation* with *alternative design choices*, thus allowing the software developers to pursue different solutions without having to deal with efficiency concerns. This is the reason why prototyping has a key role in the software engineering methodology.

Nevertheless, in this chapter, we want to show other "views" of MASs that have been proposed by agent researchers and still concern agent technology, especially those known as *mobile agency* and *concurrent systems modelled as MASs*.

1.2 MASs and concurrency

In the agent community has also come into being a particular characterisation of MASs according to which they are regarded as a *special kind of concurrent systems*, where the role previously played by processes is now played by agents.

MASs imply a high degree of concurrency, while most of the conventional programming languages — and, therefore, the agent frameworks built upon them — have only limited support for concurrent systems.

Even the common thread-based model imposes many restrictions on the agent developer, leading to systems which do not provide maximum concurrency due to technical limitations. Additionally, a lot of syntactical or management overhead is needed when implementing concurrent systems using such techniques, which does not bring into focus the view on the essential concurrency and synchronisation concepts.

MASs differ from ordinary concurrent applications in the type of interactions that exist among their components. The interactions inside a MAS have a higher level of sophistication than those inside concurrent systems: the reader might just, for instance, imagine the interactions that take place during an electronic auction or a negotiation.

Moreover, the components of MASs are characterised by reasoning capa-

bilities that "decide" their behaviour according to interactions and any information on the environment. The programming languages and systems for prototyping concurrent applications are not directly applicable to MASs because the interactions among components in ordinary concurrent systems are too simple and mostly exchange information rather than more complex knowledge representations.

Another noticeable difference between them is the lower level of abstraction that characterises the development of a concurrent system: more concern is usually given to the underlying protocols and message-passing services. As an example of environment supporting the development of "concurrent MASs", we show the Mozart specifically designed to create highly concurrent applications, including MASs.

1.3 Mobile agents

The increasing popularity of the Internet and, as a consequence, of search engines, distributed computing, and e-commerce, has led to more requests for excellent services from it. To achieve this aim, the Internet as been virtualised as a *whole*, where *software agents can freely move over the entire network*, leading to what are now called ***mobile agents***.

The interest in mobile agents is quite easily explained: mobile agents wandering around the Internet can search for information, find great deals on goods and services on behalf of the users, and interact with other agents that also wander around networks (and meet in some place) or remain bound to a particular machine.

Thus, in recent years significant research and development have been carried out in the area of *mobile agency*, so many mobile agent architectures are now available.

Some architectures use custom scripting languages, while others use Java which helps to solve the problem of portability.

Most of the available distributed systems today are based on the client/server model, which is implemented through some form of remote procedure calls, or remote objects in the case of the object-based model. The client/server paradigm, however, has some limitations.

A *mobile agent* is a term used to indicate a software entity that can travel freely among the hosts in a heterogeneous network, interact with other agents or share resources.

A mobile agent is a *special agent*, in the sense that, besides autonomy, reactivity, proactivity and reasoning (considered basic characteristics), expresses *mobility* as well, which means it *moves independently from computer to computer in the network* and completes the task assigned to it by the user.

The mobility shown in a heterogeneous network environment also means the capacity of the software agent to dynamically adapt to the changing network environment in distributed computing. Additionally, it must be able to find other agents, since they might appear, disappear, or move at any time, and must be able to interact with them.

Mobile agents are an approach that is becoming *a new paradigm for distributed computing* and are considered very interesting since reliability of the network connection is not crucial in their case.

Mobile agents can be thought of as resulting from combining software agent technology and distributed computing technology. What changes substantially is the traditional network computing model since mobile agents have nothing to do with Remote Procedure Calls (RPCs).

While mobile agents can move continuously from one node to another, and travel according to their own needs and choices, that is not true for the ordinary migration processes that, in fact, cannot decide themselves where/when to go.

Additionally, mobile agents are different from Java applets, since applets can only travel one-way from server to client, while mobile agents can move bi-directionally: from server to client and vice versa. A mobile agent-based application consists of a group of mobile agents: each mobile agent moves to the node that has resources for computing, according to its own goal and environment, and may cooperate with other mobile agents during the computation.

The whole computing process can be divided into several steps: after each single step is completed, the mobile agent will choose its next action independently and will continue to do so until all the tasks have been carried out.

The ordinary distributed computing model typically uses RPC, process immigration and Client/Server architecture, but all these computing models have limitations: among all, the most important is that all the nodes involved in computing have to be available in the network at the same time during the interaction and, if a required resource is not accessible, the whole computing process will fail.

The mobile agent technology allows communication among computers, so the distributed computing is not any more a service request made by a computer to another computer, but it also provides executable computing processes to other machines.

Moreover, data is not the only information exchanged over the network, since even computing processes and the state of their computing can be transferred.

The mobile agent technology helps reduce network load significantly. Distributed systems often rely on communication protocols involving multiple interactions to accomplish a given task. The result is a lot of network traffic.

The technical difficulties that can arise with this technology are mainly:

- ***portability*** - since Internet is a heterogeneous environment, a mobile agent must be able to run on many different platforms, so it must be written in a machine independent language;

- ***security*** - agents are executed on remote machines, so there must be some mechanism that guarantees they will not do any harm to the servers they are executed on, and also they will not be damaged by malicious agents while roaming the network.

Mobile agent technology has already moved from the stage of theory to practical usage, and some mature development platforms and execution environments keep coming out. Theoretically, mobile agents can be written in any language (such as C/C++, Java, Perl, Tcl and Python) and can be executed on any computer.

Since mobile agents must support different software and hardware environments, the development platform must be independent of specific languages. Java is an ideal language to develop mobile agents. After the compilation, Java binary code can be executed by any Java interpreter, which is cross-platform.

Mobile agent technology has been studied for many years, but the real applicable mobile agent system did not become available until 1996.

1.3.1 Intelligent networks

It is the ability to communicate, coordinate, and cooperate that makes agents and MASs a worthwhile metaphor in computing and that makes them attractive when it comes to dealing with some of the requirements in *next-generation telecommunications systems*.

In order to realise the potential created by technological innovations, some important requirements for the next generation of networks need to be satisfied:

- *ubiquitous personal assistance and reachability anywhere at anytime*, so users will have at their disposal added-value and help in navigating through the massive content and service offerings;
- *intelligent user interfaces over multiple devices*, like mobile phones, next generation Personal Digital Assistants (PDAs), web pads, and PCs can support the users in exploiting the potential of multimedia, large bandwidth access to content and services;
- *more security mechanisms* will have to be incorporated into intelligent telecommunication infrastructures, since security issues will become more critical, due to the fact that more devices carry personal information (such as user profiles) and computers are turning into personal assistants that carry out legally relevant transactions (such as purchasing or signing contracts);
- *new ways of designing and managing networks* are needed since networks are becoming more complex and dynamic: support for scalability through self-organisation and for hand-over between different networks is lacking.

Mobile agents distribute code to proxies in the network as well as to mobile devices: thus, the number of network transactions necessary to provide a service can be reduced by local processing as well as local service customisation.

This, in turn, might lead to less bandwidth consumption and lower latency. Different *real* ***mobile agent platforms*** have already been proposed: they are software intended to manage the mobility and task execution of the agents.

Exmaples of agent platforms are, for instance, MOLE [MOL], IBM Aglets [IBM], Objectspace Voyager [RS], the OMG-MASIF initiative [OMGa], Grasshopper [IKV], Semoa [IGD] and Swarm [Gro].

Furthermore, the ***ACTS*** (**A**dvanced **C**ommunications **T**echnologies and **S**ervices) EU program [ACT] has featured a cluster of agent-based telecommunications projects, named ***CLIMATE*** (**Cl**uster for **I**ntelligent **M**obile **A**gents for **T**elecommunication **E**nvironments). Each single project in CLIMATE has been organised according to the following topics:

- ***intelligent networks and mobility***
- ***communication and management***
- ***agent systems***

- ***agent platforms***

The CLIMATE projects involve strong industrial participation, including Vodafone/Mannesmann, Motorola, Siemens, Sony, DTAG, NEC, Alcatel, Hitachi, British Telecom, Swisscom, and Philips.

The desired role of agent technology in telecommunications is to be a key vehicle for:

- achieving enriched, higher level communication;
- enabling more intelligence in providing services and network management (for instance, by personalising and integrating different services to value-added services and negotiation of quality of service);
- dealing with the increasing amount of information and functions, and
- allow self-organising networks.

In order to be able to achieve these goals, agents need to communicate to discover their peers, to negotiate and to cooperate in open environments where everybody can add their contribution when and how it seems appropriate.

Most importantly, agent systems will have to interface with a variety of existing and upcoming developments and standards at the underlying network systems level, including support for a wide range of devices.

Therefore, the network systems level needs to offer some degree of programmability. Programmable networking aims at opening low-level access to network elements (routers, switches, base stations) by defining appropriate interfaces accessable by various entities (protocols, agents), which offer advanced and customisable services.

This concept can then be extended with (agent-based) mechanisms for distributing and executing code which programs the interfaces on behalf of individual applications (including those running on end systems). The new paradigm has been called *active networking*.

Some researchers are convinced that agent-based active networking could influence the design of the next generation of wireless packet networks.

1.3.2 Agent-based management of wireless/mobile networks

Mobile agents appear to be an interesting combination of current research on network management and agent-related research. On the one hand, the research on network management is focused on finding new ways to overcome the limitations of current client-server technology, on the other, research on mobile agents and peer computing provides technologies and architectures to enable decentral, peer-to-peer communication.

In the last years, distributed programming has become more and more popular, especially due to the widespread of the Internet that provides the first steps towards a global infrastructure for distributed applications: a global namespace (URLs) and a global communications protocol (TCP/IP). Platforms based on the Java language and on the CORBA standard take advantage of this infrastructure and have become widely-used.

Distributed programming is still a hard task: writing efficient, open, and robust distributed applications remains much harder than writing centralised applications. Moreover, if the distributed applications also have to guarantee security, then the difficulty increases.

The abstractions offered by Java and CORBA, for example the notion of distributed object, provide only rudimentary help, since the programmer must still strongly concern about distribution and fault-tolerance.
The main advantages of agent technology are the following:

- ***Distribution of management code***: instead of carrying large amounts of management data to stations via the network, mobile agents can transport network management code to the Simple Network Management Protocol (SNMP) agents. This saves bandwidth, reduces bottlenecks, and makes the architecture more scalable, a very important feature since in a wireless network the resources are a few.

- ***Decentralisation***: mobile agents can effectively decentralise network management functions. They can proactively and autonomously carry out administration tasks such as installing and upgrading software, or periodically monitor the network. Decentralisation is more interesting in a network scenario where the traffic needed for management must be minimised.

- ***Dynamically changing network policies***: as networks dynamically change, the rules underlying network management (the so-called *policies*)

need to be changed from time to time. In current network management systems, this is done following a complete "rewrite, compile, run" cycle; using agents, these changes can be done dynamically and incrementally, by replacing agents or agent functions one at a time.

- ***Network monitoring***: mobile agents are beneficial for surveillance of SNMP variables and long-term monitoring of network elements. This is important for wireless networks as the node configuration might change over time (considering, for example, ad-hoc networks or a variable number of users in a single cell of a cellular network).

- ***Data collection***: agents can search, collect and filter network data. They can be used to process data-intensive requests from network elements. In this case, the agent acts as a "smart query" that visits the data and performs the necessary computation locally, instead of passing large chunks of data over the network.

- ***Reactiveness***: agents can quickly react to local events, such as the breakdown of a link.

- ***Robustness***: agents can, at least to a degree, perform their tasks even if parts of the network are not temporarily reachable. This makes them particularly valuable for mobile computing, where links are expensive and unstable.

1.4 The MadKit Project

MadKit (a **M**ulti-**A**gent **D**evelopment **K**it) [GF00] is a highly customisable, scalable, generic multi-agent distributed platform for developing and executing distributed applications.

This platform is based on the **AALAADIN project** [FG98] (designed by the same authors of MadKit), namely a meta-model used to structure MASs, considered as characterised by different forms of artificial organisations. This meta-model is the means by which MadKit provides agent heterogeneity in languages, applications and architectures.

The main goal of the platform is to constitute a *foundation layer for various agent models*, as well as offering basic services completely extensible and replaceable. It is written in Java and based on an *organisational-*

oriented paradigm, the ***AGR*** (**A**gent/**G**roup/**R**ole) paradigm, according to which agents belong to groups and play roles (see Subsection 1.4.1).

In business environments, it is often needed to consider the behaviour of the global system and incorporate collective characteristics designed from a top-down perspective; in other words, the organisational design must be able to consider characteristics of the organisation, not easy to gather from a model taking into account only the behaviours of the individual agents.

Multi-agent systems, or *agent societies*, represent the interactions between agents and are as such the virtual counterpart of real-life societies and organisations. The proposed multi-agent paradigm uses agents, groups and roles as the basic standpoint for building complex applications, but the developer can create the MAS without taking into account the organisations of agents.

MASs are described on the basis of social structures built from groups and roles, independently from the actual nature of the agents, so the internal architecture of agents is totally up to the developer that is free to implement whichever agent architecture he/she prefers.

MadKit general architecture is based on ***micro-kernels*** which provide only the basic facilities: messaging to *local* agents, management of groups and roles, launching and killing of agents. All the other features — as distant messages, display, monitoring and control of agents — are performed by agents.

The underlying mechanism that allows MadKit to implement distributed applications does not rely on remote access techniques as RMI or CORBA, but still allows efficient communication. The communication inside MadKit is based on a peer-to-peer mechanism so, together with the simplicity of the platform, it allows the users to quickly develop distributed applications using multi-agent principles.

MadKit provides a full set of management facilities/agents for launching, displaying, developing and monitoring agents and organisations, including message passing, agent lifecycle management and distribution.

The *micro-kernels* are intended to incorporate a number of key facilities that allow the efficient deployment of agent toolkits.

Agents in MadKit may be programmed in different languages: Java, Scheme (Kawa), Jess (a rule-based engine), BeanShell (interpreted Java), Pyhton (Jython) or any script language may be easily added.

MadKit can also be used as a GUI-oriented platform or as a server, and has an easy to use interface, from which it is possible to launch, display and monitor agents.

MadKit is also provided with a set of "containers" (intending execution

environments for running applications), so it can run in different contexts: as a development environment or as an embedded tool for applications (in JSP, Applet, Console mode, GUI desktop, etc.). Since the platform is implemented in Java, it is a "cross-platform", meaning that it can be run on Win XX, Linux, Mac Os, and other Unix based operating systems that provide a Java interpreter.

MadKit is a free software based on the GPL/LGPL license; the current version is 4.0.9 (Enchanted Lands) and has been released January 2005.

1.4.1 The Agent/Group/Role model

The AALAADIN model is a formal model oriented to software design, implemented and used in the MADKIT tool. An *organisation* is considered as a framework for *activity* and *interaction* that is defined through groups, roles and their relationships. By avoiding an agent-oriented viewpoint, an organisation is regarded as a structural relationship between a collection of agents.

The model is not a static description of an agent organisation, in fact it allows to define rules to specify the dynamics of the organisation. It is based on *three core concepts* that are used to describe and model social and spatial structures of a real system:

Agent An *agent* is only specified as an *active communicating entity* which plays roles within groups. The designer will choose the most adequate internal architecture for the agent, since it is not considered by this model.

Role A *role* is an abstraction representing what function/service or identification the agent has as a member of a group. Each agent can play multiple roles: each role is local to a group. The candidate agent must request to handle a role in a group, and this role is not necessarily assigned to it. Roles, therefore, define *abstract communication schemes*.

Group A *group* is defined as a set of agents that can interact. One group cannot contain other groups, but groups can overlap since an agent can belong to one or more groups. When necessary, a member of a group is chosen as the "representative" role of the group and acts as the proxy for the whole group in the external world. A group has the purpose of describing collective structures by means of *behaviour types* that are represented by roles. Agents that play

the same collective behaviour type differ according to their individual features.

The particular mechanism for role access within a group is not defined, in order to allow more flexibility.

Thus, an organisation can be described only on the basis of its structure, namely by the way groups and roles are "arranged to form a whole", without being concerned about the way agents actually behave. This implies MASs being analysed from the "outside", as a set of interaction modes, and the specific architecture of agents is purposely not addressed.

Moreover, groups are dynamic structures since agents can enter/leave them by acquiring/resigning a role, and they represent organisational levels, while roles represent functions within these levels.

The aim of the Agent-Group-Role formalism is to "resolve" the inherent duality between individual and collective levels of a system.

This approach supports heterogeneous languages, applications and architectures in the design of complex multi-agent software.

Support to MAS development in MadKit

MadKit is built from a set of Java packages that implement the agent kernel, the various libraries of messages and agents, including a graphical development environment and standard agent models.

The generic agent is a class defining a basic abstract life-cycle — what to do upon activation, execution, and termination. A subclass extending that agent has the aim of adding support for concurrent, thread-based execution, which is the natural model for a collaborative or cognitive agent.

Additional subclasses implement synchronous execution through an external scheduler, focused on reactive or hybrid architectures. Communication is achieved through asynchronous message passing: a primitive is used to directly send a message to an agent identified by its unique identifier. There also is a higher-level version that sends or broadcasts to one/all agents having a given role in a specific group. Groups and roles requests and actions are also defined at this level.

The agent developer is completely free to define the agent behaviour, but the organisational view will be always available to structure the MAS. Various specific agent models have been built upon this generic architecture by the developers of MadKit: Jess[1] agents, reactive systems, componential architec-

[1] an expert system shell and scripting language written in Java and based on the Clips

tures.

MadKit provides a set of tools which are useful to the developer of multi-agent applications, like the following:

- **System agents**: are the main tools that a MadKit developer uses to explore, launch, visualise and trace agents.
- **Communicator**: an agent which allows to build distributed applications without being concerned about distribution (in the last version it is substituted by the more flexible **NetCom**).
- **JSynEdit**: an "agentified" version of an editor which has been integrated into MadKit and may be used to edit and launch Java and Scripted agents.
- **SEdit**: is both an editor and animator of diagrams that can be used to view and manipulate information represented as graphs.
- **PluginManager**: can be used to install, update and configure MadKit as needed.

Finally, a "graphic shell" called the *MadKit desktop* or *desktop* for short, launches the kernel and loads the interfaces for the various agents managing them in a global GUI.

Multilinguality in MadKit

It is possible to program MadKit agents in several languages: Java, Python, Scheme (Kawa), BeanShell and Jess. More languages will be added in the future, but it is actually quite straightforward to add a new scripting language in MadKit (the reader is referred to the **MadKit Development Guide** available online at the url: *http://www.madkit.org/madkit/doc/devguide/devguide.html*).

Java is the main language of MadKit because the platform has been programmed in Java. However, because Java is a compiled language, it is somehow more difficult to develop agents in Java because the code needs to be compiled, put into a Jar, placed in a specific location, and then MadKit has to be restarted to execute the agents.

Conversely, if the developer uses one of the available scripting languages — for instance, Python, Scheme (Kawa), BeanShell and Jess —, the agent can

rule system

quickly be executed. The only drawback is that Java agents are more efficient. So, it is up to the developer to choose which of the available languages to use for the agents.

Support to AOSE in MadKit

One of the software tools provided by MadKit is ***SEdit*** that stands for **S**tructure **Edit**or. This tool allows the design and animation of *structured diagrams*. A diagram contains *nodes* and *arrows*, that are called *elements*: an arrow connects two nodes.

A diagram can be compared to a *directed graph* in graph theory. Diagrams are categorised into types called *formalisms*: in SEdit a formalism describes both the structural aspect (the types of nodes and arrows, their relationship and their behaviour) and the graphical aspect (the way nodes and arrows are graphically represented on the editor) of diagrams.

There is a set of pre-defined formalisms, but users can describe their own formalism. A formalism is a model definition that represents domain-specific nodes and links (for instance, UML models, Petri nets, logical gates, etc.).

The pre-defined formalisms may be directly used or exploited for defining more specific formalisms. Some predefined formalisms are:

- **Automaton**: it allows the user to describe and simulate finite state automata.
- **Petri**: can be used to describe and simulate Petri nets (also coloured Petri nets).
- **Bric**: it allows to build and simulate the behaviour of MadKit agents using an extension of coloured Petri nets.

Diagrams may be displayed, arranged and modified in various ways using the editor diagram. In order to animate the diagrams, the user must associate Java classes to arrows and nodes and to the diagram (seen as a whole).

1.5 The Mozart Programming System

The ***Mozart Programming System*** [Con] is an advanced *development platform for intelligent, distributed applications*. It has come into being after a long-term effort of the **Mozart Consortium**, a research collaboration

between the *German Research Center for Artificial Intelligence* (DFKI) in Germany, the *Swedish Institute of Computer Science* (SICS) in Sweden, the *Université Catholique de Louvain* (UCL) in Belgium, and *the Universität des Saarlandes* (UdS) in Germany. The project started in 1991, while the work on distribution started in 1995. The first public release was in January 1999 (*http://www.mozart-oz.org*).

The developers of Mozart, whose current release is 1.3.1 dated July 2004, have dedicated their research to programming language design and implementation, constraint-based inference, distributed computing, and human-computer interfaces.

It has an *interactive incremental development environment* runnable on Unix, Windows, MacOS X and other operating systems. The full source code of Mozart is free of charge and all use is allowed, including commercial use; it provides support in two areas: open distributed computing and constraint-based inference.

It is a *platform* for developing both general-purpose distributed applications and applications for hard problems requiring sophisticated optimisation and inferencing abilities. Many applications have been developed with Mozart, including sophisticated collaborative tools, MASs and digital assistants, as well as applications in natural language understanding and knowledge representation, in scheduling and time-tabling.

The Mozart system implements **Oz 3**, the latest in the Oz family of multi-paradigm languages based on the concurrent constraint model. Oz 3 is almost completely upward compatible with its predecessor Oz 2.

Oz 3 is a concurrent language since users can dynamically create any number of sequential threads that can interact with each other. However, it differs from conventional concurrent languages in the fact that each Oz thread is a dataflow thread, so an Oz statement is executed only when all real dataflow dependencies on the variables involved are resolved.

In Mozart, a process belonging to the operating system is called *site*. So, a distributed program is a program partitioned between sites that are not necessarily running on the same machine. The Mozart system supports network-transparent distribution of Oz computations, meaning that multiple Oz sites can connect together and automatically behave like a single Oz computation, sharing variables, objects, classes, and procedures.

Sites disconnect automatically when references between entities on different sites stop existing.

The primary aim of Mozart is to separate the fundamental aspects of pro-

gramming a distributed system: application functionality, distribution structure, fault tolerance, security, and open computing.

The current Mozart release completely separates application functionality from distribution structure, also providing primitives for fault-tolerance, open computing, and partial support for security.

The **OPI** (**O**z **P**rogramming **I**nterface) is the primary tool for interaction with the Mozart development system: it offers special support for editing Oz code, running Mozart as a sub-process, and interacting with the development tools provided by Mozart.

Essentially all the distribution features of Mozart are given by four modules:

- ***Connection***: this module provides the basic mechanism (known as *tickets*) for active applications to connect with each other.
- ***Remote***: it is used by an active application to create a new site, that is a local or remote operating system process, and connect with it. The site may be on the same machine or a remote machine.
- ***Pickle***: it is exploited by an application to store and retrieve arbitrary stateless data from files and URLs.
- ***Fault***: it includes the basic primitives for fault detection and handling.

The first three modules are extremely simple to use, since they have just a few basic operations. For example, *Connection* has just two basic operations: offering a ticket and collecting a ticket. The fourth module, *Fault*, is the one supporting the creation of fault-tolerant abstractions.

Recent projects involving the use of Mozart and concerning problems deriving from the real world can be found listed at the following url *http://www.mozart-oz.org/archive.cgi*. Mozart and Oz are being used in research, education, and industry in many areas, including distributed programming, software engineering, programming languages, constraint programming, simulation, graphical user interfaces, security, and artificial intelligence.

Support to MAS development in Mozart

By combining concurrent and distributed programming with logical constraint-based inference, Oz becomes a candidate for developing multi-agent systems. The developer implementing distribution must not be concerned with details regarding the underlying network, that is open and fault-tolerant. It is a

platform characterised by an efficient implementation, it is portable, open source, has simple formal semantics and flexibility (with few limits).
The aspects that make it a candidate for developing agent applications are:

- ***concurrency***;
- ***inferencing***;
- ***distribution***;
- ***graphical user interfaces***;
- ***components and agents***;
- ***formal semantics***.

The system supports distributed and networked applications: it is possible to connect Oz computations located on different sites, resulting in *a single network-transparent computation*.

Moreover, Mozart allows users to dynamically create any number of sequential threads (more precisely, *dataflow threads*) and each one, while executing an operation, can suspend until all the needed operands have a well-defined value.

The network transparency to the user implies that a program will perform exactly the same computation, independently of how it is divided between a set of operating system processes.

The underlying representation of the language entities is inaccessible to the programmer, therefore Oz provides language security: all language entities, in fact, are created and passed explicitly, so an application cannot make references nor access references that have not been explicitly given to it.

Multilinguality in Mozart

Mozart is based on the **Oz language**, a multiparadigm high-level programming language which supports declarative programming, object-oriented programming, constraint programming, and concurrency. **Oz** is designed for advanced, concurrent, intelligent, networked, soft real-time, parallel, interactive, pro-active and reactive applications, providing fundamental features of object-oriented programming.

All versions of Oz are based on a concurrent constraint programming model, extended to support stateful computations, in other words computations on mutable objects.

The original Oz computation model, Oz 1 (released in 1995), supported a *fine-grained* notion of concurrency where *each statement* could potentially be executed concurrently. Oz 1 showed that its model, even though theoretically appealing, made it very hard for the programmer to control the resources of the application under development. Additionally, debugging programs was very hard and the object model tended to become unnecessarily awkward.

Oz 2 originated to overcome these problems: it used a *thread-based concurrency model*, with explicit creation of threads. A powerful new object system was designed and traditional exception handling constructs were added. In addition, the constraint solving and search capabilities were greatly enhanced. Oz 3 (from now on abbreviated with Oz) is the current language of the system and conservatively extends Oz 2 with two concepts:

- ***functor***: it is a kind of software component that specifies a module in terms of the other modules it needs, supporting *incremental construction* of programs from components that may be addressable over the Internet by URLs.

- ***future***: it is a logic variable that can be read but not written, allowing safe dataflow synchronisation over the Internet.

Oz provides programmers and system developers with a wide range of programming abstractions to enable them to develop complex applications quickly and robustly. It merges the several benefits of the various programming paradigms — object-oriented, functional and constraint logic programming — into one design.

It offers the most interesting features of *object-oriented programming*: state, abstract data types, classes, objects, and inheritance.

Whereas, from *functional programming* it derives a compositional syntax, first-class procedures, and lexical scoping: in fact, every Oz entity is first class, including procedures, threads, classes, methods, and objects.

Finally, from *logic programming and constraint programming* it exploits logic variables, disjunctive constructs, and programmable search strategies.

Furthermore, besides supporting many programming styles, Oz even supports combinations of them.

Support to AOSE in Mozart

The Mozart system mainly supports concurrency and distribution. From the logic and constraint programming paradigms it inherits the possibility to implement complex behaviours, whereas from the object-oriented paradigm it can obtain abstraction.

Graphical tools for modelling the MAS or for animating diagrams describing the architecture of the MAS are not supported. The underlying language seems to offer a wide range of programming abstractions to enable the programmers to quickly and robustly develop complex applications, but the various phases of the software engineering methodology do not seem to have been taken into account, or at least they are not explicitly supported by the platform.

It does not address issues as formal verification and validation, even though the presence of a logic and constraint programming paradigm in Oz should allow at least simple forms of verification of properties of the system being developed.

1.6 The Jack Platform

JACK (**J**ava **A**gent **C**ompiler and **K**ernel) ***Intelligent Agents***^{TM} [JACa, JACb] is an agent-oriented development environment created by the Agent Oriented Software Pty Ltd. JACK has been conceived to be an environment for creating, running and integrating commercial Java-based multi-agent software using a *component-based approach*.

Systems based on JACK are commercially deployed worldwide and are composed of reasoning entities that cooperate to achieve their goals.

It is built on top of Java and is fully compatible with other Java programs. The target application domains for JACK are: distributed business systems, command and control, intelligent applications and simulation.

The most important element in JACK is the *JACK Agent Language*, that is a programming language extending Java with agent-oriented concepts. Some of the Java classes defined in this language are:

- ***Agent***: an instance of this class represents a JACK intelligent agent and implements all the functionality associated with it. An agent is composed of capabilities, plans, databases, and events. An agent can

address other agents and post events to them, thus modelling inter-agent communication.

- ***Capability***: an instance of this class represents a functional aspect of an agent that can be plugged into it as required.
- ***Event***: an instance of this class represents an event, that causes an activity within JACK; it is distinguished in normal or BDI event. It arises internally to an agent as reasoning progresses, as a result in a change in the beliefs of the agent, or on receiving a communication from another agent.
- ***Plan***: an instance of this class describes a sequence of actions that an agent can take in response to an event occurred.
- ***BeliefSet***: an instance of this class represents the maintenance of agent beliefs that are central to the notion of BDI.

One approach to agent-oriented systems, corresponding to the **BDI architecture**, views the system as a rational agent having certain mental attitudes of ***B****elief*, ***D****esire* and ***I****ntention* (**BDI**). These mental attitudes determine the behaviour of the system and are critical for achieving adequate or optimal performance when deliberation is subject to limitations imposed by the resources. Each agent has, in fact:

- a set of *beliefs* about the world (its data set);
- a set of *events* that it will respond to;
- a set of *goals* that it may desire to achieve (either at the request of an external agent, reacting to an event, or when one or more of its beliefs change);
- a set of *plans* that describe how it can handle the goals or events that may occur.

JACK implements *BDI agents* since their reasoning behaviour follows the theoretical BDI model of artificial intelligence. A JACK agent is a software component that can exhibit both a *pro-active* (or goal-directed) and *reactive* (event-driven) behaviour.

Besides supporting the BDI agent model, it also supports Simple Team, an extension that allows team-based reasoning. A generic software system

implementing a BDI agent is an *event-driven program*: in reaction to an event, for instance a change in the environment or in its own beliefs, a BDI agent adopts a plan as one of its intentions.

A step of a plan can consist of adding a goal (that is, a desire to achieve a certain objective) to the agent itself, changing its beliefs, interacting with other agents, and any other atomic action on the state of the agent or on the external world.

The agent executes the steps of the plans until further deliberation is required; this may happen because of new events or the failure or successful conclusion of existing intentions.
An agent can be compared to a person having certain plans to achieve its goals, in particular a JACK agent is able to exhibit the following properties associated with rational behaviour:

Goal-directed focus The agent focuses on the objective and not on the way to achieve it.

Real-time context sensitivity The agent remembers which options are applicable at each given moment, and then makes decisions about what to try and retry according to present conditions.

Real-time validation of approach The agent will perform certain actions only while specific maintenance conditions are true.

Concurrency The agent system is multi-threaded. If new goals and events arise, the agent will be able to shift priority between them and multi-task as needed.

The three main extensions to Java introduced by JACK are:

- ***syntactical extensions***: keywords to refer to the main components of an agent (like *agent*, *plan* and *event*) and various statements for defining and manipulating such components;

- an ***ad-hoc compiler*** to convert the above components into ordinary Java classes, in order that they can be used by other Java code;

- the ***kernel***, namely a collection of classes that implements the runtime needed by the generated code, including a high performance infrastructure implementing concurrency management and communication in multi-agent applications.

Support to MAS development in JACK

JACK is a system that provides programming constructs for representing and implementing reasoning.

Its framework supplies a high performance, lightweight implementation of the BDI architecture, and can be easily extended to support different agent models or specific application requirements.

Moreover, JACK provides the core architecture and infrastructure for developing and running software agents in distributed applications.

The BDI model has demonstrated itself to be well suited for modelling intelligent behaviour in rapidly changing worlds, in fact it has been successfully adopted in different fields as simulation of military tactics, applications of business rules in workflows, and others.

A JACK agent is a computational implementation of the BDI model and as such it provides a reasoning framework with a specific set of language constructs. These constructs are just one possible implementation of the BDI model but provide the programmer with a modelling framework that is a mix of the high-level representational abstraction of the BDI model and the low level detail of the Java language.
Since it is based on Java, it allows an easy integration with legacy software.

BDI systems able to operate in dynamic, volatile, or real-time environments can be constructed by adding an operational semantics associated to plans and consistent with the notions of bounded or limited resources.

Multilinguality in JACK

JACK does not use a specific communication language and lacks a strong communication model. Java is the only language provided to implement the agents: both their knowledge and their behaviour.

The underlying Java infrastructure certainly makes it easier to integrate the system with other systems or languages, but such integration must be made by the developer.

Support to AOSE in JACK

The BDI approach offers a way to verify and validate the model of the application. The ***JACK Development Environment*** (JDE) offers a high-level design tool, a graphical plan editor and graphical tracing of plan execution, that provide a powerful and flexible program development environment.

The JDE also allows non-programmers to specify high-level agent reasoning methods that can then be produced by programmers. Team plans can contain graphical reasoning methods.

The agent-oriented model used in JACK is an extension of the encapsulation that is defined in the object-oriented model: the aim of the former model is to include a higher level representation of behaviour.

JACK offers easy integration and reuse since it allows modular design using plugins that can be added to the system extending it.

1.7 Conclusions

Since the introduction of powerful *abstractions* by means of the object-oriented programming paradigm, the inherent complexity of software systems has been dealt with more easyly and software engineering has greatly improved.

On the other hand, the agent-based computing has become a promising approach to developing various complex, typically distributed, computer systems. Agent-based solutions have already been developed for many different application domains, too.

The main characteristic that distinguishes agents from other software components is their *autonomy*: an agent has the ability to independently decide what to do in order to meet a purpose. However, agents usually do not exist "alone" but interact with other entities by exchanging messages, by means of asynchronous communication, and are able to collaborate while preserving their autonomy.

Agents are not native concepts of the language chosen to implement a MAS, but are higher level entities that use specific communication and execution mechanisms in order to conform with the agent paradigm.

Nowadays, the MAS topic is being dealt with in many disciplines outside of Artificial Intelligence, as for instance: biology, sociology, economics, complex systems, and philosophy.

Models, technologies and even methodologies are thus needed in order to support the developers to engineer MASs in a robust, reliable, and even re-

peatable fashion. Therefore, to facilitate the development of MASs, researchers create infrastructures that provide communication languages, interaction protocols and agent architectures.

Examples of "popular" agent frameworks that have been presented in the past years (late 1997) are: ***Voyager*** [RS], ***Aglets*** [IBM] and ***Odyssey*** [Mag].

Many methodologies and models borrowing abstractions and concepts from the organisation and sociology disciplines have been proposed to understand, model, reason, analyse and design MASs.

The point is that, although these models and methodologies all view a MAS as an organised society of agents, the organisation abstractions, concepts, assumptions and models of it are actually different. In fact, different methodologies and models have different purposes and assumptions about organisation, model the MAS at different levels, use different organisational and social concepts that are used at different stages in the software engineering lifecycle, and therefore have different application domains.

In addition, the same concept may have various interpretations and definitions in models and methodologies.

In this chapter, we have chosen to describe a few of the different approaches followed by creators of tools supporting the development of MASs: one focuses on *concurrency* between the agents while the other one on *distribution*.

As an example, we have also briefly introduced three software platforms that are specifically designed for building MASs (or can be used to this purpose as well), and analysed the aspects that we think should be taken into account when creating these tools:

- support to MAS developing;
- support to multilinguality (at any level);
- support to the AOSE process.

In the next chapter we describe our approach, which focuses on tools for rapid prototyping of MASs composed of heterogeneous agents.

Chapter 2

The DCaseLP environment

2.1 Introduction

DCaseLP [Mig02, AMMM02] stands for ***D****istributed* ***CaseLP***, where CaseLP [MMZ99, BDM+99] is the acronym for ***C****omplex* ***a****pplication* ***s****pecification* ***e****nvironment based on* ***L****ogic* ***P****rogramming*.

DCaseLP is a software environment designed and developed by the *Logic Programming Group* at the Department of Computer Science of the University of Genova in Italy, with the aim of providing a development tool for specifying, implementing, executing and debugging prototypes of multi-agent systems (abbreviated with **MASs**).

DCaseLP has originated from the intention of overcoming the deficiencies present in a previously developed software tool, namely CaseLP, which provides a prototyping method and a set of tools and languages to support the realisation of prototypes of complex applications.

Correctness and reliability of a developed software are diffiult to be guaranteed, particularly for distributed software systems where a set of entities have to cooperate and coordinate in order to exchange information. Moreover, many distributed systems must use existing software modules and, consequently, must integrate information from a potentially large number of different sources.

Thus, *integration* and *reuse* of different kinds of *information* and *software tools* constitute an urgent need that new software products must satisfy. CaseLP came into being with the purpose of helping the developer of MASs to accomplish these tasks.

In this chapter we first introduce the **ARPEGGIO** (***A****gent based* ***R****apid*

***P**rototyping **E**nvironment **G**ood for **G**lobal **I**nformation **O**rganisation*) framework [DKM+99], which proposes a general open approach for the specification, rapid prototyping and engineering of agent-based software.

ARPEGGIO encourages the use of logics and logic programming in realising an open MAS: logic-based formalisms are considered suitable for the specification, direct execution and verification of the prototype. CaseLP was, indeed, created as an instance of this general framework.

After introducing ARPEGGIO, we will outline the advantages and disadvanteges of CaseLP first, and then of the first release of DCaseLP, ending with the rationale of integrating into it "logical agents", proceeding towards the current release of DCaseLP.

2.2 ARPEGGIO

The ARPEGGIO framework [DKM+99] proposes an open approach based on Logic Programming that has been set up with the aim of providing tools and methodologies to support a software engineer realise new distributed and heterogeneous software applications modelled as MASs.

In other words, the objective of the framework is to provide the building blocks for the development of a hybrid, customisable AOSE methodology, without forcing the resulting system to be hybrid.
Three international research groups:

- the *Logic Programming Group* at the Department of Computer Science of the University of Maryland in USA,
- the *Logic Programming and Software Engineering Group* at the Department of Computer Science and Software Engineering of the University of Melbourne in Australia,
- and the *Logic Programming Group* at the Department of Computer Science of the University of Genova in Italy,

decided to pool their experiences in logic programming to reach the goal of creating a unique "development model" characterised by a modular AOSE approach and by the property of being *open*, both to existing software and to its distribution.

In the ARPEGGIO's philosophy, *rapid prototyping* is considered to be a fundamental activity in the software development process because it combines execution with debugging.

Figure 2.1: The ARPEGGIO's Approach

A well-established method is needed not only to develop applications whose components have complex interaction, but also to provide modelling techniques and tools that allow testing, verification, refinement of (partial) specifications and prototypes, at different levels of detail.

The intended approach (see Figure 2.1) is one that simultaneously carries out specification and prototyping, in order to obtain an *iterative* and *incremental* development of a final validated formal specification.

The focus is on the realisation of an environment capable of supporting the complementary techniques of *animating* parts of the specification and providing specification-based *testing* and formal verification of correctness.

With the term "animation" we intend making the (declarative) specifications *executable*, offering in this way a type of throw-away prototype and the possibility to interactively test a model and its properties. There are very good reasons to be interested in this process of animation, mainly being: it can be

highly automated, it is cheap to perform and, through the static analysis of its results, provides important information about the model so far developed and that is usually implicit, as assumptions and implications.

Animation brings a formal specification to life in the sense that, in an interactive way, it validates the software engineers' current understanding of the system against that of the client, enabling both to provide valuable feedback.

The first step of software engineering — the requirements analysis — can, therefore, be completed through experimentation by means of animation, obtaining more information on the system than that available from the client's model. It certainly helps, in this matter, the fact that a client is more likely to understand an animation than a formal specification.

For what we have said until now, it should be clear that the approach followed within the framework is a *testing oriented* one.

2.2.1 The general architecture

The ARPEGGIO's architecture is as general as possible, but it has been conceived to be *open*, so it also takes into account logic programming, or more generally declarative programming, beside the more popular graphical languages (like UML) and imperative languages.

Logic programming has many interesting features: for instance, it is a high level programming paradigm that can help to make specifications and code seem alike, and it can provide the right combination of formalism and experimentation.

Since a logic program can be viewed as an *executable* specification, the logic paradigm constitutes a step forward for rapid prototyping. The metaprogramming features of logic programming contribute to the "open" property of the MAS since they allow the definition, in a flexible way, of agents with different architectures and control, suitable for different tasks, to be easily added to the system.

A fundamental aim of ARPEGGIO is to create an environment that provides a set of different languages for specifying particular aspects of the MAS's architecture and behaviour: the developer will then choose the most useful between the available, non-executable and executable, specifications.

For every specification language, there must exist a (semi-)automatic compiler that can produce an executable form of any specified agent, in a logic programming target language. If architectural details of the agents are not included in the specification, then such (semi-)automatic tools will add the

default details for building an executable prototype.

Once a prototype of the MAS has been created by compiling the various components of the specification, tests can be carried out in order to detect problems with the specification itself.

Using an executable specification language means being able not only to directly interpret the given specification before compiling and animating it, but also to analyse the specification behaviour and to refine it without having to develop the prototype. Both, direct execution and animation, have the purpose of providing a better understanding of the system, thus facilitating the *refinement of the specification*, until the obtained executable prototype behaves exactly as expected.

It should be possible to integrate into the prototype legacy software, data and, perhaps, also external specifications.

Agents are distinguished from external legacy modules since the latter are software components that provide functionality without communication capability, autonomy, or intelligence and can only be inserted into a MAS if there are agents that can interface with them.

Thus, specification languages can be used not only for defining agents, but also for describing the behaviour of an external module. When dealing with not-executable languages, an ad-hoc compilation process can be applied to create an animated form of the specification.

Finally, the executable MAS prototype should be implemented in the most appropriate programming language from those available.

The agents specified with different languages, and then compiled into executable forms, are able to communicate with each other because the target language in which they are compiled is either the same or an interface can easily be defined between them so they can interact without losing efficiency.

The communication between the agents is accomplished through a common agent communication language.

2.2.2 The proposed methodology

So, the methodology proposed by the ARPEGGIO framework can be summarised as follows:

- *specification step*: lists the agents that compose the MAS, the services they provide, which services they require, and the communication channels that exist between agents. During this phase the most appropriate

architecture for each agent in the system is decided — for example, reactive, proactive or BDI. This stage also describes the agents behaviour, in the sense that it states how they operate;

- *animation step*: serves to analyse how the previously specified system behaves. In this stage the specification is tested and/or analysed and verified to check how much it corresponds to the requirements. Testing can take place only if the specification is executable;

- *refinement step*: analysis, verification and testing can give feedback to the previous steps so, exploiting the results obtained by the animation, this step refines the specification, perhaps choosing another specification language and/or modelling lower-level details or different aspects from those described before;

- *improvement step*: the system is improved by resolving the problems that were encountered by animating the refined specification;

- *execution step*: the prototype is executed and tested with respect to "real" communication among agents and actual interaction with legacy programs, data or specifications. Any error or misbehaviour found in this step may require a revision of the choices made in the previous steps. After some revisions, the behaviour of the implemented prototype will match all the initial requirements.

2.2.3 The ARPEGGIO partners

The *Logic Programming Group* at the Department of Computer Science of the University of Maryland in USA has worked for years on semantically integrating different, and possibly heterogeneous, information sources and reasoning systems. They accomplished such a task by executing programs — the mediators — written in the **HERMES** (A **HE**terogeneous **R**easoning and **M**ediator **S**ystem [LNS96, AS95]) system they developed, based on the theory of Hybrid Knowledge Bases. A very interesting project is also IMPACT (**I**nteractive **M**aryland **P**latform for **A**gents **C**ollaborating **T**ogether [ESP99, ES99, AEK+99]); it includes both a theory as well as a software implementation, that facilitates the creation, deployment, interaction, and collaborative aspects of software agents in a heterogeneous, distributed environment.

The *Logic Programming and Software Engineering Group* at the Department of Computer Science and Software Engineering of the University of Melbourne in Australia has developed great experience in the animation of logic specifications, in order to execute and test them [SCT96], in particular with the PipeDream project (**P**rototyp**I**ng s**PE**cifications, **D**esign and **RE**quirements **A**t **M**elbourne). PipeDream aims to improve the outcomes of requirements analysis by using formal methods, or more precisely mathematical modelling, to determine, analyse and verify requirements. Logic programming can provide the basis for a light weight approach to achieving better analysis of specifications.

Finally, the *Logic Programming Group* at the Department of Computer Science of the University of Genova in Italy has developed **CaseLP**, an experimental environment for building prototypes of complex and distributed applications modelled as MAS. The CaseLP tool allows to specify agents using logic languages (such as $\mathcal{E}_{hhf}$ [Del97]), its implementation is based on Prolog and has been developed as an instance of the ARPEGGIO framework. During the development of the tool, much work has been done to integrate more widely used specification languages with logic ones, in order to take advantage of the benefits of logics and logic programming and to reach a wider group of potential users.

Moreover, within the ARPEGGIO framework, the Italian and Australian groups have defined a new AOSE approach with the purpose of creating and reusing already existing AOSE methodologies in a modular way.

What is interesting about a modular approach is that it enables developers to build custom project-specific methodologies from AOSE features just like applications are built from reusable off-the-shelf components. The interested reader can find a conceptual framework for creating and reusing modular methodologies in [JMMS03].

Such conceptual framework is based on the definition of an *AOSE feature*, which performs one or more development activities, such as analysis, and addresses one or more software quality attributes, such as privacy. An AOSE feature encapsulates software engineering techniques and models, together with supporting CASE tools, and development knowledge such as design patterns.

The applicability of this approach has been demonstrated by modularising into AOSE features four existing methodologies, Prometheus [PW02], ROADMAP [JPS02], CaseLP and the conventional Object-Oriented approach.

2.3 DCaseLP

Now we will talk about the main subject of this thesis: **DCaseLP** (***D****istributed* ***CaseLP***) [Mig02, AMMM02], a rapid prototyping software environment that supports the development of MASs and, as its predecessor **CaseLP** (***C****omplex* ***a****pplication* ***s****pecification* ***e****nvironment based on* ***L****ogic* ***P****rogramming*) [MMZ99, BDM+99], has been designed and developed by the Logic Programming Group at the Department of Computer Science of the University of Genova in Italy.

DCaseLP aims at providing the developer of a MAS with an AOSE methodology and a software environment to be used during the requirements analysis, the design and the development of a working prototype. A fundamental goal is to support the development of MASs consisting of *multilingual* agents.

More precisely, *more than one language* must be available not only to specify the agents belonging to the system, but also to define their architecture, behaviour and state, allowing both the existence and communication in the environment of agents created using such different languages.

As its name emphasises, DCaseLP has not been created from scratch, but has come into being to overcome deficiencies present in its predecessor CaseLP: the latter supports multilingual agents at the specification level, but not at a lower level, since all the agents are coded in the logic programming language Prolog.

The previous, and first, release of DCaseLP [AMMM02] overcame most of CaseLP's limitations but did not allow to develop a completely multilingual prototype.

The work described in this thesis — the integration of logical agents — has lead to the current working version of DCaseLP, which provides an additional language to implement agents in the MAS but is not, yet, the final version since it is an ongoing work and can be subject to additional integrations.

2.3.1 DCaseLP's focuses

It is quite clear that the development of a working prototype of a MAS can require a long time and different skills, and is more complicated if the system is composed of *heterogeneous agents*. During prototyping, the heterogeneity of agents emerges from three different features:

- the *specification language*;
- the *architecture*;

- the *implementation language*.

Specification language: during the specification stage, the developer might need, maybe for only some of the agents, to emphasise particular features of their behaviour and, to do so, would like to use one or more different specification languages than the one currently in use and probably satisfactory for specifying other agents in the system.

Architecture: it determines the way the agent reasons, represents its knowledge, plans its behaviour and makes decisions. The developer must be able to choose from the variety of architectures that have been defined in the agent community, in order to maintain the differences existing between the agents being modelled.

Implementation language: the developer should be able to define the behaviour and the state of each agent using the more appropriate language, taking into account the architecture of the agent itself or the particular application domain, instead of being forced to implement all the agents with the same language. By integrating into the same running prototype different implementation languages, some of which directly executable, it is also possible to directly implement some of the agents, skipping the specification stage when this one is not relevant.

A software environment claiming to integrate heterogeneous agents must show the basic attribute of *multilingualism* in all these three aspects, and this is what DCaseLP wants to accomplish.

The development strategy that DCaseLP proposes (refer to [Mig02] for details) is that one of specifying each "view" of the MAS — as, for example, the MAS' architecture, the interaction protocols between agents, the internal architecture and functioning of each agent — *using the language that is most suitable for the current description*, and to subsequently verify, execute, or animate the obtained specifications. These specifications are checked not through formal validation and verification methods (not yet dealt with in DCaseLP) but by producing an executable code and running the developed prototype.

It should be clear by now that the methodology proposed by DCaseLP, together with the feature of providing more than one language to define different aspects of an agent, allow to consider it an instance of the ARPEGGIO framework, as its predecessor CaseLP.

Agents need to interact and exchange information in order to cooperate or compete for the control of shared resources; this interaction may follow sophisticated communication protocols to which the developed prototype must adhere.

Such interaction protocols are considered a basic matter in DCaseLP and, as detailed further in this chapter, DCaseLP provides the developer with the means to exactly specify which agents can take part to a"conversation" and in which order they interact, making the debugging of the prototype much easier.

Before describing DCaseLP, it is best to detail the features characterising CaseLP, so the reader will have a better understanding of the DCaseLP environment and of the overall ideas behind it that have brought it into existence.

2.3.2 CaseLP

CaseLP[MMZ99, BDM+99] provides a well-defined prototyping method, as well as a set of tools and languages which support the developer during the realisation of the MAS prototype and are helpfull in testing distributed software applications.

A logic programming language has been chosen as the basis for the prototyping environment because it is powerful, declarative and an executable specification language and, together with the agent technology, can play a very effective role in the rapid prototyping, testing and refinement of a wide spectrum of software applications.

At the *system specification level*, an architectural description language can be adopted to describe the prototype in terms of agents classes, their instances, the services they provide/require and their communication "links".

At the *agent specification level*, a rule-based, not executable language can be used to easily define reactive and proactive agents. An executable, linear logic language can define more sophisticated agents and the system in which they operate. Furthermore, an imperative language, **HEMASL** [Mar99, MMMZ00a, MMMZ00b], can describe both the agents architectures, the agents classes, their instances and the environment in which they are embedded.

Finally, at the *implementation level*, a Prolog-like language, extended with additional primitives, has been used. Obviously, in order to really be a supporting development environment, CaseLP also offers simulation tools to visualise the execution of the prototype and to collect statistics about it.

In Figure 2.2 the reader can see the main languages that CaseLP offers to the developers.

The communication among agents takes place through message passing and the agent communication language used is **KQML** [FLM95] based on speech-acts, although any other communication language can be easily adopted by the agents.

Figure 2.2: The main languages available in the CaseLP environment.

The Limitations

By providing a set of specification languages for the definition of the behaviour of the agents, as well as a methodology to translate/integrate them into a unique executable specification, *CaseLP has succeeded in obtaining multilingualism in both the specification and the architecture of the agents, though missing the multilingualism in the implementation language.*
Besides this lack, CaseLP also has other limitations:

① centralisation;

② poor support to concurrency;

③ limited portability.

Centralisation

In CaseLP, the prototype is executed through a *centralised* round-robin scheduler that activates all the agents in turn in the MAS, following a fixed cyclic order. Once an agent is activated, it behaves accordingly to the rules that define the actions it should take, and when it has terminated the scheduler activates the next agent. The scheduler controls the global clock of the MAS by managing the simulation time, and also handles the exchange of messages between agents.

Concurrency
CaseLP does not allow the real concurrent execution of agents, since there is no way to have more than one agent activated in the MAS at the same time. As we have said above, the CaseLP's scheduler activates only one agent at a time, and it is not possible to have more than one scheduler running in the MAS: concurrency among agents is only simulated.

Portability
CaseLP was initially developed as an extension of a constraint logic programming language with theories, communication predicates and "safe" state update predicates (that guarantee no permanent effect in case of failure), therefore it has been implemented in Prolog. Unfortunately, at present Prolog is not widely used as a programming language for (commercial) applications, thus there is not a Prolog interpreter to consider "portable" to the systems available in the industry domain.

These features are, nowadays, a *must* for the majority of commercial (and not) applications, thus they had to be added in some way. Since it did not seem easy to achieve such characteristics directly from the Prolog infrastructure on which CaseLP is built upon, the adopted solution has been to develop a totally new environment, DCaseLP indeed, based on another programming language (Java^{TM} [Mic]) widely used nowadays, with the intention of realising the same project from which CaseLP originated.

2.3.3 DCaseLP's first release

The first version of DCaseLP has been developed mainly concentrating on overcoming the three above-cited lacks: such purpose has been fulfilled through the introduction of **JADE**^{TM} (***J**ava **A**gent **DE**velopment Framework* [Til]), a software framework fully implemented in the **Java**^{TM} [Mic] programming language and whose minimal system requirement is the version 1.4 of the Java run-time environment (more details on JADE can be found in the next chapter, in Section 3.3).

Since JADE runs as a Java application, it runs in a JVM (**J**ava **V**irtual **M**achine) and, therefore, is portable on most of the available operating systems. Agents in JADE are implemented with Java threads, thus it is possible to execute, while the JADE platform is running, more than one agent simultaneously achieving the aimed concurrency. Distribution comes directly from the

possibility of distributing JVMs and from Java's ***R**emote **M**ethod **I**nvocation* (RMI) mechanism.

This release of DCaseLP represented a step forward towards realising the aimed multilingualism since it proposed two implementation languages: the Java language and the **Jess** [FH] expert system language.

Unfortunately, the multilingualism of this version was still too limited and CaseLP's languages/tools were quite far from being exploitable by it.

As far as the multilingualism at the specification level is concerned, this release offered the possibility to specify some aspects of the MAS using AUML (***A**gent-based **U**nified **M**odelling **L**anguage*) [AUM], and automatically create Jess agents from the AUML specification.

In Figure 2.3, we have shown the languages available in the DCaseLP environment.

Figure 2.3: The languages available in the first release of the DCaseLP environment.

The new agents: Java and Jess

The introduction of the JADE platform has, at first, lead to the new type of agents executable in a prototype of MAS: the ***Java agents***.

JADE is a framework for implementing working MASs and provides many functionalities that facilitate the management of all the agents that constitute the system. As it is, it cannot be considered a real *software development tool* for MASs because it does not support the entire software engineering process: it does not provide any means by which to create an abstract model of the system to develop, nor by which to easily pass from the designed components/agents to the software objects.

It supplies many Java packages among which there are the ones containing the classes to use to implement agents, their ontologies and their behaviours. An agent is created extending an appropriate class that allows to take basic actions and to define new ones (more details can be found in JADE's documentation downloadable from *http://jade.tilab.com/*). Both the knowledge and the behaviour of these agents are entirely implemented in Java, and this is why they are called **Java agents**.

Beside the Java agents, another type of agents was introduced in the first version of DCaseLP: the ***Jess agents***.

Their behaviour and knowledge is represented using declarative rules written in Jess, a rule engine and scripting language entirely written in Java. Jess is a language inspired by the **CLIPS** [Cen] expert system shell: it allows to create a knowledge base and to interact with it through the use of an inference engine; in other words, it adds to applications the capacity to reason using knowledge supplied in the form of declarative rules.

A **Jess agent** is a JADE agent extended in order to embed a Jess interpreter and Jess rules. The architecture is a deliberative one: it is composed of a list of *Jess rules* defining the behaviour of the agent, a list of *Jess facts* representing its internal state and a *Jess interpreter* that represents the core of the agent.

Once the rules representing the behaviour of an agent have been specified, it is possible to create a Jess agent embedding such rules and then run it in the JADE platform (more details can be found in [AMMM02]).

As it will be detailed forward, DCaseLP provides a semi-automised mechanism that can transform graphical diagrams (describing the behaviour of an agent) into the Jess agent to be run in the platform.

Introducing an expert system language as implementation language

Expert systems were the most important AI technology of the 1980s: they are programs that emulate human expertise in well defined problem domains, with

the purpose of solving problems or giving advice.

An expert system closely resembles human logic in its implementation and is, therefore, easier to develop and maintain than a usual software system. The languages or tools used to build them exploit techniques which allow the modelling of information at higher levels of abstraction. The simplest expert system uses *facts*, symbolically represented, to derive some conclusion.

A common technique used for developing expert systems is the "rule-based" programming: it uses *rules* to represent heuristics — commonsense sets of rules intended to increase the probability of solving some problems — and to specify a set of actions to be performed for a given situation.

A "rule" is, basically, a conditional statement in which the "if" portion is a series of *patterns* which specify the facts (or data) that cause the rule to be applicable, while the "then" portion is the set of actions to be executed when the rule is applicable. The process of matching facts, currently present in the knowledge base, to patterns is called *pattern matching* and always occurs whenever changes are made to such facts.

An expert system tool provides a mechanism — the *inference engine* — which automatically matches facts against patterns and determines which rules are applicable. An inference engine constitutes the "reasoning mechanism" of the system.

The most important distinction between agents and expert systems is that the latter are inherently disembodied: they do not interact directly with any environment and their information is obtained not through sensors, but through a user acting as an intermediary.

In the same way, they do not act on any environment, but rather give feedback or advice to a third party. In addition, co-operation is usually not considered by expert systems.

To understand the advantage deriving from implementing agents with a language for expert systems, it should be mentioned that a declarative knowledge, compared to an imperative one, is:

- reusable;
- more modular and flexible;
- has better semantics;
- makes detecting and correcting contradictory knowledge easier;
- provides an abstraction of the (real) world in a natural way.

In addition, the use of meta-programming techniques in a declarative setting provides a support for the integration of different kinds of knowledge.
Therefore, the introduction of the Jess language in DCaseLP has made possible the *explicit representation of the knowledge of the agent* and, as a consequence, has made possible in the overall MAS to:

- model rationality of the agent;
- verify system's properties;
- enhance knowledge sharing and communication between agents.

Jess was chosen because it is small, light, one of the fastest rule engines available and is written in Java. Moreover, being a ruled-based language, Jess is suitable for representing both the event-driven and the goal-driven behaviours of the agents.

The development of Jess Agents

Using the standard Object modelling language **UML** (OMG's ***U**nified **M**odelling **L**anguage* [OMGb]) and other languages, it is possible to specify the architecture of an agent — in other words, its internal structure and functioning —, the roles played by the agent classes, the interaction protocols between agent roles, the ontologies [1] known by the agents, the associations between roles and agent classes and instances of agent classes.

The *role* played by an agent defines an interaction protocol to which the agent must adhere: it establishes which received messages the agent can "understand", what kind of messages can be used to reply, and the order with which the message exchanging must take place with other roles. The roles are the entities taking part to the communication in a protocol and must not be confused with the agent instances.

An agent can play more than one role, the collection of which characterises its "social ability". Roles are, therefore, fundamental not just for the functioning of individual agents, but also for the overall communication in the MAS since they dominate it.

An *agent class* groups together all the agents that play identical roles, have the same architecture and ontologies; those belonging to an agent class are,

[1] Even though included in the descriptive characterisation of an agent and conceptually integrated, ontologies have not yet been explicitly integrated in DCaseLP and are still implicit in the code of the agents.

therefore, equal in their behaviour but different in the data that represent their knowledge. Interaction is easier between agents that have the same "ontologies", since they have in common the way in which knowledge is represented and structured.

In the specification stage, especially to describe these interaction protocols that the agents must follow, DCaseLP also allows the use of **AUML** (**A**gent-based **U**nified **M**odelling **L**anguage) [AUM], an extension of UML to represent agents and their interactions, developed by the FIPA (The **F**oundation for **I**ntelligent **P**hysical **A**gents) [FIP] Modelling Technical Committee. Particularly useful are the *protocol diagrams* proposed by AUML, but in DCaseLP it has been necessary to impose some restrictions to their use:

- there can only be *one* role beginning the communication in the protocol;
- a protocol regarding more than two roles should be divided in sub-protocols, each dealing with only two roles;
- a protocol containing a concurrent sending of messages is represented in an explicit way, in order to emphasise the fact that the thread is divided in sub-threads, and each of these sends a message.

The choice of UML and AUML, initially developed for documentation purposes, to *represent interaction protocols* in DCaseLP is motivated by the wide support that they are obtaining from the agent research community. AUML cannot be considered a standard agent modelling language yet, but it has many chances to become such.

The UML/AUML diagrams corresponding to an agent class are automatically translated by DCaseLP into a set of Jess rules; these rules, though, must be manually completed with the details not included in the specifications but necessary for executing the prototype as, for example, some kind of constraints, the initial state of each agent instance and so on.

The Jess rules, obtained by the automatic translation process supplied by the environment, are only partially istantiated because the protocol diagrams used in DCaseLP only specify *the flow of the messages exchanged* and not under which conditions such messages must be sent/blocked. Completing these rules, the developer creates a collection of Jess facts that constitutes the initial state of each agent instance.

Since both Jess and JADE are implemented in Java, the Jess agents thus obtained can be easily executed in the JADE platform and, using JADE's tools, monitored and debugged in order to perform the "coherence check": the results of the execution runs are compared to the "interaction specifications" to see if the "conversations" described in the latter are respected throughout the various executions of the prototype.

Since it is not possible yet to carry out, *a priori*, a formal proof of properties of the resulting system, the research group that has developed DCaseLP is investigating how to integrate formal methods to perform validation tests, including checking if the AUML descriptions adhere to the specifications derived from the requirements analysis.

By using DCaseLP, therefore, the developer can create in a semi-automatic way agents that satisfy the interaction protocols and the UML/AUML diagrams and, by executing the prototype, can check if the requirements analysis has been correctly carried out.

The idea of translating UML and AUML diagrams into a formalism and checking their properties by either animating or formally verifying the resulting code is shared by many researchers working in the AOSE field [Hug02, MEH02, SA03].

The reader interested in more detail on the passages that lead to the Jess agents (starting from the diagrams) is referred to [AMMM02, AMMR03].

As it can be seen in Fig.2.4, the semi-automatic translation from UML/AUML diagrams into Jess agents is achieved by exploiting two different configuration files that are written in the XSL format, which is normally exploited to express style sheets. These configuration files have been developed as part of the work of the master thesis described in [Mig02].

Figure 2.4: The semi-automatic translation process that transforms UML diagrams into Jess agents.

The Limitations

Two are the missing features of the first release of DCaseLP on which we have focused our attention on:

- the possibility to reuse the Prolog-based code and instruments already developed for CaseLP;
- the ability to reason about properties of the interactions occurring between the agents.

This thesis work, that deals with the integration of tuProlog into DCaseLP, represents a step forward towards recuperating the functionalities offered by CaseLP. CaseLP is implemented in **SICStus Prolog** [SIC] and its agents are mainly SICStus Prolog code extended with ad–hoc communication primitives.

A lot of work has been done to study and define semi-automatic translators from high-level specification languages to CaseLP's implementation language:

the environment contains tools that semi-automatically translate $\mathcal{E}_{\boldsymbol{hhf}}$ [Del97], **HEMASL** [Mar99, MMMZ00a, MMMZ00b] and ***AgentRules*** [Mas02] into Prolog.

$\mathcal{E}_{hhf}$ has been, for example, used to model high-level interaction protocols in applications developed with CaseLP.

2.4 DCaseLP's current release

In the current release of DCaseLP, in which we have added the **tuProlog** [DOR01] logic programming language, there are now three kinds of agent that can be run in a developed prototype:

- *Java agents*
- *Jess agents*
- *tuProlog agents*

tuProlog is a declarative logical language for which there exists an inference engine written in Java. The *tuProlog agents* are "logical agents" intended as logic programs: the behaviour of these agents (as well as their knowledge) is, in fact, implemented by a Prolog theory.

We will detail these agents in Chapter 4, while the tuProlog and JADE systems are dealt with in Chapter 3. The tuProlog agent is the result of this thesis work and was, therefore, not present in the first version of DCaseLP.

Their integration into the DCaseLP environment does not affect Java/Jess agents developed with the previous version of the tool.

A developer that has more skills in logic programming rather than in object-oriented programming, can code agents belonging to the MAS entirely in Prolog now.

The next step to take to achieve the aim is to recuperate (from CaseLP) the use of $\mathcal{E}_{hhf}$, *AgentRules* and HEMASL: we think that having added Prolog to DCaseLP, this task will be more easy to accomplish.

Verification of specifications

By executing the $\mathcal{E}_{hhf}$ specification of the MAS, the prototype can be tested and its correctness verified by using the $\mathcal{E}_{hhf}$ interpreter. CaseLP is, thus, capable of providing a limited support to *formal verification of specifications*.

Without introducing the Prolog language, all that work would have not been exploitable from DCaseLP. The ability to specify agents as Prolog theories, should make it easier to use the tools available in CaseLP, since the latter is totally implemented in a logic programming language.

Reasoning about interaction protocols means to check if a certain set of properties holds after a conversation has taken place. This can, for instance, allow to determine which protocol (from a set of available ones) satisfies a goal of interest, or it can help to find out which protocols can be combined to accomplish complex tasks.

After proving desired properties of the interaction protocols, the developer can animate them through the facilities offered by DCaseLP.

Furthermore, we have also discussed (not as part of this thesis) about the advantages of adding to our environment an agent programming language, called ***DyLOG*** [BGMP, BBMP03, BBG$^+$05], which allows to reason about actions and change [BBG$^+$04].

Since DyLOG is based on modal logic, it can be exploited both as an implementation language and for verifying properties. DyLOG is considered particularly helpful in this matter, since it makes possible to formally prove if every conversation generated by a DyLOG program is correct with respect to a specification expressed by an AUML diagram.

In the first release of DCaseLP, AUML's interaction protocols could only be translated into Jess code and, thereby, could not be formally verified but just executed.

Adopting the DyLOG language in DCaseLP, we offer the developer the possibility of converting AUML's sequence diagrams into a DyLOG program, automatically verify if the latter is consistent with the AUML specifications and if it satisfies certain properties.

The existing DyLOG interpreter, implemented in SICStus Prolog, is the means by which a developer can carry out such formal proofs.

The integration of DyLOG into DCaseLP is, for now, "methodological": it extends the set of languages supported by DCaseLP during the MAS engineering process and augments the verification capabilities of such environment, without requiring any "real" integration of the DyLOG interpreter.

Initially, DyLOG was implemented in SICStus Prolog (as CaseLP); currently, the designers of DyLOG have developed an implementation of the language, including its communication kit, totally based on the Java language [SLBB04].

Moreover, DyLOG's new implementation exploits classes offered by the

tuProlog system and a Java interpreter of DyLOG programs is being developed. This "use" of tuProlog by Dylog's implementation seems to suggest that a physical integration of DyLOG into DCaseLP should not be too difficult: the tuProlog agents can be exploited as an interface.

In Figure 2.5 we have shown the languages available in the current release of DCaseLPand those that will be in the future: the two dotted arrows that point from DyLOG to Prolog and Java, respectively, represent the future work of integrating DyLOG into tuProlog agents as well as into Java agents.

Once the physical integration into DCaseLP will be completed — namely DyLOG's Java interpreter will be available together with its integration into JADE — it will be possible to animate complete DyLOG agents into DCaseLP, ending with agents specified in Jess, Java, tuProlog and DyLOG all runnable on the same platform and all able to communicate.

Figure 2.5: The languages available in the current and future release of the DCaseLP environment: the dotted lines are part of the future work.

2.5 Conclusions

In this chapter it has been outlined the "evolution" of the CASE tool DCaseLP: more precisely, it has been shown that DCaseLP is the result of a work concentrated on optimising a previously developed prototyping environment, CaseLP, which is the instance of a more general framework, namely ARPEGGIO.

As it has been mentioned, CaseLP was the first tool developed by the Logic Programming Group at the Department of Computer Science of the University of Genova in Italy, in order to provide support to the developer of a MAS during all the phases of software engineering.

It originated as an experimental logic-based prototyping environment for MASs and, by allowing different specification languages — referred as *multilingualism at the specification level* — for modelling different aspects of the MAS, it provides the flexibility needed to describe a MAS from different points of view.

In time, the researchers became aware that, even though CaseLP proved to be very useful in developing real applications run by industries, it lacked multilingualism at the implementation level.

So, to overcome the main limitations of CaseLP, the Logic Programming Group developed DCaseLP: the latter is not implemented by a logic programming language as the former, but instead it is built upon the Object-Oriented programming language Java. The latter did, indeed, overcome the former's limitations but, unfortunately, it lost the ability of exploiting all the languages available in the former.

Finally, we have extended the first release of DCaseLP obtaining a seamless integration of a logic programming language in the Object-Oriented infrastructure, and simultaneously made a first step towards being able to use the tools already offered by CaseLP.

Moreover, we have realised a "methodological integration" (not part of the aim of this thesis though) of DyLOG, which can be used to formally verify the graphical specifications produced during the development of the MAS. It is part of the future work to complete this integration in order to include that language as an implementation language too.

An important issue of all the work that has been done in CaseLP and in both the releases of DCaseLP is that of *not forcing* the developer of the MAS to be an expert of *all* the supported languages in the tool: the developer is free to use the language he/she is more familiar with, allowing thus more reliable specifications and implementations.

Chapter 3

The tuProlog and JADE systems

3.1 Introduction

The objective of this thesis is to integrate *logical agents* into DCaseLP, which means to enable the developer to implement an agent in the MAS choosing between not only Java and Jess, as it is possible in the first version of the environment, but also the logic programming language *Prolog*.

Prolog is a simple subset of predicate logic and the prime example of a logic programming language; it has become a standard tool in Artificial Intelligence (**AI**) programming since it is the most widely available in the logic programming paradigm.

Mainly, three are the features that make Prolog a suitable language for "facing" problems in AI:

① its *syntax and semantics are much closer to formal logic*, the most common means by which in the AI literature facts and reasoning methods are represented;

② it *provides automatic backtracking*, making considerably easier the "search" process, the most important of all AI techniques;

③ it *supports multiway reasoning*, namely, the same predicate can be used in many ways since the input/output mode can be arbitrary assigned to its parameters. Logic programs may be executed either in a "forward" direction or in a "backward" direction, according to the bounding of

the variables during execution itself. This leads to a flexible reasoning mechanism.

Furthermore, Prolog has find great application in the processing of natural language since it has a high ability in manipulating symbols. It has also gain popularity for *planning and control* involving backtrack searching and incomplete searching.

So, to integrate the Prolog language in the tool, we had to integrate a Prolog interpreter: we have chosen the one available with the **tuProlog** [DOR01] system, whose features are forward described in section 3.2.

Both the tuProlog and JADE systems are run from a Java Virtual Machine (**JVM**), so they can be executed from any operating system that supports the Java technology: the Java Runtime Environment is sufficient, so J2SE (**J**ava **2** Platform, **S**tandard **E**dition) is the least required.

This chapter is organised as follows: first, there is a brief description of the tuProlog system, then insights into the JADE platform are given, since it constitutes the backbone of the DCaseLP environment.

3.2 The tuProlog system

tuProlog [DOR01] is a Java-based Prolog system that provides a *bidirectional interface between Java and Prolog*:

- ***Prolog FROM Java:*** a Java-based Prolog virtual machine and an ad-hoc Java API (**A**pplication **P**rogram **I**nterface) allow the use of Prolog from a Java program;
- ***Java FROM Prolog:*** the ***JavaLibrary*** provided by the tuProlog system allows a tuProlog theory to contain predicates that are able to access a Java entity, since the latter can be represented as a Prolog term.

The Java API that builds the system is composed of three packages:

- ***alice.tuprolog***,
- ***alice.tuprologx*** and
- ***alice.util*** .

The name "alice" appears in the name of the packages because *aliCE* is the name of the research team that has developed the system.
The *alice.tuprolog* package can be considered the basic API since it contains the major classes needed to create Prolog terms from inside Java code.

By importing classes from such package, a Java application is able to create a Prolog engine, a Prolog theory, and solve a Prolog goal making the engine attempt to prove that the goal logically follows from the theory.

The powerful characteristic of tuProlog's API is, therefore, that it enables a Java programmer to easily construct objects which have *reasoning capabilities*.

A tuProlog engine, as well as any other tuProlog term, is a normal object within the Java code and, as a consequence, more than one tuProlog engine can be created within a Java program and each of them configured with its own libraries and knowledge base.

tuProlog also supports interaction via TCP/IP and RMI, and can be provided as a CORBA service: all desirable features in applications that run in the Internet, where communication and services are key words.

From the Prolog side, Java objects can directly be used within tuProlog code: therefore, tuProlog components can be created by choosing the most suitable paradigm, either declarative/logic or imperative/Object-Oriented.

The reasons for which tuProlog has been designated as the "ideal" candidate for implementing *logical agents* in DCaseLP, among several Prolog interpreters available and for which there exists an interface with Java, are:

- it offers a light-weight and easily deployable Prolog engine, entirely implemented as a Java class;
- it is free for any non-commercial use;
- it is dynamically configurable.

It provides a "minimal", but still efficient, Prolog engine that is easy deployable since it is a light-weight Java object, accessible through a minimal interface. Such minimality makes tuProlog suitable for building (possibly mobile) applications to run on the Internet, where overloading must be avoided.

A tuProlog engine can be extended and dynamically configured by loading/unloading tuProlog libraries, which implement predicates, and are developed in Java. This feature allows to develop more flexible applications that can be adapted to different environments.

tuProlog can be downloaded from the Web site: *http://www.lia.deis.unibo.it/tuprolog*. The release that has been used in this thesis is **2p-1.2.0**.

The tuProlog's GUI

The tuProlog system offers a graphic environment from which the user can issue queries, view the substitutions applied to the variables in the solution of a goal, load a theory, and so on.
Such environment is created by executing the ***alice.tuprologx.ide.GUIConsole*** class from a JVM. If the path of the tuProlog library **2p.jar** has been included in the environment variable CLASSPATH, the GUI is launched by executing the following command line from a JVM:

```
java alice.tuprologx.ide.GUIConsole
```

Figure 3.1 is a snapshot of the GUI: as an example, we have solved the goal:

```
?- X is 5,Y is 6,Z is X+Y.
```

by typing it in the ***Query*** frame.
As it can be seen in the ***Answer*** frame, the resolution has succeeded, yielding the answer "yes" and the variables substitutions that the Prolog interpreter has applied to the goal.

So, launching this GUI, it is possible to interact with a tuProlog engine, modify its knowledge base, solve goals and view the solutions.

Moreover, tuProlog libraries can be dynamically loaded/unloaded using the appropriate predicates provided by the system (the reader is referred to the manual of tuProlog for a complete list of the available tuProlog's built-in predicates).

Figure 3.1: The graphic interface to tuProlog created by executing the **alice.tuprologx.ide.GUIConsole** class.

The tuProlog's console

Alternatively, a user can execute the ***alice.tuprologx.ide.CUIConsole*** class from a JVM, which loads the tuProlog's console. If the path of the tuProlog library **2p.jar** has been included in the environment variable CLASSPATH, the console is launched by executing the following command line from a JVM:

```
java alice.tuprologx.ide.CUIConsole
```

Figure 3.2 shows a snapshot of this console: as an example, we have tried to solve the same goal that we have solved in the GUI (*nl* is the "new line" predicate that is defined in the tuProlog library *IOLibrary*).

The IOLibrary is one of the libraries that are loaded by default in the tuProlog interpreter that runs inside the console. It defines the Prolog predicates that enable interaction between Prolog programs and external resources, typically files and I/O channels.

The predicate *nl* fails only when it is not possible to add a new line control character on the current output stream.

In the example, the predicate succeeded everytime yielding the "yes" answer on a new line, and not on the goal's following line as it usually happens. From inside the GUI or the console, it is possible to load a Prolog theory as a text file and then solve a goal within that theory.

As it will be detailed, the integration of the logical agents in DCaseLP has been carried out without using the GUI or the console, but just exploiting some Java classes available in the tuProlog packages.

The Prolog syntax supported by the tuProlog engine is almost ISO compliant: it does not support ISO exceptions, ISO I/O predicates and some ISO directives.

In order to program the tuProlog agents, the developer of the MAS *does not have to know the tuProlog system*, but just the logic programming paradigm and what standard Prolog predicates are available in tuProlog.

3.3 The JADE platform

JADE (**J**ava **A**gent **DE**velopment Framework) [Til] is a middleware for the development of distributed multi-agent applications: it is a free software distributed in open source under the terms of the LGPL (**L**esser **G**eneral **P**ublic **L**icense Version 2) by TILAB, the copyright holder.

Figure 3.2: The tuProlog "console" created executing the **alice.tuprologx.ide .CUIConsole** class.

The current growth of network-based applications has lead to an increasing importance of middleware technologies. In a distributed computing system, *middleware* is defined as the software layer that lies between the operating system and the applications on each site of the system.

Middleware is comparable to a "black box": it helps a programmer get the job done quicker and not have to worry about unnecessary details. Most application programmers can keep such black box "closed": all they need is to know how to make "calls" to the box and get results back, but not to understand how the box works.

JADE is a software framework, fully implemented in the Java language, built with the purpose of *simplifying the implementation of MASs*, offering a set of graphical tools to support the debugging and deployment phases.

Moreover, each agent executed in the platform adheres to the FIPA's specifications and can rely on a library of FIPA's interaction protocols ready to be used, where FIPA (The **F**oundation for **I**ntelligent **P**hysical **A**gents) [FIP] is an international organisation that promotes agent technologies.

A JADE-based application consists of one or more agents that can communicate with each other through a message exchange mechanism which is at disposal through the environment embedding them. The agents can "act" on such environment and "perceive" events from it, as requested by the general "situatedness" feature of intelligent agents.

Intelligence, initiative, information, resources and control can be fully distributed on mobile terminals as well as on computers belonging to a hard-wired network.

JADE allows to build distributed *peer-to-peer* applications in wired/wireless networks: as opposed to the traditional "client/server" model, the information, the resources and the control can be distributed among the entities of the system, allowing each of them to connect and directly exchange data.

According to the needs of the application, the environment can dinamically evolve, implying agents appearing or disappearing from it. Any agent running in a wired or wireless network can start a communication with another agent or respond to it.

A description of the FIPA's standard architecture is now presented to the reader, before showing how the JADE platform is run.

3.3.1 FIPA's standards

FIPA's main concern is to create and maintain *specifications*, and such work is carried out by Technical Committees. The various standards proposed by the organisation are grouped according to their subject; the subjects dealt with are:

- *Applications*
- *Abstract Architecture*
- *Agent Communication*
- *Agent Management*
- *Agent Message Transport*

The *Applications* specifications provide examples of areas in which FIPA agents can be deployed. Every specification describes an ontology for the particular domain and the agents that compose the overall system.

The *Agent Communication* specifications describe the structure of the messages that are exchanged, the interaction protocols used, the kind of communicative actions carried out by the agents — based on the theory of speech acts — and the content language representations.

The *Agent Management* specifications describe what is considered essential in controlling and managing agents residing in the same platform or distributed in more platforms. They propose an agent management model that includes a list of the basic services and define open standard interfaces for accessing such services.

The model provides a standard framework within which FIPA-compliant agents exist and operate: it supplies a logical model for the *creation*, *registration*, *location*, *communication*, *migration* and the *termination* of agents.

The entities that appear in the model are logical services and do not imply any physical configuration. Moreover, the implementation details of individual agent platforms and agents are matters that must be dealt with during the actual development of an agent system. The logical components of the model represent services that can be implemented by one or more physical entities in a real agent platform.

The *Agent Message Transport* specifications outline the transport mechanism of messages among agent platforms and the representation of messages across different network transport protocols, including wired and wireless environments.

3.3.2 A FIPA-compliant architecture

The *FIPA Abstract Architecture Specification* (downloadable from http://www.fipa.org/specs/fipa00001/) defines which are the main architectural elements in an agent system and their relationships.

Such an architecture cannot be directly implemented, but it should be used as a basis by developers of *concrete* architectural specifications which describe in detail how to construct an agent system, including the agents and the services that they rely upon, in terms of concrete software components, such as programming languages, applications programming interfaces, network protocols, operating system services, and so forth.

It does not characterise a unique real architecture but *multiple concrete realisations*: in fact, it merely addresses how interoperable features should be implemented but does not forbid additional features.

The developers of *concrete* specifications of FIPA-compliant agent systems, willing to provide specifications with appropriate levels of interoperability, must ensure that their work conforms to such abstract architecture.

Similarly, developers specifying applications that will run on FIPA-compliant agent systems, will have to know what services and features they can exploit from their applications.

The internal design and implementation of agents and agent management infrastructure are not dealt with in this specification, as well as elements inherently coupled with specific implementations of the architecture.

The main concern of the FIPA Abstract Architecture is to represent a standard for agent systems that describes a semantically meaningful message exchange between agents regardless of the means of transport used for messages, the communication language that represents the messages and, the content languages with which the information is encoded.

In order to create agent systems deployable in commercial settings, it is important to understand and to use existing software environments.

The overall goal is, in fact, to *support the creation of agent systems that can seamlessly be integrated within a specific computing environment while interoperating with other agent systems situated in separate environments.*

To encourage *interoperability* and *reusability* it is necessary to identify "architectural abstractions" that can be formally related to *every* valid implementation.

An abstract description of agent systems gives more insights into the relationships between their fundamental elements: from the set of abstract archi-

tectural elements and relations it is possible to derive a collection of possible concrete architectures that will be able to interoperate with each other because they share a common abstract design.

So, a concrete architectural specification that is FIPA-compliant satisfies certain properties and provides the following mechanisms:

- *agent registration*
- *agent discovery*
- *inter-agent message transfer.*

The components of the Abstract Architecture are the *minimum required elements* of an agent system: their *realisation* is totally up to the designer of the concrete architecture who just defines them in terms of concrete architectural terms.

This architecture avoids issues regarding the internal structure of an agent and all the details concerning the services offered by agents: they are all dealt with by the concrete architectures.

In addition to a number of standard services, including *agent-directory* services and *message-transport* service, the Abstract Architecture defines a general services model that includes a *service-directory* service. The way in which these services are supplied is not a matter of concern at the abstract level: they may be implemented either as agents or as software that is accessed via method invocation or programming interfaces.

Agent-directory service: the basic role of this abstract element is to offer a *registration mechanism* to agents, in order to make it easy for agents to "locate" agents with which they wish to interact. It is a shared information repository in which agents may "register" themselves and in which they may search for other agents.

Service-directory service: the basic role of this abstract element is to help agents/services to find services. More precisely, the *service-directory* service is "where" agents/services register the descriptions of the services to be used inside the platform and "where" agents/services can look for a service they need. It differs from the agent-directory service since it finds services and not agents. A service-directory service is also used to store the service descriptions

of application-oriented services, such as commercial and business-oriented services.

Message-transport service: the basic role of this abstract element is to support the sending and receiving of messages between agents by managing all the details regarding the actual transportation.
At an abstract level, two agents willing to interact with each other need at first to ***locate*** one another before they can communicate: that is why they must ***register*** themselves and, finally, they can communicate through message passing.

3.3.3 Communication in a FIPA-compliant platform

The kind of "conversations" that agents have between them are very often much complex and the simple *request-response* communication, typical of the client-server model, is not sufficient.

Agents communicate by exchanging messages which represent speech acts, and which are encoded in an agent communication language (ACL).

In agent communication there are three fundamental aspects to consider: the structure of the messages sent/received, their representation and their transport.

The various Agent Communication specifications have been grouped according to the aspect of communication that they deal with, namely the following:

- ***Communicative Acts***
- ***Content Languages***
- ***Interaction Protocols***

Communicative Acts
The Communicative Acts (CAs) specifications beside providing a library for communicative acts and requirements for new ones, formally define the communication language and its semantics. The formal definitions provide a clear and unambiguous way for expressing the standardised meaning of the communicative actions carried out by the agents, mainly the messages they exchange and the protocols they use. For more details, see the "FIPA Communicative Act Library Specification" downloadable from *http://www.fipa.org/specs/fipa00037/*.

Content Languages
The Content Language (CL) specifications deal with different representations of the content of messages and describe the SL content language. SL stands for Semantic Language and is the formal language used to define the semantics of FIPA's Agent Communication Language (ACL). A concrete syntax for the SL language is presented: such syntax and its associated semantics are suggested as a candidate content language for the ACL. For more details, see the "FIPA SL Content Language Specification" downloadable from *http://www.fipa.org/specs/fipa00008/* .

Interaction Protocols
The Interaction Protocols (IPs) specifications deal with pre-agreed message exchange protocols for ACL messages. They constitute frameworks for the communicative acts defined in the CAs specifications. Every IP specification describes the flow of messages exchanged by the participants to the particular kind of conversation that the IP is meant to detail. The various IP specifications are viewable at *http://www.fipa.org/repository/ips.php3*.

A *message* represents a communicative act of an agent towards another or more agents. To realise a communicative act it is necessary to encode in some way the message and then transmit it to the destination(s).

The so-called *ACL message* is the entity that is transferred among the agents: the description of its structure can be found in "FIPA ACL Message Structure Specification" (downloadable from *http://www.fipa.org/specs/fipa00061/*).

In order to accomplish "sensible" conversations, agents must share a common vocabulary so they can understand the meaning of the terms, objects, relations and so on, represented in the exchanged messages.

An *ontology* actually provides such a vocabulary. It allows to represent and communicate knowledge regarding some topic and a set of relationships and properties that hold for the entities denoted by its vocabulary. It is a collection of symbols that have a corresponding interpretation associated to them which may be shared by some agents or services: such symbols usually refer to objects and relationships regarding the subject domain.

An ACL message contains the message, called *content*, together with additional information like the type of communicative act that it represents (for instance, a request), the sender and receivers and, the ontologies to use to interpret the content.

Obviously, the parameters needed to effectively communicate vary accord-

ing to the application domain, so the only parameter that is required in all the messages is the *performative*, although presumably most of the messages will also contain an indication regarding the sender, receivers and the content. The messages transmitted within specific implementations can include user-defined parameters in addition to the ones proposed by FIPA's standard.

The content of an ACL message refers to a domain or topic area and is expressed using a content language. The content of the message is understood by the receivers with the aid of one or more ontologies.

FIPA's standard does not impose any content language nor its encoding. The way in which a message is encoded depends on the concrete architecture.

3.3.4 JADE's components

The JADE [1] framework is composed of various Java packages containing the classes necessary to develop agents to run in the platform, together with the run-time environment from which the basic services and other useful classes are available in the development of a MAS.

JADE's packages

Among the various Java packages that "constitute" JADE, there are:

- ***jade.core:*** it implements the "core" of the system, especially the *Agent* class that must be extended by developers in order to create JADE agents.
- ***jade.lang.acl:*** it is necessary to process Agent Communication Language according to FIPA's standard specifications (see Section 3.3.3).
- ***jade.content:*** its aim is to support user-defined ontologies and content-languages.
- ***jade.domain:*** it enables the creation of all the elements needed for the agent management, including the entities defined by FIPA's standard.
- ***jade.gui:*** it contains classes useful in creating GUIs that can display and edit agent identifiers, agent descriptions, ACL messages, etc. .

[1] The release used during the development of the thesis is **JADE 3.0b1**, last updated 27th February 2003.

- ***jade.mtp:*** it supplies a Java interface that every Message Transport Protocol should implement to be easily integrated into the JADE framework.

- ***jade.proto:*** it offers classes to use to model standard interaction protocols or to create new ones.

- ***jade.wrapper:*** it provides wrappers for JADE's elements that implement higher-level functionalities in order to allow the use of JADE as a library by Java applications that can, therefore, launch JADE agents or platforms.

- ***jade.tools:*** it simplifies the platform administration and application development by providing ad-hoc tools, most of which are very easy-to-use graphical interfaces.

3.3.5 Running the JADE platform

Before running a JADE platform, it is necessary to make sure that the following JAR files — the **J**ava^{TM} **AR**chive file format that enables to bundle multiple files into a single file — have been added to the CLASSPATH environment variable (explanations about JAR files and how to manage them can be found at *http://java.sun.com/docs/books/tutorial/jar/basics/index.html*):

- *Base64.jar*
- *http.jar*
- *iiop.jar*
- *jade.jar*
- *jadeTools.jar*

The fundamental Java class is with no doubt the **Boot** class in the ***jade*** package since it implements the JADE's run-time environment, namely the means by which JADE agents "live": every JADE platform must run *at least* one instance of this class.

The term ***container*** is used to refer to any instance of the class Boot because it can contain agents, while ***main container*** only refers to the first one that is launched in order to start the platform.

So, a JADE platform is a collection of containers in which there is *only one* main container: the containers can run in JVMs deployed anywhere in a

network to which the main container belongs and, are not forced to run in the same environment of the main container.

The main container has a special role because it is always active in the platform and has additional responsibilities, like keeping track of which containers are active in the platform, where they are deployed in the network, and so on.

A container that is not the main one is launched with the **`-container`** option following `java jade.Boot`, whereas the main one is launched simply with `java jade.Boot` or with the **`-main`** option.

Loading agents into containers

There are three different ways to load agents into a container:

- **from the command line:**

  - ★ `java jade.Boot [options] [`**`list of agents`**`]`

    The user can directly load the agents into the container at the moment in which the latter is started. In the command line issued to the JVM, the user must specify the `list of agents`: it is a sequence of strings (separated by a blank space), each representing an agent to load in the platform (in Chapter 4, this matter will be detailed while presenting how the tuProlog agents are loaded into DCaseLP).

  - ★ `java jade.Boot` **`-gui`** `[options]`

    Launching the container with the `-gui` option causes a graphical user-friendly interface (GUI), called RMA (**R**emote **M**onitoring **A**gent), to appear (see Figure 3.3). From this GUI the user is then able to load any JADE agent.

- **from a Java application:**

  - invoking the static method `instance` of the *jade.core.Runtime* class, the application is able to obtain the single instance of such class;
  - calling the method `createMainContainer` or `createAgentContainer` on the *Runtime* instance, the application obtains an instance of the class *jade.wrapper.AgentContainer*, which allows to access most of the functionalities available in an agent container;

- calling the method `createNewAgent` on the *AgentContainer* instance the application is able to load an agent into the container.

For more details, the reader is referred to the "JADE programmers guide" viewable at *http://jade.cselt.it/doc/programmersguide.pdf*.

Figure 3.3: The graphical user-friendly interface RMA obtained issuing the `-gui` option when launching an agent container.

3.3.6 The main container

The role of every container is to provide an homogeneous layer upon which execute the agents and to manage the complexity and diversity of the underlying tiers (hardware, operating systems, networks, etc.) on behalf of the developers of the agents.

Any container belonging to the platform must register with the main container when it is executed: the user must input to it the *host* and the *port number* of the main container upon which it "depends".

Only one container can run in a JVM but it is sufficient as a stand-alone environment for executing several agents and allowing their concurrent execution on the same host.

A host can simultaneously run more than one JVM, implying that more agent containers can be launched on the same host. All the containers, remote

or not, that connect themselves with the same main container are part of the same agent platform.

What distinguishes the main container from the other ones is that two agents are automatically run when it start its execution: the AMS (***A****gent* ***M****anagement* ***S****ystem*) and the DF (***D****irectory* ***F****acilitator*).

The AMS is fundamental for the management of the agents belonging to the platform: it provides the naming service (ensuring that each agent in the platform has a unique name) and represents the administrator of the platform since it can create/kill either agents on remote containers or the containers themselves.

Therefore, there is only one AMS in a platform and controls any kind of access to the platform. It provides a *White Pages* service and manages the life-cycle of the agents by maintaining a directory of **a**gent **id**entifiers (**AIDs**) and agents state; each agent, in fact, must register with an AMS in order to get a valid AID.

After connecting to the main container, the other agent containers can provide a complete run-time environment for the execution of any set of JADE agents.

The DF implements a *Yellow Pages* service: it stores a list of *descriptions of services* that agents in the platform offer to other agents.

An agent *publishes/registers* one or more services by providing their descriptions to the DF.
To every published service the DF associates a description including:

- the AID of the agent providing the service;
- the list of languages and ontologies that the other agents must know to interact with the agent that realises the service;
- the service type;
- the service name;
- service-specific properties.

Useful classes in creating such descriptions and their contents can be found in the ***jade.domain.FIPAAgentManagement*** package.

When an agent terminates its execution, it should *de-register* services that it has published and not already de-registered.

An agent looking for services must provide the DF with a *template description*. As a result of the search, the DF returns to the agent the list of all the descriptions that *match* the provided template.

A description matches the template if all the fields specified in the template are present in the description and contain the same values that are specified in the template.

Other DF agents can be activated in the platform and several DF agents (including the default one created with the main container) can be federated, thus providing a single *distributed Yellow Pages Catalogue*.

Moreover, if an agent wishes to be notified as soon as a certain service is published, JADE allows it to subscribe to the DF for such notification.

3.3.7 Communication in JADE

Since the agents belonging to a JADE platform can be distributed over a network, a communication mechanism for a distributed environment is necessary in order to effectively implement interaction between such agents: Java's RMI (***R**emote **M**ethod **I**nvocation*) mechanism is used, so there is no need for additional software since it is included in the Java language.

The AMS and DF agents are not the only entities automatically created when the main container is loaded: the other object that comes into being is, indeed, the ***RMI registry*** which represents the platform's RMI server and must reside on the host on which the main container runs.

RMI is used for *intra-platform* communication, whereas CORBA or HTTP are used for *inter-platform* communication.

FIPA's MTS (**M**essage **T**ransport **S**ystem), also called ACC (**A**gent **C**ommunication **C**hannel), is the software component to which is assigned the task of controlling the exchange of messages both within the platform and between (remote) platforms.

Furthermore, the physical transfer of messages between two ACCs is achieved through the MTP (**M**essage **T**ransport **P**rotocol) protocol.

Since an ACC is delegated to the delivery of a message, it is not unusual for it to access information provided by other services (such as the AMS and DF) that reside on other platforms.

According to FIPA's standard, agents do not directly communicate with each other: an agent can either communicate with its ACC or with the receiver's ACC.

JADE fully complies with FIPA's architecture and when one of its platforms is launched, the AMS and DF are immediately created together with the ACC module which is set to allow message communication.

The ACC does not interpret the content of messages: its role is to deliver the message to the agent receiver, relieving the developer from the burden of dealing with all the transfer details.

The communication paradigm adopted in JADE is the *asynchronous message passing*. The agent sender/receiver of a message is specified by its name, since in a JADE platform agents are identified by a *unique name*.

If an agent wants to interact with another agent, regardless of its actual location — the same container, a different container of the same platform or another platform —, all it needs is just to know the *name* of such agent.

A name is a string composed of a "nickname" (usually chosen when the agent is loaded into the platform) and an "address" separated by the "at" (@) sign: for instance, **`RMA@ai:1099/JADE`** is an agent whose nickname is **`RMA`** and whose address is **`ai:1099/JADE`**.

The addresses are in RMI format, namely they consist of a *host name* (**`ai`**) in the example shown above) and, a *port* on which the RMI naming service is active (**`1099`** is the default one for RMI).

In the example, the host name does not specify the domain but a full name is also valid as a JADE address.

The ending string "/JADE" distinguishes the RMI invocations made by a JADE entity from others made by RMI services that have nothing to do with JADE.

For each agent, the run-time environment keeps a *message queue* where it "piles" the messages received by the agent considered. An agent is notified each time that a message is put into the queue: it is totally up to the developer of the agent to establish if and when the agent will process that message.

Agents belonging to the same platform may be subject to different policies regarding the "collecting" and "processing" of their messages, depending on the developer's choices.

So, JADE creates and manages the queue of incoming ACL messages, private to each agent and, offers the agents different access modes to such queue: blocking, polling, timeout or based on pattern matching.

Not only the full FIPA's communication model has been implemented, but JADE's transport mechanism adapts to different situations by choosing the best available protocol among Java RMI, event-notification, HTTP, IIOP, and user-defined protocols.

JADE offers classes/interfaces that can be used within the framework to implement most of FIPA's interaction protocols, the SL language and the agent management ontology, as well as user-defined content languages and ontologies.

3.3.8 Agents behaviours

Agents are subject to various external events and often have to carry out several concurrent tasks in response to them. Therefore, each agent running in JADE is implemented as a single *Java thread*, which means that it can exploit the multi-thread solution offered by the JAVA language.

Furthermore, in order to improve the parallel execution of the tasks of an agent, JADE has implemented a *scheduling of cooperative behaviours*, and ***Behaviour*** is the class whose instances represent and implement the tasks of the agent.

The developer can implement multi-threaded JADE agents, but the framework does not supply specific support for them, except for the synchronisation of the ACL message queue associated to each agent.

JADE keeps a list of tasks for each agent and the developer is able to add tasks to it since the framework offers more than one "skeleton" as a basis for creating not only common behaviours in agent programming but also totally new ones.

A scheduler, implemented by the agent but hidden to the programmer, pursues a round-robin non-preemptive scheduling policy between all available behaviours (which form the *ready queue*), executing one behaviour at a time.

The `done` method of the Behaviour class is invoked by the scheduler in order to know if the task has been completed or not: if the behaviour has not finished, then it will be rescheduled at the next round. When a behaviour has terminated its job, the scheduler removes it from the ready queue.

The entire computation state of a behaviour must be stored in instance variables belonging to the behaviour. To avoid wasting CPU time, every behaviour can block its computation and enter the "blocked queue", and is rescheduled as soon as a new message arrives.

Moreover, if needed, a behaviour can block itself for a certain amount of time, specified by the programmer.

Since the scheduling policy of the behaviours is non-preemptive, the developer must be careful not to create behaviours that are endless loops or that perform long operations, otherwise such a behaviour could stop the others associated to the same agent.

If an agent behaviour is blocked for some reason, the behaviours of another agent will carry on achieving their aims due to the fact that agents run in different Java threads.

So, there is no way of interrupting a behaviour to let the agent execute another behaviour and to then start the previous from the point it was interrupted.

One of JADE's most interesting features that is worth taking into account and has not been particularly highlighted here is the possibility for the administrator of the MAS to remotely control the whole platform.

It should also be remembered that all the administration activities within the platform are made much easier with the aid of the GUIs offered by the JADE framework.

3.4 Conclusions

This chapter has been dedicated to briefly introducing the essential features of the tuProlog system and, on the contrary, to describing in more detail the JADE system that constitutes the backbone of the DCaseLP environment.

As it will be clear in the next chapter, the developer of tuProlog agents in DCaseLP only needs logic programming skills rather than having a good familiarity with the tuProlog system itself.

JADE and tuProlog have, indeed, a common aspect: many of their interesting properties are directly derived from features of the language with which both are implemented, the Java programming language.

JADE is an agent platform that was already present in the first release of DCaseLP: this chapter has just recalled its main characteristics in order to make it easier for the reader to understand the integration of tuProlog into DCaseLP (dealt with in the next chapter) and the general mechanism that determines the behaviour of the agents in the platform.
The main qualities of JADE can be summarised as follows:

- it is a middleware for the development and deployment of *distributed* multi-agent applications;
- it is *portable*, since it is fully implemented in the Java language;
- it is FIPA-compliant;
- it offers GUIs to ease the debugging and monitoring of agents/containers;

- it provides mechanisms to remotely administrate the platform.

tuProlog, on the other hand, has been chosen to implement the logical agents because:

- it provides an easy deployable Prolog engine implemented as a light-weight Java object, accessible through a minimal interface;
- the tuProlog engine can be dynamically extended and configured by loading/unloading tuProlog libraries, which can be implemented in Java or in tuProlog;
- the tuProlog system provides packages that allow a *bidirectional use of Prolog*: "Java in Prolog code" and "Prolog in Java code".

Furthermore, both JADE and tuProlog are free for non-commercial use.
In the next chapter the reader will finally see how the "tuProlog agents" were created and how JADE agents are really *shells* for agents implemented in languages that can interface with Java.

Chapter 4

The interface between Prolog and DCaseLP

4.1 Introduction

The solution we developed satisfies two important requirements: it is a *transparent integration* and guarantees the *communication* between Java, Jess and *logical* agents.

The term "transparent" has a different meaning in computing to that normally intended: it does not mean the quality of being easily seen through, but indicates instead the property of being *invisible*. Normally, a transparent integration is one that is undetectable in subsequent uses of the system involved, so our integration allows already developed MASs with the first release of DCaseLP to still be executed without the need of making any modification to their code.

The result of our work is the Java ***tuPInJADE*** package: it contains the classes that implement what we call *tuProlog agents*, to be run in the JADE environment, and whose behaviour is fully specified by means of the tuProlog language.

We use the term "logical" to refer to the tuProlog agents because we want to stress the fact that these agents not only have an inference engine, but also a *rational behaviour* incorporated in them that is the result of a resolution or proof technique.

We will now fully detail the contents of the Java package tuPInJADE that represents an interface between tuProlog and JADE, whereas in Chapter 5 we will give an example of the tuProlog agents use.

4.2 The tuPInJADE package

The tuPInJADE package contains the following files:

- ***ErrorMsg.java:*** it defines the Java class ***ErrorMsg*** that is used by the tuProlog agents with the aim of displaying a pop-up window with error and failure messages, since the JADE platform does not provide its agents with a similar mechanism.

- ***JadeShell42P.java:*** this represents a tuProlog agent and, as the name suggests, it behaves as a general "agent shell" for a tuProlog agent in JADE that incorporates a Prolog inference engine (implemented by tuProlog) and, when launched in a JADE platform, needs an input file containing the Prolog theory defining the agent behaviour.

- ***JadeShell42PGui.java:*** is similar to the JadeShell42P class, but differs from it since when loading the agent it displays an additional GUI from which the user can browse the file system and select the theory file to input to the agent.

- ***TuJadeLibrary.java:*** it is a tuProlog library (developed entirely in Java) necessary for a tuProlog agent to communicate in a JADE platform since it defines the communicative predicates based on the facilities that JADE offers to its agents for communication in a platform and with other platforms.

We will now detail every class belonging to this package.

4.2.1 The ErrorMsg class

tuProlog does not support exceptions so when goals fail because of exceptions, they simply end up being `false`. JADE does not provide a mechanism for handling exceptions: it just reports them.

This class implements a dialog window that is used by the tuProlog agents during their execution to inform the user/administrator of the platform about awkward situations that have occurred. For instance, the failure of a goal present in the ***main*** predicate that determines the agent behaviour, causes this window to appear.

To display any message, the agent invokes the method ***write*** of this class, with two strings as arguments: the first string is the title to display on the bar

of the window, while the second is the message that will be displayed in the text area of the window.

Once this window has popped-up, the agent stops its execution and remains in an idle state.

In Figure 4.1 we can see the window that the tuProlog agent named my2Pagent raised because its main goal has failed. The theory file given in input to the agent just contained "`main:-fail.`" that, in fact, forces the goal to fail.

Figure 4.1: The ErrorMsg GUI that the tuProlog agent my2Pagent pops-up since its ***main*** goal fails.

Here is the list of the possible messages that a tuProlog agent may display:

Title Bar	Message displayed	Explanation
The Theory Of Agent AGENT is Missing	**Agent AGENT has no theory specified.**	It appears if the agent AGENT is launched with no arguments, while the name of the theory file is expected.
Too Many Inputs For Agent AGENT	**You can input only ONE theory file of agent AGENT.**	It appears if the agent AGENT is launched with more than one argument, while only one file is expected to contain the theory.

Title Bar	Message displayed	Explanation
Error Loading The TuJadeLibrary Of Agent AGENT	**The TuJadeLibrary is not a valid tuProlog library.**	It appears if errors occur while loading the TuJadeLibrary (included in the tuPInJADE package) into the agent AGENT.
Error Loading The Theory Of Agent AGENT	**Cannot find the theory file FILE of agent AGENT.**	It appears if the system cannot find the theory file whose name FILE has been given in input when launching the agent AGENT.
Error Opening The Theory File Of Agent AGENT	**A security error occurred opening the theory file FILE of agent AGENT.**	It appears if the system encounters a problem while opening the theory file FILE.
Error Opening The Theory File Of Agent AGENT	**An error occurred opening the theory file FILE of agent AGENT.**	It appears if the system encounters an I/O problem while reading the theory file FILE.
Error In Theory Of Agent AGENT	**The file FILE does not contain a tuProlog theory. An error was detected while parsing line LINE.**	It appears if the theory file FILE does not contain a tuProlog theory.
Cannot Close Theory File Of Agent AGENT	**Cannot close the theory file FILE of agent AGENT.**	It appears if the system encounters an I/O error or a problem while closing the theory file FILE.

Title Bar	Message displayed	Explanation
Error Loading The Theory Of Agent AGENT	**There is an error in line** LINE **of the tuProlog theory contained in file** FILE.	It appears if in line LINE of the theory file FILE the tuProlog engine encountered an error.
Syntax Error In Theory File Of Agent AGENT	**The theory file** FILE **of agent** AGENT **does not begin with a word.**	It appears if the theory file FILE of the agent AGENT does not begin with a word, whereas the word `main` is expected.
Syntax Error In Theory File Of Agent AGENT	**In the theory file** FILE **of agent** AGENT **the key word** `main` **is not followed by** `:-`.	It appears if in the theory file FILE of the agent AGENT the key word `main` is not followed by the characters `:-`.
Syntax Error In Theory File Of Agent AGENT	**In the theory file** FILE **of agent** AGENT **the key character** `:` **is not followed by** `-`.	It appears if in the theory file FILE of the agent AGENT the key character `:` is not followed by the character `-`.
Syntax Error In Theory File Of Agent AGENT	**In the theory file** FILE **of agent** AGENT **there is another character following** `main` **where** `:` **is expected.**	It appears if in the theory file FILE of the agent AGENT the key word `main` is not followed by the character `:`.

Title Bar	Message displayed	Explanation
Syntax Error In Theory File Of Agent AGENT	**The theory file FILE of agent AGENT begins with a word that is not main.**	It appears if the theory file FILE of the agent AGENT begins with a word that is not `main`.
Syntax Error In Theory File Of Agent AGENT	**An error occurred while reading the theory file FILE of agent AGENT.**	It appears if the system encounters an error while reading the theory file FILE of the agent AGENT.
Error Obtaining The TuJadeLibrary From Agent AGENT	**The TuJadeLibrary was not loaded in agent AGENT.**	It appears if the system encounters a problem loading the tuProlog library *TuJadeLibrary* in the agent AGENT.
Error Solving The ''main'' Goal Of Agent AGENT	*A message stating the error encountered while evaluating a predicate defined in TuJadeLibrary, during the demonstration of the main goal of the agent* AGENT	It appears when the demonstration of the *main* goal of the agent AGENT fails because of the failure of a predicate defined in TuJadeLibrary.
Theory Of Agent AGENT Must Be Selected In GUI	**The theory of agent AGENT should not be specified as an argument but should be selected from the GUI that pops-up when the agent is launched.**	It appears if an argument is specified when launching the agent AGENT since the theory file is expected to be selected in the GUI.

Title Bar	Message displayed	Explanation
Theory Of Agent AGENT Not Selected	**No theory has been selected in the GUI displayed for agent AGENT.**	It appears if the user/administrator of the platform does not select the theory file of agent AGENT.

The only information provided to the user, as can be seen by the messages listed above, concerns I/O errors regarding the theory file, or the *failure* of the demonstration of the goal (`main`) representing the agent behaviour. The message displayed normally reports the error encounted by the tuProlog engine while carrying out the resolution.

4.2.2 The JadeShell42P class

"JADE PROGRAMMER'S GUIDE" and "JADE ADMINISTRATOR'S GUIDE", both downloadable from *http://jade.cselt.it/doc/index.html*, contain instructions on how to use the platform, together with useful implementation guidelines that the developer should follow in order to code (Java) agents executable in a JADE agent container.

JADE provides packages that define various classes specifically designed to be extended by the developers and that implement basic functionalities.

In the JADE framework, any instance of a Java class that extends the ***Agent*** class defined in the ***jade.core*** package is considered an agent. By extending the *jade.core.Agent* class, any agent inherits the features (ready-to-use) by which it can carry out basic interactions with the agent platform — such as registration, configuration, remote management, etc. — and a basic set of methods that can be called to implement the actions of the agent — like sending/receiving messages, using standard interaction protocols and so on.

The JadeShell42P class that we have developed represents a *shell* for an agent programmed in tuProlog: it is like an oyster to which the programmer provides the pearl (see Figure 4.2 and Figure 4.3).

Figure 4.2: The JadeShell42P agent before it is loaded in the JADE platform

Figure 4.3: The JadeShell42P agent once it has been loaded in the JADE platform

This shell incorporates a tuProlog engine, implemented by the ***alice.tuprolog.Prolog*** class, and is, therefore, capable of making demonstrations of *goals* (those defined in logic programming).
More precisely, when loading this agent in JADE the user must *complete* it by giving in input the name of the file containing a tuProlog theory that determines the behaviour (this will be explained later) of the agent.
Such a theory, besides being compliant with tuProlog syntax, must satisfy only one constraint: it has to begin with the definition of the `main/0` predicate.
The JadeShell42P agent can be loaded in two ways:

① **from the RMA GUI**

By clicking the "Start New Agent" item from the "Actions" menu in the title bar, or the button shown in the image on the right of Figure 4.4, the window shown in Figure 4.5 is displayed on the screen.
After having inputted the appropriate parameters, the new agent is created in the selected agent container (the "Main-Container" in the two images of Figure 4.4).
The parameters that the user must input are: the local name of the agent, the fully qualified name of the Java class implementing the agent (**tuPInJADE.JadeShell42P**) and the path of the theory file (in the *Arguments* field visible in the GUI of Figure 4.5).

Figure 4.4: They show the two ways offered by the RMA GUI to start a new agent.

Figure 4.5: GUI that pops-up when the user creates a new agent in the container selected in the RMA GUI.

② **when launching an agent container from the command line, by adding the following string**

AGENT NAME`:tuPInJADE.JadeShell42P(`THEORY FILE`)`

AGENT NAME is a word indicating, indeed, the agent name that is local to the platform in which it is loaded; the globally unique name that JADE assigns to it is a string formed by concatenating this local name, the symbol "@" and the platform's identifier, respectively (for more details the reader is referred to JADE manuals).

THEORY FILE is the name (or the entire path when necessary) of the file containing the tuProlog theory defining the *main* predicate.

One can specify more than one agent when launching the container: it is sufficient to separate each string of this form by a blank space (as with non-tuProlog agents).

When this agent is loaded into JADE, its first action is to check if the user has inputted the name of the file that defines the ***main*** predicate: if the file has not been specified or the system has problems finding/opening/reading it, an error message is displayed by an ad-hoc window (similar to the one shown in Figure 4.1) and the agent is "killed".

If, however, the file can be opened and read by the system, then the agent checks if it begins with "`main:-`" (only lower-case letters are valid in the word *main*) and if it also contains a valid tuProlog theory.

A theory file containing the fact "`main.`" and no other definition of the *main* predicate is valid, but the corresponding agent does not ever do anything.

If any of the previous checks ends negatively, then the user is made aware of it by the above-mentioned error window, and the agent "dies".

On the other hand, if these checks end positively, the tuProlog engine is created and, by default, it contains the standard tuProlog libraries (implemented in Java):

- ***BasicLibrary:*** is implemented by the **alice.tuprolog.lib.BasicLibrary** class and allows the use of common Prolog built-in predicates, functors and operators, except for predicates regarding I/O;
- ***IOLibrary:*** is implemented by the **alice.tuprolog.lib.IOLibrary** class and contains some of the basic and common I/O predicates;
- ***ISOLibrary:*** is implemented by the **alice.tuprolog.lib.ISOLibrary** class and defines those predicates that are part of the ISO standard and not present in the previous libraries;
- ***JavaLibrary:*** is implemented by the **alice.tuprolog.lib.JavaLibrary** class and provides predicates that allow to create and/or access Java objects and classes.

These libraries are fully detailed in the "tuProlog Users Guide" that is available in the documentation provided with the tuProlog system, or in the "API docu-

mentation for tuProlog J2SE version 1.2.0" downloadable from *http://lia.deis.unibo.it/research/tuprolog/* .

The agent so far created is a "logical" one since it is able to carry out *logic reasoning* when a query is submitted to it. After adding the inference engine, the agent tries to extend it with the ***tuPInJADE.TuJadeLibrary*** (detailed in this section): if it succeeds, the agent can now communicate with any agent running in JADE platforms by using the communicative predicates that we have developed, otherwise it terminates its life cycle.

A tuProlog theory is represented by text consisting of a series of clauses and/or directives, each followed by "`.`" and the blank space.

The text file inputted to the agent and containing a tuProlog theory finally *completes* the agent since it determines its knowledge base and the rules by which it accomplishes demonstrations.

What happens now is that *every time* that JADE schedules the agent, the latter proves the "`main.`" goal. If, at any time during the agent life cycle, the resolution does not succeed, then an error message is displayed to the user.

A typical behaviour of a tuProlog agent may be to read a new message from its messages queue (automatically created by JADE for each agent), handle it and take some action (such as the update of its knowledge and/or message delivery) according to the facts and rules currently present in its theory.

The demonstration process is not visible to the programmer: to see the bindings of the variables made during the resolution, the programmer of the agent has to explicitly write the variables on the standard output or in files.

So, the behaviour of every logical agent is to prove the *main* predicate, but they can differ in the facts and clauses that rule their reasoning. Comparing them to the other kind of agents runnable in JADE, the tuProlog agents have only one activity to fulfil, that is the demonstration.

4.2.3 The JadeShell42PGui class

This class extends the JadeShell42P class: it has the same functionalities, but differs in the way the theory file is given in input to the agent.

When loading an agent implemented by JadeShell42P, the text file containing the tuProlog theory must be inputted as an *argument*, whereas this agent does not expect any input when loaded because it displays a GUI (see Figure 4.7) with which the user/administrator can browse the file system and select

from the list of files the above-mentioned file.
So, to load this agent in a JADE platform there are two ways:

① **from the RMA GUI**
When the GUI for inputting the parameters needed to create the agent is displayed (see subsection 4.2.2 for details), the only information that the user/administrator has to input are the local name of the agent and the fully qualified name of the Java class implementing the agent (**tuPInJADE.JadeShell42PGui**), as shown in the example of Figure 4.6.
The *Arguments* field, visible in the GUI of Figure 4.6, must not be filled since the theory file defining the behaviour of the agent is inputted through the GUI that appears straight afterwards (shown in Figure 4.7).

If no file is selected from such GUI, an error message is displayed and the agent "dies".

② **when launching an agent container from the command line, by adding the following string**

AGENT NAME`:tuPInJADE.JadeShell42PGui`

Figure 4.6: GUI that pops-up when the user creates a new agent using the RMA GUI.

Once the theory file has been inputted, this agent behaves exactly as a JadeShell42P agent: it solves the "`main.`" goal. If the resolution succeeds, no message is displayed on the screen and the agent solves that goal every time that it is scheduled by the JADE platform, otherwise the error that caused the failure (if detected by the tuProlog inference engine) is included in the error message.

Figure 4.7: GUI that pops-up when a JadeShell42PGui agent is loaded in JADE.

4.2.4 The TuJadeLibrary class

The previously mentioned classes, JadeShell42P and JadeShell42PGui, implement the tuProlog agent, but without the predicates defined in the TuJadeLibrary class the agent would not be able to communicate with other agents (Java, Jess or tuProlog) in JADE.

This class implements a tuProlog library defining predicates that allow the logical agents to *send/receive messages to/from other agents running in JADE.*

To develop *Java libraries* that can be loaded into the tuProlog engine, and thus added to its current theory, it is necessary to extend the ***alice.tuprolog.Library*** class, and then define appropriate methods corresponding to the desired predicates.

The method defining a predicate must have a default signature: it must return a `boolean` value, its name must be formed concatenating the predicate symbol, an underscore and the number of arguments taken by the predicate and, finally, its argument must be an object of the ***alice.tuprolog.Struct*** class.

The *alice.tuprolog.Struct* class represents a compound Prolog term or an atom: here, it is used to retrieve the arguments of the predicate when it is "solved" by the engine.
The methods (corresponding to tuProlog predicates) that we have included in this class are:

- **public boolean** send_3 (**Struct** g);
- **public boolean** receive_3 (**Struct** g);
- **public boolean** blocking_receive_3 (**Struct** g);
- **public boolean** blocking_receive_4 (**Struct** g);
- **public boolean** pack_2 (**Struct** g);
- **public boolean** unpack_2 (**Struct** g).

Besides the predicates for sending and receiving messages, there are two predicates for converting strings into tuProlog terms and vice versa.

The ordinary content of a JADE message is a *string*, even though an object can be used instead. We have chosen to make the logical agents send messages whose content is a string in order to maintain the lightweight mechanism of messaging provided by the JADE framework: this is why we have added the ***pack*** and ***unpack*** predicates.

So, on the one hand, when the tuProlog agent has to send a message, the

developer includes a call to the *pack* predicate first, and then the call to *send*.

On the other hand, when a tuProlog agent has to receive a message, the developer includes the call to the *receive* predicate and, afterwards, converts the content of the message into a tuProlog term by calling the *unpack* predicate; then, the agent will be able to reason over the received content as it usually does with ordinary terms.
We will now detail, for each predicate, their arguments and on which of these the predicates are evaluated in the `true`/`false` value:

- ***send/3***
  This predicate is the Prolog counterpart of the Java ***send*** method (available to any agent running in JADE) for sending a message. The use of the *send* method implies creating the message, "filling in" the necessary details (as one would do with an envelope) and then passing the message to the method as an input parameter. In our case, the message is automatically created and only the performative, the content and the address/addresses of the receiver/receivers are necessary.
  The call to the goal `send(Performative,Content,Receiver)` succeeds if, and only if:

  * the `Performative` term is a string that corresponds to (or a term that unifies with) one of the available performatives of an *ACLMessage* in JADE (the reader is referred to Appendix B for a complete list of such performatives);
  * the `Content` term is a string (or a term that unifies with a string) that does not contain just the Prolog anonymous variable ("_");
  * the `Receiver` term is a string/list of strings that corresponds to (or a term that unifies with) a *valid* (not necessarily corresponding to an "existing") JADE address/list of JADE addresses.

  When the goal succeeds, the agent — the one containing the tuProlog engine that evaluates the predicate — sends a message to the agent whose address is represented by `Receiver`. The content and the performative of the sent message are `Content` and `Performative`, respectively. In case `Performative` is not a valid one, the goal still is successful, but the message that is sent has the default performative **NOT_UNDERSTOOD**.

  The address of the sender is automatically inserted into the message.

If, on the other hand, the call fails, an error message is displayed by the agent.

- ***receive/3***
  This predicate is the Prolog counterpart of the Java ***receive*** method (available to any agent running in JADE) for receiving a message. The use of the *receive* method implies taking out a message from the message queue belonging to the agent and then "reading" the relevant information contained in the message; whereas, if the message queue is empty, the method returns a *null* message. In our case, when there is at least one message in the queue, it is automatically read and the performative, the content and the address of the sender are made available to the agent; on the contrary, if the queue is empty, the goal still succeeds, but its arguments remain unbound variables.
  The call to the goal `receive(Performative,Content,Sender)` succeeds if, and only if:

  * the `Performative` term is a variable not bound yet;
  * the `Content` term is a variable not bound yet;
  * the `Sender` term is a variable not bound yet;

  If the goal is successful, there are two possible situations:

  – *the message queue contained at least a message*: in this case, the `Performative`, `Content` and `Sender` variables are bound to the performative, the content and the address of the sender of the message, respectively;
  – *the message queue was empty*: the `Performative`, `Content` and `Sender` variables remain unbound.

  On the other hand, if it fails, an error message is displayed by the agent.

- ***blocking_receive/3***
  This predicate is the Prolog counterpart of the Java ***blockingReceive*** method (available to all the agents running in JADE) for blocking the agent until it reads a message from its message queue. In our case, as soon as a message is available in the queue, it is automatically "read" and the performative, the content and the address of the sender are made available to the agent.

The call to the goal `blocking_receive(Performative,Content, Sender)` if, and only if:

* the `Performative` term is a variable not yet bound;
* the `Content` term is a variable not yet bound;
* the `Sender` term is a variable not yet bound;

If the goal is successful, the `Performative`, `Content` and `Sender` variables are bound to the performative, the content and the address of the sender of the message received, respectively.

On the other hand, if it fails, an error message is displayed by the agent.

- ***blocking_receive/4***
  This predicate is the Prolog counterpart of the Java ***blockingReceive*** method (available to all the agents running in JADE) for blocking the agent, for a specified amount of time expressed in milliseconds, until it reads a message from its message queue. In our case, if a message is available in the queue, before the specified time has passed, it is automatically "read" and the performative, the content and the address of the sender are made available to the agent.

  The call to the goal `blocking_receive(Performative,Content, Sender,Msec)` succeeds if, and only if:

  * the `Performative` term is a variable not yet bound;
  * the `Content` term is a variable not yet bound;
  * the `Sender` term is a variable not yet bound;
  * the `Msec` term is a number or a term that unifies with a number;

  When the goal is successful: if the agent does not receive a message during the specified time, the `Performative`, `Content` and `Sender` variables remain unbound, otherwise they are bound to the performative, the content and the address of the sender of the message, respectively.

  On the other hand, if it fails, an error message is displayed by the agent.

- ***pack/2***
  The developer should use this predicate to convert a tuProlog term into

a string to insert it as the content inside a message: it is usually evaluated before the *send* predicate mentioned above. The call to the goal `pack(Term,StrTerm)` succeeds if, and only if the string representation of the `Term` term can be unified to the `StrTerm` term.
If it fails, an error message is displayed by the agent.

- ***unpack/2***
  The developer should use this predicate to convert a string into a tuProlog term: it is normally evaluated after one of the *receive* predicates mentioned above. The call to the goal `unpack(StrTerm,Term)` succeeds if, and only if the `StrTerm` term representing a string can be unified to the `Term` term.
  If it fails, an error message is displayed by the agent.

The implementation of the ***send*** and ***receive/blocking_receive*** predicates exploits the *send* and *receive/blocking_receive* methods available to JADE agents. Therefore, they allow communication not only between tuProlog agents, but also between *ordinary* JADE agents and tuProlog agents.

It has been possible to implement these communicative predicates thanks to the ***shell*** object, belonging to the **JadeShell42P** class and defined in this library. After this library has been loaded into the tuProlog engine of a logical agent, *shell* is the only way to refer to the agent itself from the library: so, through the *shell* object the implementation of the predicates can use the JADE communication methods.

4.3 Conclusions

This chapter has been dedicated to explaining how the *logical agents* have been integrated into DCaseLP. We have described our work by describing the tuPInJADE Java package and each class it contains.

The JadeShell42P and JadeShell42PGui classes implement the so-called *tuProlog agent* and can be compared to an *oyster without its pearl*: this oyster will be able to run in JADE as soon as it is provided with its *pearl*, that is represented by a text file containing a tuProlog theory defining the *main* predicate.

So, as it is, the behaviour of this agent is totally implemented by Prolog and the only constraint is the definition of a *main* predicate that does not take any argument. Therefore, the developer only needs logic programming skills, not familiarity with either the tuProlog or the JADE systems, but must know the predicates provided by tuProlog.

Moreover, these oysters are able to interact with the "rest of the world" since, when they are loaded in the JADE framework, the tuProlog inference engine that is part of their shell is *extended* with the tuProlog library TuJadeLibrary, containing the communicative predicates that we have developed.

These predicates use the communicative facilities offered by the underlying JADE framework; in particular, we have not developed an ad-hoc type of message, thus allowing any kind of agent running in JADE to deal with messages *from*/*to* logical agents.

Finally, these agents do a little of error handling by informing the user/administrator, through a pop-up window displaying a text message, about the failure of the demonstration of the *main* predicate or about other awkward situations.

In the next chapter we will demonstrate that tuProlog agents are able to interact with both tuProlog and JADE agents by showing a small MAS prototype.

Chapter 5

Java and tuProlog agents interacting - An example

5.1 Introduction

As mentioned before, one important requirement that our integration has to satisfy is to allow **communication**: logical agents must be able to interact not only with any other logical agent but also *with any agent running in a JADE platform*.

So, as a demonstration of the real effectiveness of our work, we have developed a working prototype of a MAS representing a simple application, *a distributed marketplace scenario*, and in this chapter we will give details about it, together with the results of its execution in JADE, thanks to the monitoring tools that it offers.

Our primary scope is to produce evidence that demonstrates that the tuProlog agents (that are the result of the work of this thesis) actually *satisfy* both the requirements submitted at the beginning of our work.

Thus, with our small example we want to show the ability of a logical agent to "reason" and to send/receive messages to/from a tuProlog agent or a different kind of agent running in DCaseLP (more precisely, running in JADE).

To demonstrate that a tuProlog agent is really capable of *logical reasoning*, it is sufficient to show that it is able to ***solve logical goals***; furthermore, by *dinamically* asserting and retracting facts or clauses in its database, the agent has a flexible behaviour.

Communication, compared to the reasoning ability, is easier to check, due to the tools that one is able to exploit in the JADE platform: one has the

possibility to "spy" the message queues of the agents and "read" the messages that they contain.

Therefore, proving that tuProlog agents can send messages to each other and that a tuProlog agent can send/receive a message to/from a Java agent, has not been such a difficult task at all.

Due to lack of time, we did not develop a prototype MAS running Java, Jess and tuProlog agents, but there seems to be no reason for it not to be possible, since the Jess agents have already been integrated into JADE [Mig02].

5.2 A distributed marketplace scenario

The MAS prototype that we have designed is as simple as possible: we have imagined that in our marketplace there are *two agents* playing the roles of "***fruit buyers***", and *three agents* playing the roles of "***fruit sellers***".
The fruit these agents are interested in are: *oranges*, *apples* and *kiwi*.
So, the five agents interacting in our MAS are:

* ***buyer1***: a **tuProlog** agent willing to buy fruit;
* ***buyer2***: another **tuProlog** agent willing to buy fruit;
* ***seller***: a **Java** agent willing to sell fruit;
* ***seller1***: a **tuProlog** agent willing to sell fruit;
* ***seller2***: another **tuProlog** agent willing to sell fruit.

The agents that sell fruit can receive two kinds of messages from those buying: these messages have the REQUEST performative, since they represent the communicative action, made by the senders, of *asking* the receivers something.

The content of these "request" messages is a term consisting of either the ***price*** or the ***buy*** predicate: in the first case, the intention of the buyer is to ask the price of the fruit that appears as argument of *price*, whereas in the second case the intention is to buy some amount of fruit, so both the amount and the kind of fruit appear as arguments of *buy*.

For simplicity, the buyers try to buy only a *fixed amount* of fruit: such amount differs for each type of fruit.

The replies made by the sellers are, obviously, different depending on the type of request they receive.

The reply to a price request is always the same: the seller sends a message, characterised by the INFORM performative, that informs the buyer (that has sent the request) about the price of the fruit.
The reply to a demand for buying is always a message with an INFORM performative, but the content depends on whether or not the seller has enough fruit to sell:

- if the *quantity* of fruit that the buyer is willing to buy is *less or equal to that possessed by the seller*, the content is the ***bought*** predicate applied to the fruit (specified as argument) that has been sold;
- if the *quantity* of fruit that the buyer is willing to buy is *more than that possessed by the seller*, the content is the ***no_more*** predicate applied to the fruit (specified as argument) that cannot be sold.

The scenario that we have implemented is the following: at the beginning, each buyer sends requests for the price of each fruit to all the sellers; once it knows the prices of the fruit, it sends requests for buying the fruit to the agents that sell that fruit at the cheapest prices.

The buyers keep sending messages requesting to buy fruit while they have money and the sellers have enough fruit to sell, whereas the sellers just reply to the messages that they receive.
We will now detail the implementation of these agents.

5.2.1 The buyer agents

As mentioned above, the two buyer agents are both logical ones so, when they are loaded into the JADE platform, they should also receive the text file defining the behaviour or, simply speaking, defining the "main" predicate and all the auxiliary user-defined predicates.

Now, we will proceed by analysing the files containing the theory for the buyer agents and explaining the meaning of the clauses and facts that they contain.

First of all, both the files must begin with the definition of the ***main/0*** predicate, otherwise an error is displayed in a window and the relative agent is "killed".
So, at the beginning of the file we find the following rule:

```
main :- handle_msgs,ask_prices,buy_goods.
```

The ***handle_msgs/0***, ***ask_prices/0*** and ***buy_goods/0*** predicates are all "custom", which means that they are not predefined in tuProlog and must, therefore, be defined somewhere inside the theory.

The facts that are initially present in the knowledge base of the agent are only those declared in these files: they infer how much fruit in stocks the agent can buy (***buys/2***), how much money it owns and it can spend (***money/1***), the price at which each fruit is sold by each seller (***price/3***) and, finally, if the agent has requested a particular seller for the price of a certain fruit (***asked/3***).

For simplicity, we also suppose that at the beginning of the scenario the agents do not have (***goods_possessed/2***) any fruit because they have not yet started to buy.

Moreover, the cost of the fruit is intended for "transaction" and not for quantity: we suppose that the seller applies a price for the fruit, regardless of how much is bought by each buyer (this is obviously not true in the real world).

In the theory file corresponding to *buyer1*, for instance, we have defined the following facts:

```
buys(goods(oranges),quantity(2)) :- true.
buys(goods(apples),quantity(3)) :- true.
buys(goods(kiwi),quantity(12)) :- true.

money(200) :- true.

price(seller,oranges,na) :- true.
price(seller1,oranges,na) :- true.
price(seller2,oranges,na) :- true.
price(seller,apples,na) :- true.
price(seller1,apples,na) :- true.
price(seller2,apples,na) :- true.
price(seller,kiwi,na) :- true.
price(seller1,kiwi,na) :- true.
price(seller2,kiwi,na) :- true.

asked(oranges,seller,no) :- true.
asked(apples,seller,no) :- true.
asked(kiwi,seller,no) :- true.
asked(oranges,seller1,no) :- true.
asked(apples,seller1,no) :- true.
asked(kiwi,seller1,no) :- true.
asked(oranges,seller2,no) :- true.
```

```
asked(apples,seller2,no) :- true.
asked(kiwi,seller2,no) :- true.


goods_possessed(oranges,0) :- true.
goods_possessed(apples,0) :- true.
goods_possessed(kiwi,0) :- true.
```

The theory file corresponding to *buyer2* contains exactly the same facts, but with different arguments, since this agent does not buy the same quantity of fruit and owns a different amount of money than *buyer1*. The term ***goods****(term)* (appearing as the first argument of the *buys* predicate) indicates a type of fruit, while ***quantity****(term)* (as second argument) is used to express how many kilograms of the relative fruit the agent buys.

The constant ***na***, used in the *price/3* predicate as third argument, has the meaning of "not available" and means that the agent is not yet aware of the price at which the fruit (specified as second argument) is sold by the agent whose name appears as first argument.

On the other hand, the constant ***no***, used in the *asked/3* predicate as third argument, has the meaning of "not asked" and means that the agent has not yet asked the price of the fruit (specified as first argument) to the agent whose name appears as second argument.

Besides these facts already mentioned, we have added the following:

```
address_name("seller@ai:1099/JADE",seller) :- true.
address_name("seller1@ai:1099/JADE",seller1) :- true.
address_name("seller2@ai:1099/JADE",seller2) :- true.
```

in order to refer to the seller agents by using the constants **seller**, **seller1** and **seller2**, instead of the strings representing their JADE addresses. As future work, we could use the *Directory Facilitator* agent available in JADE in order to obtain from it the addresses of the agents offering a certain service.

The main activities carried out by the buyer agents are, in order: handling incoming messages, asking the price of the fruit to sellers (performed until they know the price at which each fruit is sold by every agent) and buying fruit. We will now detail every single activity.

Handling messages

The code that we describe below has been used for both the buyer agents: the theory files of these agents, in fact, only differ in the arguments of some facts. The predicate that deals with the handling of the incoming messages does not take any argument and is defined as follows:

```
handle_msgs :- receive(Performative,Message,Sender),
               select(Performative,Message,Sender).
```

In the body of the clause we use the predicate ***receive/3*** that is one of the communicative predicates we have defined in the *TuJadeLibrary*.

The `Performative`, `Message` and `Sender` variables are bound only if the agent has received a message, otherwise they remain unbound. The *select* predicate has a multiple definition:

```
select(Performative,Message,Sender) :- bound(Performative),
       bound(Message),address_name(Sender,Name),
       unpack(Message,TermMsg),
       handle(Performative,TermMsg,Sender).
select(_,_,_) :- true.
```

If the agent has received at least one message, it reads the first one appearing in the queue[1]: if the sender is not one of the seller agents — in which case the `address_name(Sender,Name)` check fails — then the message is discarded, otherwise it is read and handled.

If, on the other hand, the agent does not have a message in the message queue, then it proceeds evaluating the *ask_prices* goal.

In the body of the first clause defining the *select* predicate we have used the ***unpack/2*** predicate, defined in the *TuJadeLibrary*, in order to create the `TermMsg` term corresponding to the string that is contained in the message.

The *handle/3* predicate has four clauses defining it: the first three handle one of the three different types of messages that the buyer agent can receive from

[1]The message queues belonging to the agents are directly managed by JADE.

the seller agents, while the last one is to force the success of the resolution of the goal when the agent receives a not expected message.

```
handle("INFORM",price(Goods,Price),Sender) :-
        bound(Goods),bound(Price),address_name(Sender,S),
        retract(price(S,Goods,_)),assert(price(S,Goods,Price)).
```

This clause handles the message whose content is the `price(Goods,Price)` term: it has been sent from a seller agent (whose name is `S`) to inform this agent about the price (specified by `Price`) at which it sells a fruit (specified by `Goods`).

The only action taken by this agent is to cancel the previously stored selling price, if any, of the corresponding fruit and store the price received in this message.

To do so, we retract the *price* fact whose first argument corresponds to `S` and second argument to `Goods`; since we are not interested in the old price stored, we do not specify the third argument and let it unify with anything.

After successfully retracting this fact, we assert the *price* fact whose first two arguments are still `S` and `Goods`, respectively, but the third one is `Price`, that represents the received price.

```
handle("INFORM",bought(Goods),Sender) :-
        bound(Goods),address_name(Sender,S),price(S,Goods,P),
        retract(money(M)),retract(goods_possessed(Goods,X)),
        buys(goods(Goods),quantity(Q)),N is X+Q,P \= na,
        NM is M-P,assert(money(NM)),
        assert(goods_possessed(Goods,N)).
```

This clause, instead, handles the message whose content is the `bought(Goods)` term: it has been sent from a seller agent (whose name is `S`) to inform this agent that the transaction has been successfully carried out and, therefore, this agent has bought the fruit (specified by `Goods`).

The resulting actions taken by this agent are to decrement the owned amount of money and to increment the quantity of fruit that it possesses.

To do so, we retract the `money(M)` and the `goods_possessed(Goods, X)` facts, taking care to store the old amount of owned money and of owned fruit in the M and X variables, respectively.

Afterwards, we calculate the new quantity of fruit `Goods` belonging to the buyer by adding the bought amount (given by the `Q` variable in the *buys* fact) to the X variable.

Finally, we calculate the new amount of money owned by the agent by subtracting from the M variable the selling price (given by `P` in the *price* fact) of the fruit bought (`Goods`).

```
handle("INFORM",no_more(Goods),Sender) :-
       bound(Goods),address_name(Sender,S),
       retract(price(S,Goods,_)),
       assert(price(S,Goods,finished)).
```

This last clause has the purpose of managing the message whose content is the `no_more(Goods)` term: it has been sent from a seller agent (whose name is `S`) to inform this agent that it can no longer sell a fruit (specified by `Goods`) to this agent, since the quantity it owns is less than that able to be bought by this buyer.

The only action taken by this agent is to cancel the previously stored selling price, if any, of the corresponding fruit and store one indicating that it can no longer buy from the corresponding seller agent.

To do so, we retract the *price* fact whose first argument corresponds to `S` and second argument to `Goods`; since we are not interested in the old price stored, we do not specify the third argument and let it unify with anything.

After successfully retracting such fact, we assert the *price* fact whose first two arguments are still `S` and `Goods`, respectively, but the third one is the constant `finished`, that represents the not availability of the fruit.

```
handle(_,_,_) :- true.
```

If the received messages do not have the expected performative (INFORMA-

TIVE), then they will be discarded.

Now, we will analyse how the next activity of the buyer agents is implemented.

Asking prices of the fruit

We have added two more facts to the theory to make the next activities easier: they state the list of fruit available in the market and the list of JADE addresses corresponding to the seller agents. We present them here, even though they are part of the initial knowledge of the buyer agents, because they are called by the goals implementing the next activities.

```
goods_list([oranges,apples,kiwi]) :- true.

sellers_addresses(["seller@ai:1099/JADE",
                   "seller1@ai:1099/JADE",
                   "seller2@ai:1099/JADE"]) :- true.
```

The predicate that "implements" the activity of asking the fruit prices, *ask_prices/0*, relies on the *ask_prices/2* and *ask_prices_to_seller/2* predicates to accomplish the task of sending the requests, to each seller agent, for the prices charged for fruit.

```
ask_prices :- sellers_addresses(Sell),goods_list(GList),
              ask_prices(Sell,GList).

ask_prices([Seller|Others],GoodsList) :-
           ask_prices_to_seller(Seller,GoodsList),
           ask_prices(Others,GoodsList).

ask_prices([],_) :- true.

ask_prices_to_seller(Seller,[Goods|Others]) :-
           address_name(Seller,S),asked(Goods,S,no),
           pack(price(Goods),Str),send("REQUEST",Str,Seller),
           retract(asked(Goods,S,no)),
           assert(asked(Goods,S,yes)),
           ask_prices_to_seller(Seller,Others).
```

The initial action performed by the first clause defining the *ask_prices_to_seller* goal is to check if the price of a given fruit has already been asked to the seller agent being dealt with (whose name is `S`).

This check is done by evaluating the *asked* predicate on the fruit currently in consideration (`Goods`), the seller agent (`S`) and the constant `no`, respectively. If this goal succeeds, it means that the agent has yet not issued the price request and must, therefore, do it now.

So, it evaluates the *pack* goal in order to convert the `price(Goods)` into a string (`Str`) that will become the content of the message to send to the seller agent. The message requesting the price of the fruit is then sent by calling the `send("REQUEST",Str,Seller)` goal of the *TuJadeLibrary* that we have developed.

The third argument on which the *send* predicate is evaluated is `Seller` and not `S`, since it must be the JADE address of the agent receiving the message and not its name.

Before sending the message, the agent must convert the `price(Goods)` term into a string in order to insert it as the content of the JADE message: this is why the *pack* predicate is evaluated.

If the message is successfully sent, the agent must keep track of such action in order not to issue the request again. To do this, the `asked(Goods,S,no)` fact is retracted and replaced with `asked(Goods,S,yes)` by the *assert* predicate.

At this point, the agent passes to the next fruit sold by the same seller agent.

```
ask_prices_to_seller(Seller,[Goods|Others]) :-
            address_name(Seller,S),price(S,Goods,X),X \= na,
            ask_prices_to_seller(Seller,Others).
```

This clause has the purpose of checking whether the price of the given fruit is available or not. If the price is already present in the knowledge base of the agent, namely if `X` does not unify to the `na` constant, the agent passes to the

next fruit in the list of those available.

```
ask_prices_to_seller(Seller,[Goods|Others]) :-
          address_name(Seller,S),asked(Goods,S,yes),
          ask_prices_to_seller(Seller,Others).
```

On the other hand, if the price is still not present but it has been requested, the agent cannot do anything regarding that fruit so, while waiting for the agent seller to reply, it can check if it knows the price of the other fruit (by calling the `ask_prices_to_seller(Seller,Others)` goal).

```
ask_prices_to_seller(_,[]) :- true.
```

This final clause is used to stop requesting fruit prices to a certain seller agent.

Buying fruit

The last activity, though still the one characterising this type of agent, is the one of buying fruit. It is implemented by the *buy_goods/0* predicate that, in analogy with the *ask_prices* predicate, assigns the work to another predicate: *buy_goods/2*. The latter one, on the other hand, relies on the *find_min_price/5* and the *buy_from_seller/3* predicates to carry out its job.

```
buy_goods :- sellers_addresses(SList),goods_list(GList),
             buy_goods(SList,GList).
```

Instead of seeing straightaway the *buy_goods/2* predicate, we must explain the *find_min_price* predicate, since it is used in the definition of the former.

This predicate is meant to "find" the seller agent that sells a given fruit (`Goods`) at the cheapest price: after the predicate has been evaluated, that agent will be represented by the fifth argument, though it will not unify to the name of the agent but to its JADE address.

```
find_min_price([Seller|Others],Goods,Min,_,HonestSeller) :-
       address_name(Seller,S),price(S,Goods,Price),
       Price \= na,Price \= finished,
       Price < Min,
       find_min_price(Others,Goods,Price,Seller,HonestSeller).
```

The first action performed by this clause is to check if the agent already knows the price at which the given fruit is sold by the "examined" seller agent; the seller agent currently "examined" is the one appearing as first element in the list that is the first argument of the predicate. This check is easily performed by solving the *price* goal.

Finally, if the buyer knows the price, the latter (`Price`) is compared to the current minimum found (the `Min` term appearing as third argument of the predicate): if it is less than the current minimum, then the predicate is evaluated again, but this time the current seller is discarded from the seller list. The seller agent discarded from the list is kept, as the fourth argument, because it is the cheapest till now.

```
find_min_price([Seller|Others],Goods,Min,Sel,HonestSeller) :-
       address_name(Seller,S),price(S,Goods,Price),
       Price \= na,Price \= finished,Price >= Min,
       find_min_price(Others,Goods,Min,Sel,HonestSeller).
```

This clause, instead, deals with the situation in which the price charged by the examined seller is higher or equal to the minimum present as third argument (`Min`): in this case, this seller is totally ignored and the sellers remaining in the list are examined.

```
find_min_price([Seller|Others],Goods,Min,Sel,HonestSeller) :-
       address_name(Seller,S),price(S,Goods,na),
       find_min_price(Others,Goods,Min,Sel,HonestSeller).
```

In the case in which this agent does not yet know the price, the seller agent currently examined is discarded and the search is continued with the remaining sellers in the list.

```
find_min_price([Seller|Others],Goods,Min,Sel,HonestSeller) :-
       address_name(Seller,S),price(S,Goods,finished),
       find_min_price(Others,Goods,Min,Sel,HonestSeller).
```

Finally, this clause is the one taking into account the case in which the examined seller agent has finished the desired fruit, so it is eliminated from the list and the search carries on between the remaining sellers.

```
find_min_price([],_,_,Seller,Seller) :- true.
```

This clause is the one that terminates the search since all the seller agents have been examined: the agent selling the desired fruit at the cheapest price, if there is one, is represented by the last argument of the predicate.

At this point, it should be easier to understand the *buy_goods/2* predicate whose definition is as follows:

```
buy_goods(SList,[Goods|Others]) :- money(M),M > 0,
          buys(goods(Goods),quantity(Q)),
          find_min_price(SList,Goods,30000,nobody,HonestSeller),
          HonestSeller \= nobody,
          buy_from_seller(HonestSeller,Goods,Q),
          buy_goods(SList,Others).

buy_goods(_,_) :- true.
```

The first action taken by this predicate is, of course, to check if the agent has money, otherwise it does nothing.

So, once the agent has money to spend to buy fruit, the next step is to consider each fruit in the list of those available in the market and looking for the seller that charges the cheapest price for the fruit considered. Here, the *find* goal to solve has the number 30000 representing the cheapest selling price for any fruit: this is because we suppose that the seller agents will sell at a lower price, thus it will always be possible to find the agent charging the cheapest price, if there is one.

If the search terminates with the `HonestSeller` term unifying to the `nobody` constant, then the agent will try and see if it can buy another fruit and starts with the next fruit belonging to the list of those available in the market.

On the other hand, if an agent selling the fruit at a cheap price is found, the transaction is left up to the *buy_from_seller* predicate and the agent tries to buy the other fruit in the list.

```
buy_from_seller(Seller,Goods,Quantity) :-
          address_name(Seller,S),price(S,Goods,Price),
          money(M),M > 0,NM is M - Price,NM >= 0,
          pack(buy(Goods,Quantity),Str),
          send("REQUEST",Str,Seller).
```

Before sending a message requesting to buy the desired fruit, the agent must check if the money it has is enough; so, it subtracts the price of the fruit (`Price`) from the money it has (`M`), and carries on only if the result is positive or equal to zero.

Finally, by resolving the *send* predicate, the agent sends the seller agent the message with which it requests to buy a certain quantity (specified by `Quantity`) of the fruit indicated by `Goods`.

```
buy_from_seller(_,_,_) :- true.
```

Thanks to this last clause, if the money is not enough for buying the fruit, the agent will pass to the next fruit belonging to the list. We must remind the reader that if a goal in the `main` predicate fails, an error is displayed in a window and the agent is "killed", so this last clause is also necessary to force the `buy_from_seller` goal never to fail.

5.2.2 The seller agents

Two of the seller agents, precisely **seller1** and **seller2**, are logical ones, while **seller** is an ordinary Java agent. We will first look at the files containing the tuProlog theory for the logical seller agents and, afterwards, at the Java code that implements the agent **seller**.

The logical seller agents

Both the theory files inputted to these tuProlog agents (when loaded into JADE) start with the following definition:

```
main :- handle_msgs.
```

Before seeing the clauses that make up the ***handle_msgs/0*** predicate, we

will list the facts that are initially present in the knowledge base of these agents.

These facts define how much fruit these agents sell (***sells/3***), including the price and the quantity they have, and if a certain fruit can be sold to a particular buyer (***available/3***).

The *available* predicate has been introduced in order to avoid useless calculations to these agents: if a buyer agent tries to buy a certain fruit and it appears that the quantity sold by this agent is less than the quantity that the buyer buys, then the agent is capable of remembering this information by storing it in its knowledge base.

By doing this, next time the same buyer asks to buy the same fruit, the agent knows that the quantity it has is less than the requested and can deny the transaction immediately.

For simplicity, we also define facts (*address_name/2*) that are used to convert the addresses of the buyer agents into their names and vice versa.

In the theory loaded into **seller1**, for instance, we have declared the following:

```
sells(goods(oranges),price(30),quantity(900)) :- true.
sells(goods(apples),price(150),quantity(40)) :- true.
sells(goods(kiwi),price(200),quantity(30)) :- true.

available(oranges,buyer1,yes) :- true.
available(oranges,buyer2,yes) :- true.
available(apples,buyer1,yes) :- true.
available(apples,buyer2,yes) :- true.
available(kiwi,buyer1,yes) :- true.
available(kiwi,buyer2,yes) :- true.

address_name("buyer1@ai:1099/JADE",buyer1) :- true.
address_name("buyer2@ai:1099/JADE",buyer2) :- true.
```

The theory file corresponding to *seller2* contains exactly the same facts, but with different arguments, since this agent does not have the same quantity of fruit and sells at different prices than *seller1*. The term ***goods****(term)* indicates a type of fruit, while ***quantity****(term)* is used to express how many kilograms of the relative fruit the agent has, and ***price****(term)* to state the price at which the fruit is sold.

The constant ***yes***, used in the *available/3* predicate as third argument, has the meaning of "it is available" and means that the agent can sell the fruit (specified as first argument) to the buyer agent whose name appears as second argument.

The only activity carried out by the seller agents is the one of handling incoming messages:

```
handle_msgs :- receive(Performative,Message,Sender),
               select(Performative,Message,Sender).
```

By evaluating the *receive* predicate, the agents extract a message from their message queue: if the queue is empty, then the `Performative`, `Message` and `Sender` variables are unbound, otherwise they are bound to the performative, the content and the address of the sender of the message, respectively.

The select predicate is declared as follows:

```
select(Performative,Message,Sender) :- bound(Performative),
       bound(Message),address_name(Sender,S),
       unpack(Message,TermMsg),
       handle(Performative,TermMsg,Sender).

select(_,_,_) :- true.
```

If the agent has received a message, it reads the sender: if the sender is not one of the buyer agents then the message is discarded, otherwise it is read and handled.

In order to be able to manage the content of the received message, the agent must convert it into a term: it does so by solving the *unpack* goal.

The *handle/3* predicate has four clauses defining it: the first three handle the two types of messages that the seller agent can receive from the buyer agents, while the last one is to force the success of the resolution of the goal when the agent receives an unexpected message.

```
handle("REQUEST",price(Goods),Buyer) :- bound(Goods),
       sells(goods(Goods),price(Price),quantity(AQ)),
       pack(price(Goods,Price),MsgString),
```

```
        send("INFORM",MsgString,Buyer).
```

The purpose of this clause is to handle the message whose content is the `price(Goods)` term: it has been sent from a buyer agent (whose address is `Buyer`) to ask this agent the price at which it sells the fruit specified by `Goods`.

The only action taken by this agent (in reply to this message) is to send back a message to the buyer whose content is the string obtained by "packing" the `price(Goods,Price)` term.

```
handle("REQUEST",buy(Goods,Quantity),Buyer) :-
        bound(Goods),bound(Quantity),address_name(Buyer,B),
        available(Goods,B,yes),
        sells(goods(Goods),price(Price),quantity(AQ)),
        AQ >= Quantity,
        retract(sells(goods(Goods),price(Price),quantity(AQ))),
        NQ is AQ - Quantity,
        assert(sells(goods(Goods),price(Price),quantity(NQ))),
        pack(bought(Goods),MsgString),
        send("INFORM",MsgString,Buyer).
```

This clause, instead, handles the message whose content is the `buy(Goods,Quantity)` term: it has been sent from a buyer agent (whose address is `Buyer`) to ask this agent to buy the fruit specified by `Goods`. The `Quantity` term indicates to this agent the amount of fruit that the buyer agent is interested in buying.

The first action performed by the agent is to check if it can sell fruit to the buyer: this is done by solving the *available* goal.

If the fruit can be sold, the agent verifies if the quantity it has is greater or equal to the one requested: if it is, the fruit can be sold, so the transaction is carried out.

On the other hand, if the agent "knows" that it cannot sell this fruit to this particular buyer agent, then it just ignores the message.

The amount of fruit sold (`Quantity`) is subtracted from the quantity owned (`AQ`) before the transaction, and the new amount (`NQ`) that can be sold

is stored in the knowledge base of the agent.

```
handle("REQUEST",buy(Goods,Quantity),Buyer) :-
       bound(Goods),bound(Quantity),address_name(Buyer,B),
       available(Goods,B,yes),
       sells(goods(Goods),price(_),quantity(AQ)),
       AQ < Quantity,retract(available(Goods,B,yes)),
       pack(no_more(Goods),MsgString),
       assert(available(Goods,B,no)),
       pack(bought(Goods),MsgString),
       send("INFORM",MsgString,Buyer).
```

This clause is the one managing the case in which the fruit requested by the buyer agent is more than what the agent has.

In this case the agent sends a message, whose content is the `no_more(Goods)` term, to the buyer interested in buying the fruit.

Before sending the message, though, the agent stores in its knowledge base the fact `available(Goods,B,no)`, where the term `no` is here used to indicate "not possible" to sell to the buyer (whose name is `B`) the fruit (`Goods`).

```
handle(_,_,_) :- true.
```

This last clause has the purpose of forcing the success of the resolution of the goal in the cases not dealt with by the previous clauses.

The Java seller agent

We will now detail the main parts of the content of the file *Seller.java* that defines the *Seller* class and implements the agent ***seller*** of our example:

```
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
```

As any other Java class defining an agent running in JADE, this class extends the ***jade.core.Agent*** class.

Besides the *Agent* class, two more classes are imported from the *jade.core*

package of JADE: the ***AID*** class, whose instances are JADE addresses, and the ***CyclicBehaviour*** class (defined in the ***behaviours*** subpackage), whose instances are tasks that will be continuously repeated if added to the task queue of the agent.

This behaviour "cycles": once its ***action*** method is completed, it is not discarded from the task queue, as usually happens with ordinary behaviours; it is put at the end of the queue where it will be executed when its turn comes round again.

For simplicity, from now on we will use "seller agent" or simply "agent" to refer to an instance of the Seller class.

```
public class Seller extends Agent
{ private int oranges = 5;
  private int apples = 5;
  private int kiwi = 10;
```

These three integer fields, named with the name of the fruit, are meant to store the quantity of fruit that the agent has: initially, the agent has 5 kilograms of oranges, 5 kilograms of apples and 10 kilograms of kiwi.

```
  private int orangesP = 105;
  private int applesP = 80;
  private int kiwiP = 100;
```

The three above listed integer fields, whose name is formed by the name of the fruit and an ending ***P***, are meant to store the price at which the agent sells the fruit: initially, the price charged by the agent for oranges is 105, for apples is 80 and for kiwi is 100.

```
  private boolean orToB1 = true;
  private boolean orToB2 = true;
  private boolean apToB1 = true;
  private boolean apToB2 = true;
  private boolean kiToB1 = true;
  private boolean kiToB2 = true;
```

These boolean fields are used to keep track as to whether a given fruit is (or is not) sellable to the buyers. The first two letters in the name of the field indi-

cate the fruit: `or` stands for oranges, `ap` for apples and `ki` for kiwi. Whereas, the last two letters represent the buyer: `B1` stands for *buyer1*, while `B2` stands for *buyer2*. If, for instance, the value of the `apToB2` field is `false`, then it means that this agent cannot sell **ap**ples to **b**uyer**2**. At the beginning, the agent can sell all the fruit to all the buyers.

```
protected void setup()
{ SellBehaviour p = new SellBehaviour(this);
  addBehaviour(p);
}
```

The *setup* method is invoked by JADE when loading the agent and usually the developers insert the necessary code into it to "setup" the execution of the agent. The only action that we have programmed during the setup phase is that of adding a custom behaviour, the ***SellBehaviour***, to the task queue.

```
class SellBehaviour extends CyclicBehaviour
{ ....................
  public void action()
  { ACLMessage msg;
    msg = myAgent.receive();
    if (msg != null) handleMsg(msg);
  }
```

The *SellBehaviour* behaviour is the only task that the agent has to perform but, since it extends the CyclicBehaviour available in JADE, it is repeated throughout the life of the agent, unless explicitly removed from the queue by the agent.

The *action* method in *SellBehaviour* is invoked by JADE's scheduler when it "withdraws" such behaviour from the task queue corresponding to this agent. So, the main task of this seller agent is to receive messages: if the message queue is empty, it remains idle, otherwise it handles the received message by calling the ***handleMsg*** method that we have developed, passing the message to it as parameter.

```
private void handleMsg(ACLMessage msg)
{ String sender = msg.getSender().getLocalName();
  if ((sender.equals("buyer1"))||(sender.equals("buyer2")))
```

```
    { CODE1 }
    else return;
  }
```

The first action carried out by this method is to "extract" the *name* of the sender agent: if it is not one of the buyer agents, then it returns immediately, otherwise it executes **CODE1** below:

```
{ String perf = msg.getPerformative();
  String cont = msg.getContent();
  int n = cont.length();
  if (!perf.equals("REQUEST")) return;
  try if (cont.substring(0,5).equals("price"))
  { CODE2 }
  else
  if (cont.substring(0,3).equals("buy"))
  CODE3
```

The performative and content of the message are read. First, the performative is checked: if it is not the REQUEST performative, the method returns. So, if the performative is the one expected, the method proceeds by checking the content: if it is a string that does not start with the word `price` or `buy`, then it returns without doing anything.

On the other hand, if the content begins with the word `price`, **CODE2** below is executed:

```
{ String goods = cont.substring(6,n-1);
  int price = price(goods);
  if (price == -1) return;
  msg.reset();
  msg.setPerformative(ACLMessage.INFORM);
  msg.setSender(myAgent.getAID());
  AID receiver = new AID();
  receiver.setLocalName(sender);
  msg.addReceiver(receiver);
  String content = new String("price("+goods+","+price+")");
  msg.setContent(content);
  myAgent.send(msg);
```

```
}
```

If the content of the message is the string `price(`*elem*`)`, with the *elem* string equal to `oranges` or `apples` or `kiwi`, then the agent sends a message to the sender. This message has the INFORM performative and the content "`price(`*elem*`,`*pr*`)`", with the *pr* string equal to a number corresponding to the selling price of *elem*.

In the above code, the private *price* method is called: it takes a String object as an argument and returns an integer. The returned integer is -1 if the string in input is not one of the fruit in the market, otherwise it is the value of the field corresponding to the price of that fruit.

When the content of the message received by the agent begins with the word `buy`, then **CODE3** below is executed:

```
{ int comma = cont.indexOf(',');
  String goods = cont.substring(4,comma);
  if (n < comma+1) return;
  int qtity = Integer.parseInt(cont.substring(comma+1,n-1));
  if (qtity < 0) return;
  if (!available(goods,qtity))
  { if (!reported(goods,sender))
    { msg.reset();
      msg.setPerformative(ACLMessage.INFORM);
      msg.setSender(myAgent.getAID());
      AID receiver = new AID();
      receiver.setLocalName(sender);
      msg.addReceiver(receiver);
      String content = new String("no_more("+goods+")");
      msg.setContent(content);
      reporting(goods,sender);
      myAgent.send(msg);
}
```

The first four lines of **CODE3** are for "parsing" the content of the message (a string representing the *buy(goods,qtity)* term) and obtaining from it the variables representing the fruit (`goods`) and the quantity ({code) that the buyer agent wants to buy.

If the quantity of fruit that the buyer has specified in the message is not positive, the agent does nothing.

If, on the contrary, it is positive, then the agent checks if it has enough fruit to sell: it calls the private *available* method that, given the name of the fruit and the requested quantity, returns `true` only if the agent has enough fruit to sell.

If it has less fruit than the quantity requested by the buyer: it calls the private *reported* method that, given the name of the fruit and the name of the buyer agent, returns `true` only if the agent "believes" it can sell that fruit to that buyer.

If the *reported* method returns `true`, it means that this is the first time that the agent "realises" that it cannot sell the fruit to the buyer. As a result, it not only updates the boolean value of the relative field, but also sends a message to the buyer.

This message informs the buyer that, from now on, the seller will not be able to provide it with the requested fruit.

When the *reported* method returns `false`, the agent does nothing.

The last case to analyse is the one corresponding to the message containing the request for an amount of fruit available and that implies that the transaction is carried out. This is the code executed in this case:

```
{ update_qtity(goods,qtity);
  msg.reset();
  msg.setPerformative(ACLMessage.INFORM);
  msg.setSender(myAgent.getAID());
  AID receiver = new AID();
  receiver.setLocalName(sender);
  msg.addReceiver(receiver);
  String content = new String("bought("+goods+")");
  msg.setContent(content);
  myAgent.send(msg);
}
```

The first command is the call to the private *update_qtity* method that, given the name of the fruit and the number representing the quantity that the buyer wants, updates the field containing the quantity of fruit possessed by the agent. Afterwards, the agent sends the buyer the message informing it that the fruit has been bought.

5.3 Executing the prototype

Once all the agents have been specified using tuProlog and Java, they can finally be loaded into JADE and we can start executing the obtained prototype. JADE offers more than one way to debug the MAS.
For instance, the developer can exploit the ***Sniffer Agent***, that is an agent "enriched" with so-called *sniffing features*: it displays the messages exchanged by agents selected by the user, as a sort of sequence diagram.
The Sniffer Agent can be started from the RMA GUI or from the command line. To launch it from the RMA GUI, the user can either click on the ***Tools*** menu and then select the ***Start Sniffer*** item from the displayed menu (shown in Figure 5.1), or directly press the appropriate button from the menu bar (shown in Figure 5.2).

Whereas, to start it from the command line, the user must load into the platform an agent implemented by the `jade.tools.sniffer.Sniffer` Java class.

This agent is an **ad-hoc GUI** from which the user can **sniff** an agent or a group of agents belonging to the MAS: *every message* sent/received to/from the target agent or group is observed and displayed in the GUI (see Figure 5.3).

In the window on the right of the GUI in Figure 5.3, the user can view the "sniffed" agents and the flow of messages between them: the box labelled "Other" represents all the agents of the platform that are not currently sniffed, while the other boxes are labelled with the name of the sniffed agent.
Each labelled arrow represents a message sent from the agent at the tail of the arrow to the agent at the point of the arrow.

Figure 5.1: One way to launch the Sniffer agent from the RMA GUI.

Figure 5.2: The other way to launch the Sniffer agent from the RMA GUI.

Figure 5.3: The GUI representing the Sniffer Agent; as an example, we have chosen to "sniff" the RMA, DF and ams agents.

Figure 5.4: How to view the content of the "sniffed" messages.

To view the details of a message, right-click on the arrow representing this message then click the ***View Message*** box that appears, like the one shown in Figure 5.4.

After clicking *View Message*, a window appears, as shown in Figure 5.5: it lists the content, the receiver, the sender, the ontology, the language of the message, and other information.
The Sniffer Agent not only allows one to **view** every message, but also to **save** it (or all of them) to a file that can be reloaded in this GUI at a later time.

Another interesting agent that is useful in debugging the MAS is the ***Introspector Agent***. This agent is a GUI that, as the name suggests, allows a sort of "introspection" of the single agents.

To start this agent from the RMA GUI, the user can either press the appropriate button or click on the ***Tools*** menu and then select the ***Start Introspector Agent*** item from the displayed menu (both ways are shown in Figure 5.6).

One can monitor and control the life cycle of a running agent, and also view the messages that it has sent and received. The user is able to look at the queues of messages of the agent: there are the ***Incoming Messages*** and the ***Outgoing Messages*** panels.

Figure 5.5: The Sniffer Agent also allows to "sniff" the details regarding the message.

Figure 5.6: It shows the two ways in which the Introspector Agent can be started from the RMA GUI.

Figure 5.7: The Introspector Agent.

In the Incoming Messages panel are listed the ***Pending*** and ***Received*** messages, labelled by their performative; by right-clicking on one message, the user can view it, remove it or remove all the messages (see Figure 5.7).

In the Outgoing Messages panel, however, the ***Pending*** and ***Sent*** messages are listed.

The Introspector Agent also offers the possibility to monitor the queue of *behaviours* of an agent, including their execution step-by-step.

5.3.1 The results

By executing the five agents of our marketplace, we have obtained what we expected. We verified that the agents could really communicate by launching the Sniffer Agent and the Introspector Agent.
In Figure 5.8, the reader can see that the Sniffer Agent has sniffed many messages that were exchanged between the agents belonging to the prototype. Figure 5.9 has been obtained from Figure 5.8 and then enlarged, in order for the reader to view it better.

During the execution of the MAS, we have been able, thanks to the functionalities offered by JADE, to check if the agents behaved as they were intended to, even though they were randomly started by the JADE platform.

The Sniffer Agent allowed us to verify that the messages were exchanged in

Figure 5.8: A screenshot of the Sniffer Agent while running in the platform of our prototype.

Figure 5.9: A part of Figure 5.8 that we have enlarged so the reader can view it better.

Figure 5.10: Details of a message sent by *buyer1* to *seller* and whose content is `price(kiwi)`.

the expected order (for example, that all the buyers asked for the price of fruit first, and only started buying fruit afterwards), while the detailed views of the messages allowed us to verify that the content of the replies were consistent with the content of the requests.

By viewing the messages, both with the Sniffer Agent and the Introspector Agent, we have gathered that the messages sent from the buyers requesting to buy fruit were actually received by the seller agents: the messages with the INFORM performative sent from the sellers to the buyers are the proof of it.

Figure 5.10 shows the request for the price of kiwi sent from *buyer1* to *seller*.

Moreover, we have been able to test the proper working of the ***send*** and ***receive*** predicates that we have developed, and not only between tuProlog agents but also between a tuProlog and a Java agent.

In our example, in fact, the agent *seller* (that was the only Java agent) has been able to receive messages from the buyers and send replies to them.

The "error handling" that we have added to our agents has been tested as well; for example: we tried to load tuProlog agents without inputting the theory files, then we inputted files that did not contain a tuProlog theory, we deliberately caused the failure of a goal in order to force the failure of the `main` goal, and so on.

5.4 Conclusions

In this chapter we have given a small example of a prototype of a MAS consisting of two different types of agents: logical and obect-oriented ones.
The logical agents have been specified by using the tuProlog language, while those object-oriented by the Java language.

This prototype was then executed in the DCaseLP environment and monitored to check it behaved as we expected it to.

Its execution has demonstrated that the tuProlog agents are not only capable of communicating with other agents running in JADE, but also to logically "reason". In our example we have forced these logical agents to dinamically modify their behaviour and their knowledge base as the result of interacting with other agents.

All our expectations in terms of behaviour of the MAS were, at the end, met.

Additionally, an important issue has been made clear in this example: in

order *to program the tuProlog agent the developer only needs Prolog skills* and does not have to know the Java programming language.

Due to lack of time and the fact that the first release of DCaseLP [Mig02] used an older version of JADE than the one used by us, we did not include the Jess agents in our example, but we are confident of the fact there is no reason why these agents should not correctly interact with the tuProlog agents.

Our confidence is justified by an important feature regarding the logical agents: the messages they exchange are those available in JADE and not ad-hoc or specialised ones.

Conclusions

The work we have described in this thesis is a step forward towards the engineering of *heterogeneous agents* and *multilingual agent systems* by the integration of "logical agents" into the existing prototyping environment DCaseLP.

More precisely, we wanted to add a logic language to DCaseLP in order to offer the developers the possibility to choose between Java, Jess or this logic language when implementing an agent that will run in the prototype of the MAS, thus "incrementing" the multilinguality of our software prototyping tool.

Since the beginning, we intended to integrate a Prolog language because, besides the interesting features offered by this language, it was also our intention to reuse the Prolog-based code and instruments already developed for CaseLP, the predecessor of DCaseLP. Furthermore, an agent implemented by a logic programming language naturally features a *reasoning* ability since it is automatically able to carry out *demonstrations* of logic goals.

We chose tuProlog between various Java-based Prolog languages available since it seemed the one meeting all our requirements.

The integration has been carried out mainly by creating a Java package to add to our environment and containing the files defining the classes necessary to create what we call a *tuProlog agent*. The classes that we have defined can be compared to a *shell* of an oyster — the oyster in question represents the agent programmed in tuProlog and running in the DCaseLP environment —, whereas the pearl of the oyster is the tuProlog theory defining the behaviour and the initial state of the agent.

As an oyster without a pearl is not considered valuable, the same holds for a tuProlog agent: the developer must, when loading this agent into the prototype, provide it with a text file containing the theory that will be used by the inference engine incorporated in the agent. There is only one constraint on this file: it must begin with the definition of the `main/0` predicate.

The reason of this constraint is easily explained. When a tuProlog agent

is executed in the prototype, it automatically solves the goal `main.` and will always take this action everytime that it is scheduled by the JADE platform in which it runs.

Additionally, we have implemented a simple form of error handling relative to the execution of logical agents, since the JADE platform does not provide it: any time that an awkward situation occurs (like the failure of the goal `main.`) while a logical agent is running, a window containing an error message is diplayed and the agent terminates its execution.

It is responsibility of the programmer to ensure that the `main` predicate will not fail in situations that are not erroneous.

Normally, a developer of Prolog agents has mainly two ways of tracing the communication among the developed agents during the execution of the prototype:

- put breakpoints into the code, or
- write messages on the standard output or on a file.

CaseLP, the predecessor of DCaseLP, offers more sophisticated graphical debugging tools than this "by-hand" inspection, but adopting the instruments already present in a standard, FIPA-compliant and open-source platform, like JADE, represents an improvement with respect to the use of the proprietary instruments offered by CaseLP.

DCaseLP offers graphical tools (implemented in the JADE platform) to verify the correct interaction between agents running in the prototype, therefore, we have also achieved to allow the programmer of logical agents to exploit such tools and carry out some form of monitoring and debugging.

CaseLP is implemented in SICStus Prolog, extended with communicative primitives, so it is not possible to directly "port" agents developed with CaseLP into a MAS developed with DCaseLP. In order to fully achieve the compatibility between CaseLP and DCaseLP agents, the translator "*SICStus Prolog → tuProlog*" that we did not implement due to lack of time could be included in future work.

Another direction of future work could be the physical integration of DyLOG, a language based on modal logic and by means of which it is possible to reason about actions and change, since we have already analysed a methodological integration into DCaseLP of this agent programming language.

Besides being another candidate for implementing agents in DCaseLP, DyLOG will allow us to formally verify properties of the communication protocols to which the agents must adhere.

Reasoning about interaction protocols means to be able to check if a certain set of properties holds after a conversation has taken place. This can, for instance, allow to determine which protocol, from a set of available ones, satisfies a goal of interest, or it can help to find out which protocols to combine in order to accomplish more complex tasks.

Moreover, adding the DyLOG language into DCaseLP, would offer the developer the possibility of converting UML/AUML sequence diagrams into a DyLOG program, automatically verify if the latter is consistent with the AUML specifications and if it satisfies certain properties.

As soon as a Java interpreter of DyLOG programs will be available (it is an ongoing work carried out by the designers of DyLOG) the physical integration of DyLOG into DCaseLP should not be too difficult: the tuProlog agents could be exploited as an interface.

Finally, we would like to verify the full compability between Jess and tuProlog agents running in a prototype developed with the use of the DCaseLP environment.

Appendix A

The tuPInJADE Java package

This Appendix lists the classes defined in the tuPInJADE Java package that has been developed to realise the integration of ***tuProlog agents*** in DCaseLP. This package contains four Java files;

ErrorMsg.java

```
package tuPInJADE;

import java.awt.Dialog;
import java.awt.Frame;
import java.awt.FileDialog;
import java.awt.TextArea;
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

public final class ErrorMsg implements WindowListener
{  public Frame box = new Frame();
   public final Dialog error = new Dialog(box);
   public final TextArea text = new TextArea("",5,150,
                                TextArea.SCROLLBARS_NONE);
   public boolean done = false;

   public ErrorMsg () { super();}

   public void write(String title,String msg)
   { box.setSize(350,80);
     box.setLocationRelativeTo(null);
     error.setModal(true);
     error.setSize(350,80);
     error.setResizable(false);
     error.add(text);
     error.setLocationRelativeTo(null);
```

```
      error.setTitle(title);
      error.addWindowListener(this);
      text.setText(msg);
      text.setEditable(false);
      text.setVisible(true);
      error.setVisible(true);
      error.show();
      box.toFront();
      box.addNotify();
      box.pack();
    }

    public void windowOpened(WindowEvent e){ }

    public void windowClosing(WindowEvent e)
    { box.removeNotify();}

    public void windowClosed(WindowEvent e)
    { box.removeNotify();}

    public void windowIconified(WindowEvent e){ }

    public void windowDeiconified(WindowEvent e){ }

    public void windowActivated(WindowEvent e){ }

    public void windowDeactivated(WindowEvent e){ }
}
```

JadeShell42P.java

```
package tuPInJADE;

import alice.tuprolog.InvalidLibraryException;
import alice.tuprolog.InvalidTheoryException;
import alice.tuprolog.Library;
import alice.tuprolog.MalformedGoalException;
import alice.tuprolog.NoMoreSolutionException;
import alice.tuprolog.NoSolutionException;
import alice.tuprolog.OutputEvent;
import alice.tuprolog.OutputListener;
import alice.tuprolog.Prolog;
import alice.tuprolog.SolveInfo;
import alice.tuprolog.Theory;
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.StreamTokenizer;

// The JadeShell42P class is a JADE agent that behaves as a "shell"
// for a tuProlog agent incorporating a tuProlog engine.
// This agent, when loaded in JADE, needs in input the name of a file
// containing a tuProlog theory that defines the "main/0" predicate that
// represents the behaviour of the agent.
// If the theory file cannot be opened or does not contain a correct
// tuProlog theory, the user is made aware of it by the popup of a window
// displaying an error message.
// If,on the contrary, the theory file is correct, the agent loads into
// its tuProlog engine the standard tuProlog libraries and then
// extends it by loading the TuJadeLibrary as well.
// The TuJadeLibrary is indispensable to a tuProlog agent because
// it provides the means of communication with any other agent in JADE,
// including other tuProlog agents of course.
// The TuJadeLibrary defines predicates whose goal is, respectively,
// sending/receiving ACL messages to/from agents in a JADE platform.
//
public class JadeShell42P extends Agent
{  public ErrorMsg err;
   protected FileInputStream f;
   protected FileReader fr;
   protected Prolog core;// the Prolog engine of the agent
   protected StreamTokenizer in;
   protected String fileName;
```

```
protected String title;
protected String errorTxt;

protected void setup()
{ err = new ErrorMsg(); // this is the frame used to inform the
                        // programmer about errors regarding the loading
                        // of the theory file or the failure of the
                        // "main" goal
  Object[] args = getArguments();
  String myName = getLocalName();
  fileName = null;
  if(args == null)
  { title = " The Theory Of Agent "+myName+" is Missing";
    errorTxt = " Agent "+myName+" has no theory specified.";
    err.write(title,errorTxt);
    return;
  }
  else if(args.length == 1)
  { fileName = (String) args[0];
    booleandone = init();
    if(done)
    { core = new Prolog();
      OutputListener l = new OutputListener()
      { public void onOutput(OutputEvent ev)
        { System.out.print(ev.getMsg()); }
      };
      core.addOutputListener(l);
      Library lib = null;
      try{ lib = core.loadLibrary("tuPInJADE.TuJadeLibrary");}
      catch(InvalidLibraryException error)
      { title = " Error Loading The TuJadeLibrary Of Agent"
                +myName;
        errorTxt = "The TuJadeLibrary is not a valid tuProlog library.";
        err.write(title,errorTxt);
        return;
      }
      ((TuJadeLibrary) lib).shell = this;
      try{ f = new FileInputStream(fileName);}
      catch(FileNotFoundException error)
      { title = " Error Loading The Theory Of Agent "+myName;
        errorTxt = " Cannot find the theory file "+ fileName+" of
                   agent "+myName+".";
        err.write(title,errorTxt);
        return;
      }
      catch(SecurityException error)
      { title = " Error Opening The Theory File Of Agent "+
```

```
            myName;
  errorTxt = " A security error occurred opening the theory
             file "+ fileName+" of agent "+myName+".";
  err.write(title,errorTxt);
  return;
}
catch(IOException error)
{ title = " Error Opening The Theory File Of Agent "+
          myName;
  errorTxt = " An error occurred opening the theory file "+
             fileName+" of agent "+myName+".";
  err.write(title,errorTxt);
  return;
}
Theory theory = null;
try
{ theory = new Theory(f);
  f.close();
}
catch(InvalidTheoryException ex)
{ title = " Error In Theory Of Agent "+myName;
  errorTxt = " The theory in file \""+
             fileName+"\" is not a valid one.";
  errorTxt = errorTxt.concat(" Error in line "+ex.line+".");
  err.write(title,errorTxt);
  try{ f.close();}
  catch(IOException exc)
  { title = " Cannot Close Theory File Of Agent "+myName;
    errorTxt = "Cannot close the theory file "+ fileName+"
               for agent "+myName+".";
    err.write(title,errorTxt);
  }
  return;
}
catch(IOException exc)
{ title = " Cannot Close Theory File Of Agent "+myName;
  errorTxt = "Cannot close the theory file "+ fileName+
             " for agent "+myName+".";
  err.write(title,errorTxt);
}
try{ core.addTheory(theory);}
catch(InvalidTheoryException ex)
{ title = " Error Loading The Theory Of Agent "+myName;
  errorTxt = " The theory in file ¨+fileName+"ïs not a valid
             one.";
  errorTxt = errorTxt.concat(" Error in line "+ex.line+".");
  err.write(title,errorTxt);
```

```
        return;
      }
      Shell42PBehaviour behave = new Shell42PBehaviour(this,core);
      addBehaviour(behave);
    }
  }
  else
  { title = " Too Many Inputs For Agent "+myName;
    errorTxt = " You can input only ONE theory file for agent"
               +myName+".";
    err.write(title,errorTxt);
  }
}

protected boolean init()
{ boolean ok = false;
  try
  { fr = new FileReader(fileName);
    in = new StreamTokenizer(fr);
    in.lowerCaseMode(false);
    ok = checkTheory(in);
    try{ fr.close();}
    catch(IOException exc)
    { title = " Cannot Close Theory File Of Agent "+getLocalName();
      errorTxt = "Cannot close the theory file "+ fileName+" for
                 agent "+getLocalName()+".";
      err.write(title,errorTxt);
    }
    if(ok) return true;
  }
  catch(FileNotFoundException error)
  { title = " Cannot Find Theory File Of Agent "+getLocalName();
    errorTxt = "Cannot find the theory file "+fileName+" for agent"
               +getLocalName()+".";
    err.write(title,errorTxt);
  }
  return false;
}

protected boolean checkTheory(StreamTokenizer stream)
{ int read;
  String aux;
  try
  { read = stream.nextToken();
    if(read == StreamTokenizer.TT_EOF ||
       read != StreamTokenizer.TT_WORD)
    { try{ fr.close();}
```

```
  catch(IOException exc)
  { title = " Cannot Close Theory File Of Agent "+getLocalName();
    errorTxt = "Cannot close the theory file "+ fileName+" for
               agent "+getLocalName()+".";
    err.write(title,errorTxt);
  }
  aux = " does not begin with a word";
  err.write("Syntax Error In Theory File "+fileName,"The theory
            file for agent "+getLocalName()+aux);
  return false;
}
String s = stream.sval;
if(s.equals("main.")) return true;
if(s.equals("main"))
{ stream.eolIsSignificant(true);
  read = stream.nextToken();
  if((read == StreamTokenizer.TT_EOF)||
     (read == StreamTokenizer.TT_EOL))
  { try{ fr.close();}
    catch(IOException exc)
    { title = " Cannot Close Theory File Of Agent "+getLocalName();
      errorTxt = "Cannot close the theory file "+ fileName+
                 " for agent "+getLocalName()+".";
      err.write(title,errorTxt);
    }
    aux = " the key word 'main' is not followed by ':-'.";
    err.write("Syntax Error In Theory File "+fileName,"In the
              theory file for agent "+getLocalName()+aux);
    return false;
  }
  if(read != StreamTokenizer.TT_WORD)
  { s = stream.toString();
      if(s.charAt(7)==':')
      { stream.ordinaryChar(' ');
        read = stream.nextToken();
        s = stream.toString();
        if(s.charAt(7)!='-')
        { try{ fr.close();}
          catch(IOException exc)
          { title = " Cannot Close Theory File Of Agent "+
                    getLocalName();
            errorTxt = "Cannot close the theory file "+ fileName+
                       " for agent "+getLocalName()+".";
            err.write(title,errorTxt);
          }
          aux = " the key character ':'  is not followed by '-'.";
          err.write("Syntax Error In Theory File "+fileName+
```

```
                    " For Agent "+getLocalName(),"In the theory
                    file for agent "+getLocalName()+aux);
            return false;
          }
          return true;
        }
        else
        { try{ fr.close();}
          catch(IOException exc)
          { title = " Cannot Close Theory File Of Agent "+
                    getLocalName();
            errorTxt = "Cannot close the theory file "+ fileName+
                       " for agent "+getLocalName()+".";
            err.write(title,errorTxt);
          }
          aux = " there is another character following 'main' where
                there should be ':'.";
          err.write("Syntax Error In Theory File "+fileName,"In
                    the theory file for agent "+getLocalName()+aux);
          return false;
        }
      }
    }
    else
    { try{ fr.close();}
      catch(IOException exc)
      { title = " Cannot Close Theory File Of Agent "+getLocalName();
        errorTxt = "Cannot close the theory file "+ fileName+
                   " for agent "+getLocalName()+".";
        err.write(title,errorTxt);
      }
      aux = " begins with a word different from 'main'.";
      err.write("Syntax Error In Theory File "+fileName,"The theory
                file for agent "+getLocalName()+aux);
      return false;
    }
  }
  catch(IOException error)
  { try{ fr.close();}
    catch(IOException exc)
    { title = " Cannot Close Theory File Of Agent "+getLocalName();
      errorTxt = "Cannot close the theory file "+ fileName+" for
                 agent "+getLocalName()+".";
      err.write(title,errorTxt);
    }
    aux = "An error occurred while reading the theory file for
          agent ";
```

```
          err.write("Syntax Error In Theory File "+fileName,
                      aux+getLocalName()+".");
          return false;
        }
        return false;
      }
    }

final class Shell42PBehaviour extends CyclicBehaviour
{  private Prolog engine;

    Shell42PBehaviour(Agent a,Prolog core)
    { super(a);
      engine = core;
    }

    public void action()
    { String agentName = myAgent.getLocalName();
      String title;
      String errorTxt;
      try
      { SolveInfo info = engine.solve("main.");
        Library lib = engine.getLibrary("tuPInJADE.TuJadeLibrary");
        if(lib == null)
        { title = " Error Obtaining The TuJadeLibrary from Agent "+
                  agentName;
          errorTxt = " The TuJadeLibrary was not loaded.";
          ((JadeShell42P) myAgent).err.write(title,errorTxt);
          return;
        }
        if(!info.isSuccess())
        { if(((TuJadeLibrary) lib).errorMsg!=null)
          { title = " Error Solving The 'main' Goal Of Agent "+agentName;
            errorTxt = ((TuJadeLibrary) lib).errorMsg;
            ((JadeShell42P) myAgent).err.write(title,errorTxt);
            return;
          }
          while(engine.hasOpenAlternatives())
          { info = engine.solveNext();
            if(((TuJadeLibrary) lib).errorMsg!=null)
            { title = " Error Solving The 'main' Goal Of Agent "+agentName;
              errorTxt = ((TuJadeLibrary) lib).errorMsg;
              ((JadeShell42P) myAgent).err.write(title,errorTxt);
              return;
            }
            if(info.isSuccess()) break;
          }
```

```
        if(!info.isSuccess())
        { title = " Error Solving The 'main' Goal Of Agent "+agentName;
          errorTxt = " Failure solving the main goal of agent "+
                     agentName+".";
          ((JadeShell42P) myAgent).err.write(title,errorTxt);
          return;
        }
      }
    }
    catch(MalformedGoalException mge)
    { title = " Error Solving The 'main' Goal Of Agent "+agentName;
      errorTxt = " The 'main' goal isn't a valid one.";
      ((JadeShell42P) myAgent).err.write(title,errorTxt);
      return;
    }
    catch(NoMoreSolutionException error)
    { title = " Error Solving The 'main' Goal Of Agent "+agentName;
      errorTxt = " There are no more solutions.";
      ((JadeShell42P) myAgent).err.write(title,errorTxt);
      return;
    }
  }
}
```

JadeShell42PGui.java

```
package tuPInJADE;

import alice.tuprolog.InvalidLibraryException;
import alice.tuprolog.InvalidTheoryException;
import alice.tuprolog.Library;
import alice.tuprolog.MalformedGoalException;
import alice.tuprolog.NoMoreSolutionException;
import alice.tuprolog.NoSolutionException;
import alice.tuprolog.OutputEvent;
import alice.tuprolog.OutputListener;
import alice.tuprolog.Prolog;
import alice.tuprolog.SolveInfo;
import alice.tuprolog.Theory;
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.StreamTokenizer;
import java.awt.Frame;
import java.awt.FileDialog;
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

// The JadeShell42PGui class is a JADE agent that behaves as a "shell"
// for a tuProlog agent containing a tuProlog engine.
// This agent, when loaded in JADE, launches
// a file system browser with which the user must select a file
// containing a tuProlog theory that defines the "main/0" predicate
// that represents the behaviour of the agent.
// If the theory file cannot be opened or does not contain a
// correct tuProlog theory, the user is made aware of it by the
// popup of an error message.
// If the theory file is indeed correct, the agent creates a
// tuProlog engine containing the standard tuProlog libraries and
// then it extends it by loading the TuJadeLibrary as well.
// The TuJadeLibrary is indispensable to a JadeShell42PGui agent
// because it provides the means of communication with any other
// agent in JADE, including JadeShell42PGui agents of course.
// The TuJadeLibrary defines predicates whose goal is, respectively,
// sending/receiving ACL messages to/from agents in a JADE platform.
//
public final class JadeShell42PGui extends JadeShell42P
{ protected void setup()
```

```
{ Object[] args = getArguments();
    err = new ErrorMsg(); // this is the frame used to inform the
                          // programmer about errors regarding the
                          // loading of the theory file or the
                          // solving of the "main" goal
    String myName = getLocalName();
    if(args != null)
    { if(args.length >= 1)
      { title = "Theory Of Agent "+myName;
        errorTxt = "The theory of agent "+myName+" should not be"+
                   " specified in the command line:it should be "
                   +"selected from the GUI launched by the agent
                   itself.";
        err.write(title,errorTxt);
        return;
      }
    }
    Selector sel = new Selector();
    fileName = sel.select(myName);
    if(fileName == null) // no file has been selected from the displayed
                         // window browsing the file system
    { title = " The Theory Of Agent "+myName+" is Missing";
      errorTxt = " Agent "+myName+" has no theory specified.";
      err.write(title,errorTxt);
      return;
    }
    boolean done = init();
    if(done)
    { core = new Prolog();
      OutputListener l = new OutputListener()
      { public void onOutput(OutputEvent ev)
        { System.out.print(ev.getMsg()); }
      };
      core.addOutputListener(l);
      Library lib = null;
      try{ lib = core.loadLibrary("tuPInJADE.TuJadeLibrary");}
      catch(InvalidLibraryException error)
      { title = " Error Trying To Load The TuJadeLibrary Of Agent"
                +" "+myName;
        errorTxt = "The TuJadeLibrary is not a valid tuProlog library.";
        err.write(title,errorTxt);
        return;
      }
      ((TuJadeLibrary) lib).shell = this;
      try{ f = new FileInputStream(fileName);}
      catch(FileNotFoundException error)
      { title = " Error Trying To Load The Theory Of Agent "+myName;
```

```
  errorTxt = " Cannot find the theory file "+ fileName+" of"
             +" agent "+myName+".";
  err.write(title,errorTxt);
  return;
}
catch(SecurityException error)
{ title = " Error Trying To Open The Theory File Of Agent "+myName;
  errorTxt = " A security error occurred opening the theory"
             +" file "+ fileName+" of agent "+myName+".";
  err.write(title,errorTxt);
  return;
}
catch(IOException error)
{ title = " Error Trying To Open The Theory File Of Agent "+myName;
  errorTxt = " An error occurred opening the theory file "+
             fileName+" of agent "+myName+".";
  err.write(title,errorTxt);
  return;
}
Theory theory = null;
try
{ theory = new Theory(f);
  f.close();
}
catch(InvalidTheoryException ex)
{ title = " Error In Theory Of Agent "+myName;
  errorTxt = " The theory in file \""+fileName+"\" is not a
             valid one.";
  errorTxt = errorTxt.concat(" Error in line "+ex.line+".");
  err.write(title,errorTxt);
  try{ f.close();}
  catch(IOException exc)
  { title = " Cannot Close Theory File Of Agent "+myName;
    errorTxt = "Cannot close the theory file "+ fileName+" for
               agent "+myName+".";
    err.write(title,errorTxt);
  }
  return;
}
catch(IOException exc)
{ title = " Cannot Close Theory File Of Agent "+myName;
  errorTxt = "Cannot close the theory file "+ fileName+" for
             agent "+myName+".";
  err.write(title,errorTxt);
}
try{ core.addTheory(theory);}
catch(InvalidTheoryException ex)
```

```
    { title = " Error Trying To Load The Theory Of Agent "+myName;
      errorTxt = " The theory in file \""+fileName+"\" is not a
                   valid one.";
        errorTxt = errorTxt.concat(" Error in line "+ex.line+".");
        err.write(title,errorTxt);
        return;
      }
      Shell42PBehaviourGui behave = new Shell42PBehaviourGui(this,
                                    core);
      addBehaviour(behave);
  }
}

final class Selector implements WindowListener
{ public String select(String agent)
  { String directory;
    StreamTokenizer input;
    String theory;
    Frame box = new Frame();
    FileDialog dialog = new FileDialog(box);
    box.setSize(425,100);
    box.setLocationRelativeTo(null);
    dialog.setTitle("Please Select the Theory File of Agent "+agent);
    box.pack();
    dialog.setVisible(true);
    if(dialog.getFile()!=null)
    { directory = dialog.getDirectory();
      theory = directory+dialog.getFile();
      return theory;
    }
    return null;
  }

  public void windowOpened(WindowEvent e){}

  public void windowClosing(WindowEvent e){ System.exit(0);}

  public void windowClosed(WindowEvent e){}

  public void windowIconified(WindowEvent e){ System.exit(0);}

  public void windowDeiconified(WindowEvent e){}

  public void windowActivated(WindowEvent e){}

  public void windowDeactivated(WindowEvent e){ System.exit(0);}
}}
```

```
final class Shell42PBehaviourGui extends CyclicBehaviour
{ private Prolog engine;

  Shell42PBehaviourGui(Agent a,Prolog core)
  { super(a);
    engine = core;
  }

  public void action()
  { String agentName = myAgent.getLocalName();
    String title;
    String errorTxt;
    try
    { SolveInfo info = engine.solve("main.");
      Library lib = engine.getLibrary("tuPInJADE.TuJadeLibrary");
      if(lib == null)
      { title = " Error Obtaining The TuJadeLibrary from Agent "+agentName;
        errorTxt = " The TuJadeLibrary was not loaded.";
        ((JadeShell42PGui) myAgent).err.write(title,errorTxt);
        return;
      }
      if(!info.isSuccess())
      { if(((TuJadeLibrary) lib).errorMsg!=null)
        { title = " Error Solving The 'main' Goal Of Agent "+agentName;
          errorTxt = ((TuJadeLibrary) lib).errorMsg;
          ((JadeShell42PGui) myAgent).err.write(title,errorTxt);
          return;
        }
        while(engine.hasOpenAlternatives())
        { info = engine.solveNext();
          if(((TuJadeLibrary) lib).errorMsg!=null)
          { title = " Error Solving The 'main' Goal Of Agent "+agentName;
            errorTxt = ((TuJadeLibrary) lib).errorMsg;
            ((JadeShell42PGui) myAgent).err.write(title,errorTxt);
            return;
          }
          if(info.isSuccess()) break;
        }
        if(!info.isSuccess())
        { title = " Error Solving The 'main' Goal Of Agent "+agentName;
            errorTxt = " Failure solving the main goal of agent "+agentName+".";
            ((JadeShell42PGui) myAgent).err.write(title,errorTxt);
            return;
          }
        }
      }
```

```
      catch(MalformedGoalException mge)
      { title = " Error Solving The 'main' Goal Of Agent "+agentName;
        errorTxt = " The 'main' goal isn't a valid one.";
        ((JadeShell42PGui) myAgent).err.write(title,errorTxt);
        return;
      }
      catch(NoMoreSolutionException error)
      { title = " Error Solving The 'main' Goal Of Agent "+agentName;
        errorTxt = " There are no more solutions.";
        ((JadeShell42PGui) myAgent).err.write(title,errorTxt);
        return;
      }
      block();
   }
}
```

TuJadeLibrary.java

```
package tuPInJADE;
package tuPInJADE;

import alice.tuprolog.*;
import jade.lang.acl.ACLMessage;
import jade.core.AID;

/**
* This class is a tuProlog Library developed to provide JadeShell42P
* agents with predicates implementing the sending and receiving
* of FIPA compliant and asynchronous messages, in other words,
* the messages exchanged in JADE platforms.
* Since the content of the messages exchanged in JADE is always a string
* and not a java Object, it has been necessary to define the predicates
* pack(Term,StrTerm) and unpack(StrTerm,Term).
* The predicate pack(Term,StrTerm) is true if and only if StrTerm
* is a string representation of the tuProlog term Term.
* The predicate unpack(StrTerm,Term) is true if and only if Term
* is the tuProlog term represented by the string StrTerm.
* Thus, if a JadeShell42P agent wishes to send a tuProlog term
* to another JadeShell42P agent, its main theory should be as
* follows:
* ".....,pack(TermToSend,StringRepresentation),
* send(...,StringRepresentation,....),.....".
*
* On the other hand, to allow a JadeShell42P agent that
* receives a message to deal with a content that is a tuProlog term,
* instead of a string, its theory should be as follows:
* ".....,receive(...,StringRepresentation,....),
* unpack(StringRepresentation,TermReceived),.....".
*
*
* Defined predicates:
* - send(Performative,Content,Receiver)
* - receive(Performative,Content,Sender)
* - blocking_receive(Performative,Content,Sender)
* - blocking_receive(Performative,Content,Sender,Millisec)
```

```
* - pack(Term,StrTerm)
* - unpack(StrTerm,Term)
*/

public class TuJadeLibrary extends Library
// The shell field is used to reference the JadeShell42P
// agent, in order to be able to invoke its send and
// receive methods and the method that returns its address
// in the JADE platform.
protected JadeShell42P shell;

// The errorMsg field is the string displayed in the
// window used by the JadeShell42P agent to communicate
// with the programmer in case there are errors.
public String errorMsg = null;

public TuJadeLibrary() {}

public boolean send_3(Struct g)
{ /** Usage:  send(Performative,Content,Receiver).
*
* This predicate is true if, and only if, Performative
* is a string that corresponds to one of the available
* performatives of an ACLMessage in JADE, Content is a
* string and Receiver is a string corresponding to a
* valid JADE address or a list of strings corresponding
* to a list of JADE addresses.
* Any string is a valid content of the message, except
* for the string consisting only of the anonymous
* variable ("_").
*
**/
Term arg1 = g.getTerm(0); // the term representing the performative
// of the ACLMessage to send
Term arg2 = g.getTerm(1); // the term representing the content of the
// ACLMessage to send
Term arg3 = g.getTerm(2); // the term representing the receiver or the
// list of receivers of the ACLMessage to send
String performative = check1(arg1);
```

```
if (performative == null) return false; // the performative is not a
//valid one
String content = check2(arg2);
if (content == null) return false; // the content is not a valid one
AID sender = shell.getAID(); // it is the global unique identifier of
// the sender in the JADE platform, in
// other words, the address in JADE of
// the agent sender of the message
ACLMessage mail = new ACLMessage(ACLMessage.getInteger(performative));
mail.setContent(content);
mail.setSender(sender);
String receiver = check3(arg3,mail); // the global unique identifier
// of the agent in the JADE
// platform
if (receiver == null) return false;
shell.send(mail); // the message is sent invoking the send method
// of any agent in JADE
return true;
}

private String check1(Term t)
{ // It checks if the term t represents a valid performative of
// an ACLMessage in JADE.
// If it recognises a valid performative it returns the
// corresponding string.
// The null string is returned in case the term t does
// not represent a valid performative.
if (t.isCompound())
{ errorMsg = " ERROR IN SEND PREDICATE: the performative "
+t.toString()+" of the message to send is a "+
"compound term.";
return null;
}
String aux;
if (t.isVar())
{ if (!((Var) t).isBound())
{ errorMsg = " ERROR IN SEND PREDICATE: the performative "
+t.toString()+" of the message to send is an unbound variable.";
return null;
```

```
}
else aux = ((Struct) (t.getTerm())).getName();
// aux is the value bound to the variable t
}
else aux = ((Struct) t).getName();
// aux is the string representing the atom t
String [] list = ACLMessage.getAllPerformativeNames();
// list contains all the possible performatives of an ACLMessage
boolean found = false;
int j = -1;
for(int i=0;i<list.length;i++)
{ if (aux.equals(list[i]))
{ j = i;
found = true;
}
}
if(!found) return "NOT_UNDERSTOOD";
// the performative is not recognised as a FIPA compliant
// one, but the message can still be sent with the default
// JADE performative NOT_UNDERSTOOD.
else return list[j];
}

private String check2(Term t)
{ // It checks if the term t represents a valid content
// of an ACLMessage in JADE.
// Any string that differs from the string containing
// exclusively the anonymous variable, "_",
// is considered as a valid content.
// The null string is returned in case the term t
// does not represent a valid content.
if (t.isCompound())
{ errorMsg = " ERROR IN SEND PREDICATE: the content "
+t.toString()+" of the message to send cannot be a compound term.";
return null;
}
String msg;
if (t.isVar())
{ if (!((Var) t).isBound())
```

```
{ errorMsg = " ERROR IN SEND PREDICATE: the content "+t.toString()+
" of the message to send is an unbound variable.";
return null;
}
else msg = ((Struct) (t.getTerm())).getName();
// msg is the value bound to the variable t
}
else
{ msg = ((Struct) t).getName();
// msg is the string representing the atom t
if (msg.equals("_"))
{ errorMsg = " ERROR IN SEND PREDICATE: the anonymous variable '_'
is not a "+"valid content of the message to send.";
return null;
}
}
return msg;
}

private String check3(Term t,ACLMessage mail)
{ // It checks if the term t represents a valid receiver
// or a valid list of receivers of an ACLMessage in JADE.
// It returns the string corresponding to the last
// receiver read if the receivers are all valid ones.
// The null string is returned in case the term t does
// not represent a valid receiver or list of receivers.
String receiver = null;
if (t.isList()) //there is a list of receivers
{ if (((Struct) t).isEmptyList())
{ errorMsg = " ERROR IN SEND PREDICATE: "+
"there is an empty list as receiver of "+"the message to send.";
return null;
}
else // the list of receivers is not empty
{ Term head = ((Struct) t).listHead();
// head is the first element in the list of receivers
Struct tail = ((Struct) t).listTail();
// tail is the list of receivers without the first element
while(!head.isNull())
```

```
{ receiver = checkRec(head);
if (receiver != null)
{ AID rcvr = new AID();
rcvr.setName(receiver);
mail.addReceiver(rcvr);
// the JADE global unique identifier of one of the
// receivers is set in the message
}
else break;
// it has been read a term that does not represent a valid receiver
// of the message
head = tail.listHead();
if (!tail.isEmptyList()) tail = tail.listTail();
}
return receiver;
}
}
else // there is only one receiver of the message to send
{ receiver = checkRec(t);
AID rcvr = new AID();
rcvr.setName(receiver);
mail.addReceiver(rcvr);
// the global unique identifier in JADE of one of the receivers
// is set in the message
}
return receiver;
}

private String checkRec(Term t)
{ // It checks if the term t represents a valid receiver
// of an ACLMessage in JADE.
// It returns the string corresponding to the receiver.
// The null string is returned in case the term t does
// not represent a valid receiver.
if (t.isNumber())
{ errorMsg = " ERROR IN SEND PREDICATE: the AID "+
"(global unique identifier in JADE) "+t.toString()+" of one of the"+
" receivers of the message to send is a number "+"instead of a string.";
return null;
```

```
}
if (t.isCompound())
{ errorMsg = "ERROR IN SEND PREDICATE: the AID "+
"(global unique identifier in JADE) "+t.toString()+" of one of the "+
"receivers of the message to send is a "+"compound term.";
return null;
}
String rcv;
if (t.isVar())
{ if (!((Var) t).isBound())
{ errorMsg = " ERROR IN SEND PREDICATE: the AID "+
"(global unique identifier in JADE) "+t.toString()+" of one of the "+
"receivers of the message to send is an "+"unbound variable.";
return null;
}
else rcv = ((Struct) (t.getTerm())).getName();
// the value bound to the variable
}
else if (t.isList())
{ if (((Struct) t).isEmptyList())
{ errorMsg = " ERROR IN SEND PREDICATE: one of the "+
"receivers of the message to send "+is an empty list.";
return null;
}
else
{ errorMsg = " ERROR IN SEND PREDICATE: the AID "+
"(global unique identifier in JADE) "+t.toString()+" of one of the"+
" receivers of the message to send is a list "+"it self.";
return null;
}
}
else rcv = t.toString();
rcv = rcv.substring(1,rcv.length()-1);
// it removes the first and last symbols equal to the character '
int i = rcv.indexOf('@');
if (i == -1) // the AID (global unique identifier in JADE)
// of a receiver does not contain the symbol '@'
{ errorMsg = " ERROR IN SEND PREDICATE: the symbol '@' is missing "+
"in the AID (global "+"unique identifier in JADE) "+t.toString()+
```

```
" of one of the receivers of "+"the message to send.";
return null;
}
else
{ int j = rcv.indexOf(':');
if (j<i)
{ errorMsg = " ERROR IN SEND PREDICATE: the symbol ':'  "+
"is missing, or it occurs "+"before the '@' symbol, in the AID "+
t.toString()+" of one of the "+"receivers of the message to send.";
return null;
}
}
return rcv;
}

public boolean receive_3(Struct g)
{ /** Usage:  receive(Performative,Content,Sender)
*
* This predicate is true if, and only if, Performative is a variable bound
* to a string that corresponds to one of the available performatives of an
* ACLMessage in JADE, Content is a variable bound to a string and Sender
* is a variable bound to a string corresponding to a valid JADE address.
* The Content variable is a valid content of the message received only
* if it is bound to a string that does not just contain anonymous
* variable ("_").
*
**/
Term perf = g.getTerm(0);
// the term representing the performative of the ACLMessage received
Term msg = g.getTerm(1);
// the term representing the content of the ACLMessage received
Term sender = g.getTerm(2);
// the term representing the sender of the ACLMessage received
if (perf.isVar())
{ if (((Var) perf).isBound())
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the performative "+
perf.toString()+" of the message to receive is a bound variable.";
return false;
}
```

```
}
if (perf.isGround())
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the performative "+
perf.toString()+" of the message to receive is a ground term "+
"instead of a variable.";
return false;
}
if (perf.isCompound())
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the performative "+
perf.toString()+" of the message to receive is a compound "+
"term instead of a variable.";
return false;
}
if (msg.isCompound())
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the content "+msg.toString()+
" of the message to receive is a compound term instead of a variable.";
return false;
}
if (msg.isVar())
{ if (((Var) msg).isBound())
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the content "+msg.toString()+
" of the message to receive is a variable already bound.";
return false;
}
}
else
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the content "+msg.toString()+
" of the message to receive is not a variable.";
return false;
}
if (sender.isCompound())
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the AID (global unique "+
"identifier in JADE) "+sender.toString()+
" of the sender of the message "+
"to receive is a compound term instead of a variable.";
return false;
}
if (sender.isVar())
{ if (((Var) sender).isBound())
```

```
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the AID (global unique "+
"identifier in JADE) "+sender.toString()+
" of the sender of the message "+
"to receive is a variable already bound.";
return false;
}
}
else
{ errorMsg = " ERROR IN RECEIVE PREDICATE: the AID (global unique "+
"identifier in JADE) "+sender.toString()+" of the sender of the message "+
"to receive is not a variable.";
return false;
}
ACLMessage mail = shell.receive();
// the message is received invoking the receive method of any agent in JADE
if (mail!=null) // a message has actually been received
{ String performative = mail.getPerformative(mail.getPerformative());
// it reads the performative of the message received
String content = mail.getContent();
// it reads the content of the message received
String sendAddr = mail.getSender().getName();
// it reads the sender of the message received
Prolog core = getEngine();
core.unify(perf,new Struct(performative));
// it unifies the performative of the message received with the term
// representing such performative
core.unify(msg,new Struct(content));
// it unifies the content of the message received with the term
// representing such content
core.unify(sender,new Struct(sendAddr));
// it unifies the sender of the message received with the term
// representing such sender
}
return true;
}

public boolean blocking_receive_3(Struct g)
{ /** Usage:  blocking_receive(Performative,Content,Sender)
*
```

```
* This predicate is true if, and only if, Performative is a
* variable bound to a string that corresponds to one of the available
* performatives of an ACLMessage in JADE, Content is a variable bound
* to a string and Sender is a variable bound to a string corresponding
* to a valid JADE address.
* The Content variable is a valid content of the message received only
* if it is bound to a string that differs from the simple anonymous
* variable ("_").
*
**/
Term perf = g.getTerm(0);
// the term representing the performative of the ACLMessage received
Term msg = g.getTerm(1);
// the term representing the content of the ACLMessage received
Term sender = g.getTerm(2);
// the term representing the sender of the ACLMessage received
if (perf.isVar())
{ if (((Var) perf).isBound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: "+
"the performative "+perf.toString()+" of the message to receive "+
"is a variable already bound.";
return false;
}
}
if (perf.isGround())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: "+
"the performative "+perf.toString()+" of the message to receive is "+
"a ground term instead of a variable.";
return false;
}
if (perf.isCompound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: "+
"the performative "+perf.toString()+" of the message to receive is a"+
" compound term instead of a variable.";
return false;
}
if (msg.isCompound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: the content "+
msg.toString()+" of the message to receive is a compound term "+
```

```
"instead of a variable.";
return false;
}
if (msg.isVar())
{ if (((Var) msg).isBound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: the content "+
msg.toString()+" of the message to receive is a variable already bound.";
return false;
}
}
else
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: the content "+
msg.toString()+" of the message to receive is not a variable.";
return false;
}
if (sender.isCompound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: the AID "+
"(global unique identifier in JADE) "+sender.toString()+" of the sender of the
message "+
"to receive is a compound term instead of a variable.";
return false;
}
if (sender.isVar())
{ if (((Var) sender).isBound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: "+
"the AID (global unique identifier in JADE) "+sender.toString()+
" of the sender of the message to receive is a variable already bound.";
return false;
}
}
else
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE: the AID "+
"(global unique identifier in JADE) "+sender.toString()+
" of the sender of the message to receive is not a variable.";
return false;
}
ACLMessage mail = shell.blockingReceive();
// the message is received invoking the blockingReceive
// method of any agent in JADE
```

```
if (mail!=null) // a message has actually been received
{ String performative = mail.getPerformative(mail.getPerformative());
// it reads the performative of the message received
String content = mail.getContent();
// it reads the content of the message received
String sendAddr = mail.getSender().getName();
// it reads the sender of the message received
Prolog core = getEngine();
core.unify(perf,new Struct(performative));
// it unifies the performative of the message received
// with the term representing such performative
core.unify(msg,new Struct(content));
// it unifies the content of the message received with
// the term representing such content
core.unify(sender,new Struct(sendAddr));
// it unifies the sender of the message received with
// the term representing such sender
}
return true;
}

public boolean blocking_receive_4(Struct g)
{ /** Usage:  blocking_receive(Performative,Content,Sender,Millisec)
*
* This predicate is true if, and only if, Performative is a variable
* bound to a string that corresponds to one of the available
* performatives of an ACLMessage in JADE, Content is a variable
* bound to a string, Sender is a variable bound to a string
* corresponding to a valid JADE address and Millisec is a number.
* The Content variable is a valid content of the message received
* only if it is bound to a string that differs from the simple
* anonymous variable ("_").
* The Millisec variable is a number that represents the amount
* of time that the agent will remain blocked to receive a message.
*
**/
Term perf = g.getTerm(0);
// the term representing the performative of the ACLMessage received
Term msg = g.getTerm(1);
```

```
// the term representing the content of the ACLMessage received
Term sender = g.getTerm(2);
// the term representing the sender of the ACLMessage received
Term milli = g.getTerm(3);
// the term representing the amount of time for which the agent
//will wait for a message to arrive
if (perf.isVar())
{ if (((Var) perf).isBound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH "+
"TIMEOUT: the performative "+perf.toString()+
" of the message to receive is a variable already bound.";
return false;
}
}
if (perf.isGround())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT:"+
" the performative "+perf.toString()+
" of the message to receive is a ground term instead of a variable.";
return false;
}
if (perf.isCompound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT:"+
" the performative "+perf.toString()+
" of the message to receive is a compound term instead of a variable.";
return false;
}
if (msg.isCompound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT:"+
" the content "+msg.toString()+
" of the message to receive is a compound term instead of a variable.";
return false;
}
if (msg.isVar())
{ if (((Var) msg).isBound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT:"+
" the content "+msg.toString()+
" of the message to receive is a variable already bound.";
return false;
}
```

```
}
else
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT:"+
" the content "+msg.toString()+
" of the message to receive is not a variable.";
return false;
}
if (sender.isCompound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT:"+
" the AID (global unique identifier in JADE) "+
sender.toString()+" of the sender of the message to receive is a compound"+
" term instead of a variable.";
return false;
}
if (sender.isVar())
{ if (((Var) sender).isBound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT:"+
" the AID (global unique identifier in JADE) "+
sender.toString()+" of the sender of the message to receive is a "+
"variable already bound.";
return false;
}
}
else
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT:"+
" the AID (global unique identifier in JADE) "+
sender.toString()+" of the sender of the message to"+
" receive is not a variable.";
return false;
}
long mil = -1;
if (milli.isVar())
{ if (!((Var) milli).isBound())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT: "+
"the maximum time "+milli.toString()+" to wait for a message "+
"to arrive is an unbound variable.";
return false;
}
else mil = (new java.lang.Long(((Struct) (milli.getTerm())).
```

```
getName())).longValue();
}
else
{ if (!milli.isNumber())
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT: "+
"the maximum time "+milli.toString()+
" to wait for a message to arrive is not a number.";
return false;
}
else
{ if (((alice.tuprolog.Number) milli).isTypeLong())
mil =((alice.tuprolog.Long) milli).longValue();
else
{ errorMsg = " ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT: "+
"the maximum time "+milli.toString()+
" to wait for a message to arrive is not a valid number.";
return false;
}
}
}
ACLMessage mail = shell.blockingReceive(mil);
// the message is received invoking the blockingReceive
// method of any agent in JADE
if (mail!=null) // a message has actually been received
{ String performative = mail.getPerformative(mail.getPerformative());
// it reads the performative of the message received
String content = mail.getContent();
// it reads the content of the message received
String sendAddr = mail.getSender().getName();
// it reads the sender of the message received
Prolog core = getEngine();
core.unify(perf,new Struct(performative));
// it unifies the performative of the message received with
// the term representing such performative
core.unify(msg,new Struct(content));
// it unifies the content of the message received with
// the term representing such content
core.unify(sender,new Struct(sendAddr));
// it unifies the sender of the message received with
```

```
// the term representing such sender
}
return true;
}

public boolean pack_2(Struct g)
{ /** Usage:  pack(Term,StrTerm)
*
* This predicate is true if, and only if, StrTerm is the string
* representation of the tuProlog term Term.
*
**/
Term term = g.getTerm(1);
Prolog core = getEngine();
core.unify(new Struct(g.getTerm(0).toString()),term);
// it tries to unify the string representation of the first
// argument to the second argument
return true;
}

public boolean unpack_2(Struct g)
{ /** Usage:  unpack(StrTerm,Term)
*
* This predicate is true if, and only if, Term is the tuProlog term
* corresponding to the string StrTerm.
*
**/
String str = g.getTerm(0).toString();
if (str.charAt(0) == '') str = str.substring(1,str.length()-1);
Term term = g.getTerm(1);
Prolog core = getEngine();
try { core.unify(term.parse(str),term);}
catch (InvalidTermException ecc)
{ errorMsg = " ERROR IN UNPACK PREDICATE: "+
" the string "+str+" is not a valid tuProlog term.";
return false;
}
return true;
}
```

```
}
```

Appendix B

Available performatives in JADE

The performatives that can characterise a JADE message are the following:

- **ACCEPT_PROPOSAL**
- **AGREE**
- **CANCEL**
- **CFP**
- **CONFIRM**
- **DISCONFIRM**
- **FAILURE**
- **INFORM**
- **INFORM_IF**
- **INFORM_REF**
- **NOT_UNDERSTOOD**
- **PROPAGATE**
- **PROPOSE**
- **PROXY**
- **QUERY_IF**
- **QUERY_REF**

- **REFUSE**
- **REJECT_PROPOSAL**
- **REQUEST**
- **REQUEST_WHEN**
- **REQUEST_WHENEVER**
- **SUBSCRIBE**
- **UNKNOWN**

Appendix C

Error messages caused by misuse of predicates defined in TuJadeLibrary

Here is a list of the error messages that are displayed by a tuProlog agent and are due to the misuse of the predicates defined in the TuJadeLibrary:

- " **ERROR IN SEND PREDICATE**: the performative `Performative` of the message to send is a compound term.";
- " **ERROR IN SEND PREDICATE**: the content `Content` of the msg to send cannot be a compound term.";
- " **ERROR IN SEND PREDICATE**: the content `Content` of the msg to send is an unbound variable.";
- " **ERROR IN SEND PREDICATE**: the anonymous variable '_' is not a valid content of the message to send.";
- " **ERROR IN SEND PREDICATE**: there is an empty list as receiver of the message to send.";
- " **ERROR IN SEND PREDICATE**: the AID (global unique identifier in JADE) `Receiver` of one of the receivers of the message to send is a number instead of a string.";
- " **ERROR IN SEND PREDICATE**: the AID (global unique identifier in JADE) `Receiver` of one of the receivers of the message to send is a compound term.";
- " **ERROR IN SEND PREDICATE**: the AID (global unique identifier in JADE) `Receiver` of one of the receivers of the message to send is an unbound variable.";

- " **ERROR IN SEND PREDICATE**: one of the receivers of the message to send is an empty list.";
- " **ERROR IN SEND PREDICATE**: the AID (global unique identifier in JADE) `Receiver` of one of the receivers of the message to send is a list it self.";
- " **ERROR IN SEND PREDICATE**: the symbol '@' is missing in the AID (global unique identifier in JADE) `Receiver` of one of the receivers of the message to send.";
- " **ERROR IN SEND PREDICATE**: the symbol ':' is missing, or it occurs before the '@' symbol, in the AID `Receiver` of one of the receivers of the message to send.";
- " **ERROR IN RECEIVE PREDICATE**: the performative `Performative` of the message to receive is a bound variable.";
- " **ERROR IN RECEIVE PREDICATE**: the performative `Performative` of the message to receive is a ground term instead of a variable.";
- " **ERROR IN RECEIVE PREDICATE**: the performative `Performative` of the message to receive is a compound term instead of a variable.";
- " **ERROR IN RECEIVE PREDICATE**: the content `Content` of the message to receive is a compound term instead of a variable.";
- " **ERROR IN RECEIVE PREDICATE**: the content `Content` of the message to receive is a variable already bound.";
- " **ERROR IN RECEIVE PREDICATE**: the content `Content` of the message to receive is not a variable.";
- " **ERROR IN RECEIVE PREDICATE**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is a compound term instead of a variable.";
- " **ERROR IN RECEIVE PREDICATE**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is a variable already bound.";

- " **ERROR IN RECEIVE PREDICATE**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is not a variable."';
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the performative `Performative` of the message to receive is a variable already bound.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the performative `Performative` of the message to receive is a ground term instead of a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the performative `Performative` of the message to receive is a compound term instead of a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the content `Content` of the message to receive is a compound term instead of a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the content `Content` of the message to receive is a variable already bound.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the content `Content` of the message to receive is not a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is a compound term instead of a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is a variable already bound.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is not a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the performative `Performative` of the message to receive is a variable already bound.";

- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the performative `Performative` of the message to receive is a ground term instead of a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the performative `Performative` of the message to receive is a compound term instead of a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the content `Content` of the message to receive is a compound term instead of a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the content `Content` of the message to receive is a variable already bound.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the content `Content` of the message to receive is not a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is a compound term instead of a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is a variable already bound.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the AID (global unique identifier in JADE) `Sender` of the sender of the message to receive is not a variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the maximum time `Millisec` to wait for a message to arrive is an unbound variable.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the maximum time `Millisec` to wait for a message to arrive is not a number.";
- " **ERROR IN BLOCKING_RECEIVE PREDICATE WITH TIMEOUT**: the maximum time `Millisec` to wait for a message to arrive is not a valid number.";

- " **ERROR IN UNPACK PREDICATE**: the string `StrTerm` is not a valid tuProlog term.";

Bibliography

[ACL] FIPA's ACL. *A*gent Communication Language Specifications. Home Page: *http://www.fipa.org/repository/aclspecs.php3*.

[ACT] ACTS. *ACTS European program.* Home Page: *http://www.infowin.org/ACTS/PROJECTS/* .

[AEK+99] K. Arisha, T. Eiter, S. Kraus, F. Ozcan, R. Ross, and V.S. Subrahmanian. "IMPACT: A Platform for Collaborating Agents". *IEEE Intelligent Systems*, 14(2):64–72, 1999.

[AMMM02] R. Albertoni, M. Martelli, V. Mascardi, and S. Miglia. "Specifica, Implementazione ed Esecuzione di un Prototipo di Sistema Multi-Agente in D-CaseLP". In *Proc. of WOA 2002*, Milano, Italy, 2002. Pitagora editrice, Bologna. In Italian.

[AMMR03] E. Astesiano, M. Martelli, V. Mascardi, and G. Reggio. "From Requirement Specification to Prototype Execution: a Combination of a Multiview Use-Case Driven Method and Agent-Oriented Techniques". In J. Debenham and K. Zhang, editors, *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)*, pages 578–585. The Knowledge System Institute, 2003.

[AS95] S. Adali and V.S. Subrahmanian. "Intelligent Caching in Hybrid Knowledge Bases". In N. Mars, editor, *Proc. of 1995 International Conference on Very Large Knowledge Bases*, pages 247–256, Twente, The Netherlands, 1995. IOS Press.

[AUM] AUML. *An Agent-based Unified Modeling Language.* by FIPA (The Foundation for Intelligent Physical Agents). Home Page: http://www.auml.org/.

[Aus62] J. L. Austin. *"How to do things with words"*. Clarendon Press, Oxford, UK, 1962.

[BBG+04] M. Baldoni, C. Baroglio, I. Gungui, A. Martelli, M. Martelli, V. Mascardi, V. Patti, and C. Schifanella. "Reasoning about Agents Interaction Protocols inside DCaseLP". In J. A. Leite, A. Omicini, P. Torroni, and P. Yolum, editors, *Proc. of Second International Workshop on Declarative Agent Languages and Technologies (DALT2004)*, pages 250–265, New York, U.S.A., July 2004.

[BBG+05] M. Baldoni, C. Baroglio, I. Gungui, A. Martelli, M. Martelli, V. Mascardi, V. Patti, and C. Schifanella. "Reasoning about agents' interaction protocols inside DCaseLP". In J. Leite, A. Omicini, P. Torroni, and P. Yolum, editors, *Post-Proc. of the International Workshop on Declarative Agent Languages and Technologies DALT'04*, page to appear, New York, USA, July 2005. Springer. Lecture Notes in Artificial Intelligence.

[BBMP03] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. "Reasoning about self and others: communicating agents in a modal action logic". In C. Blundo and C. Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS'2003*, volume 2841 of *LNCS*, pages 228–241, Bertinoro, Italy, October 2003. Springer.

[BDM+99] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. "Multi-Agent Systems Development as a Software Engineering Enterprise". In G.Gupta, editor, *Proc. of First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, pages 46–60, San Antonio, Texas, 1999. Springer-Verlag. LNCS 1551.

[BGMP] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. "Programming Rational Agents in a Modal Action Logic". Annals of Mathematics and Artificial Intelligence, Special issue on Logic-Based Agent Implementation. To appear.

[Boe86] B Boehm. "A spiral model of software development and enhancement". *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, August 1986. ACM Press.

[Cen] NASA's Johnson Space Center. *CLIPS: A Tool for Building Expert Systems*. Home Page: http://www.ghg.net/clips/CLIPS.html.

[Con] Mozart Consortium. *The Mozart Programming System*. Home Page: *http://www.mozart-oz.org/* .

[Del97] G. Delzanno. *Logic & Object-Oriented Programming In Linear Logic*. Ph.D. thesis TD2/97, University of Pisa, 1997. ftp://ftp.disi.unige.it/person/DelzannoG/papers/thesis.ps.gz.

[DKM+99] P. Dart, E. Kazmierczak, M. Martelli, V. Mascardi, L. Sterling, V.S. Subrahmanian, and F. Zini. "Combining Logical Agents with Rapid Prototyping for Engineering Distributed Applications". In S. Tilley and J. Verner, editors, *Proceedings of the Nineth International Conference of Software Technology and Engineering (STEP'99)*, pages 40–49, Pittsburgh, PA, September 1999. IEEE Computer Society Press.

[DOR01] E. Denti, A. Omicini, and A. Ricci. "tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures". In I. V. Ramakrishnan, editor, *Proc. of the 3rd International Symposium on Practical Aspects of Declarative Programming (PADL'01)*, volume 1990 of *LNCS*, pages 184–198, Las Vegas, NV, U.S.A., March 2001. Springer-Verlag. Home Page: http://lia.deis.unibo.it/research/tuprolog/.

[ES99] T. Eiter and V.S. Subrahmanian. "Heterogeneous Active Agents, II: Algorithms and Complexity". *Artificial Intelligence*, 108(1-2):257–307, 1999.

[ESP99] T. Eiter, V.S. Subrahmanian, and G. Pick. "Heterogeneous Active Agents, I: Semantics". *Artificial Intelligence*, 108(1-2):179–255, 1999.

[FG97] S. Franklin and A. Graesser. "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agent". In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer-Verlag, 1997. published as Intelligent Agents III.

[FG98] J. Ferber and O. Gutknecht. "A meta-model for the analysis and design of organizations in multiagent systems". In IEEE Computer Society, editor, *In Proceedings of Third International Conference on Multi-Agent Systems (ICMAS 98)*, pages 128–135, 1998.

[FH] Ernest Friedman-Hill. *Jess^{TM}, the rule engine for the java^{tm} platform*. Home Page: http://herzberg.ca.sandia.gov/jess/index.shtml.

[FIP] FIPA. *T*he Foundation for Intelligent Physical Agents. Home Page: http://www.fipa.org/.

[FLM95] T. Finin, Y. Labrou, and J. Mayfield. *"KQML as an agent communication language"*, pages 265–284. Software Agents. J. Bradshaw, Cambridge, MA, mit press edition, 1995.

[GF00] O. Gutknecht and J. Ferber. "MadKit: a generic multi-agent platform". In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, volume 1-58113-230-1, pages 78–79, Barcelona, Spain, 2000. ACM Press. Home Page: http://www.madkit.org/.

[Gro] Swarm Development Group. *Swarm*. Home Page: *http://www.swarm.org*.

[Hug02] M-P. Huget. *M*odel checking agent UML protocol diagrams. CS Department, University of Liverpool, UK, 2002. Technical Report.

[IBM] IBM. *Aglets*. Home Page: *http://www.trl.ibm.com/aglets/*.

[IGD] Fraunhofer IGD. *Secure Mobile Agents Project (SeMoA)*. Home Page: *http://www.semoa.org*.

[IKV] IKV++. *Grashopper Agent Platform*. Home Page: *http://www.ikv.de/content/Grasshopper/Grasshopper_e.htm*.

[JACa] JACK. *An introduction to JACK*. Home Page: *http://www.is.pku.edu.cn/dis/*.

[JACb] JACK. *The Agent Oriented Software Group*. Home Page: *http://www.agent-software.com/shared/home/index.html*.

[JMMS03] T. Juan, M. Martelli, V. Mascardi, and L. Sterling. "Customizing AOSE Methodologies by Reusing AOSE Features". In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the 2nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 113–120, Melbourne, Australia, 2003. ACM Press.

[JPS02] T. Juan, A. Pearce, and L. Sterling. "ROADMAP: Extending the Gaia Methodology for Complex Open Systems". In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002)*, pages 3–10, Bologna, Italy, July 2002.

[JSW98] N. R. Jennings, K. Sycara, and M. Wooldridge. "A Roadmap of Agent Research and Development". *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.

[LNS96] J. Lu, A. Nerode, and V. S. Subrahmanian. "Hybrid Knowledge Bases". *IEEE Transactions on Knowledge and Data Engineering*, 8(5):773–785, October 1996.

[Mag] General Magic. *Odyssey*. Home Page: http://www.genmagic.com/technology/odyssey.html.

[Mar99] S. Marini. "Specifica di sistemi multi-agente eterogenei". Master's thesis, DISI, University of Italy, Genova, Italy, 1999.

[Mas02] V. Mascardi. *Logic-Based Specification Environments for Multi-Agent Systems*. Ph.D. thesis DISI–TH–2002–04, University of Genova, 2002. Downloadable from ftp: ftp://ftp.disi.unige.it/person/MascardiV/Tesi/mythesis.ps.gz.

[MEH02] H. Mazouzi, A. El Fallah-Seghrouchni, and S. Haddad. "Open protocol design for complex interactions in multi-agent systems". In C. Castelfranchi and W. L. Johnson, editors, *Proc. of AAMAS 2002*, pages 517–526. ACM Press, 2002.

[Mic] Sun Microsystems. *J*ava trademark of Sun Microsystems. Home Page: http://www.sun.com/ .

[Mig02] S. Miglia. "Specifica ed implementazione di ruoli e protocolli d'interazione per agenti in D-CaseLP". Master's thesis, DISI,

University of Italy, Genova, Italy, 2002. downloadable from ftp://gundam.vislink.it/stefano/thesis.pdf.

[MMMZ00a] S. Marini, M. Martelli, V. Mascardi, and F. Zini. "HEMASL: A Flexible Language to Specify Heterogeneous Agents". In A. Corradi, A. Omicini, and A. Poggi, editors, *Proceedings of WOA 2000. Dagli Oggetti Agli Agenti*, pages 76–81, Parma, Italy, 2000. Pitagora editrice, Bologna.

[MMMZ00b] S. Marini, M. Martelli, V. Mascardi, and F. Zini. "Specification of Heterogeneous Agent Architectures". In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages – Proceedings of the Seventh International Workshop ATAL 2000*, pages 275–289, Boston, MA, USA, July 2000. Springer-Verlag, Berlin. Lecture Notes in Artificial Intelligence 1986.

[MMZ99] M. Martelli, V. Mascardi, and F. Zini. "Specification and Simulation of Multi-Agent Systems in CaseLP". In *Proc. of Appia–Gulp–Prode 1999*, L'Aquila, Italy, 1999.

[MOL] MOLE. *Mole mobile agent system.* Home Page: http://mole.informatik.uni-stuttgart.de/ .

[Nwa96] Hyacinth S. Nwana. "Software Agents: An Overview". *Knowledge Engineering Review*, 11(3):205–244, 1996.

[OMGa] OMG. *Mobile Agent System Interoperability Facilities* . Home Page: http://www.omg.org/docs/orbos/97-10-05.pdf .

[OMGb] OMG^{TM}. *Unified Modeling Language.*
Home Page: http://www.uml.org/.

[PW02] L. Padgham and M. Winikoff. "Prometheus: A Methodology for Developing Intelligent Agents". In *Proceedings of the Third International Workshop on Agent-Oriented Software Engineering (AAMAS 2002)*, Bologna, Italy, July 2002.

[Roy70] W. W. Royce. "Managing the Development of Large Software Systems". In *Proceedings of IEEE WESCON*, August 1970.

[RS] Inc. RECURSION SOFTWARE. *Voyager.* Home Page: http://www.objectspace.com/products/voyager/ .

[SA03] F. Stolzenburg and T. Arai. "From the specification of multiagent systems by statecharts to their formal analysis by model checking: Towards safety-critical applications". In J. Muller M.Schillo, M.Klusch and H.Tianfield, editors, *Proc. of the First German Conference on Multiagent System Technologies*, LNAI 2831, pages 131–143. Springer-Verlag, 2003.

[SCT96] L. Sterling, P. Ciancarini, and T. Turnidge. "On the Animation of "not Executable" Specifications by Prolog". *International Journal of Software Engineering and Knowledge Engineering*, 6(1):63–87, 1996.

[Sea69] J. R. Searle. *"Speech Acts"*. Cambridge University Press, Cambridge, UK, 1969.

[SIC] SICS. (SWEDISH INSTITUTE OF COMPUTER SCIENCE) *SICStus Prolog*. Home Page: http://www.sics.se/isl/sicstuswww/site/index.html.

[SLBB04] C. Schifanella, L. Lusso, M. Baldoni, and C. Baroglio. "Design and development of a visual environment for writing DyLOG programs". In M. Baldoni, F. De Paoli, A. Martelli, and A. Omicini, editors, *Proc. of WOA 2004: Dagli Oggetti agli Agenti, Sistemi Complessi e Agenti Razionali*, pages 43–50, Torino, Italy, November 2004. Pitagora Editrice Bologna.

[Til] Tilab (Telecom Italia Lab). *"Java Agent DEvelopment Framework, an Open Source platform for peer-to-peer agent based applications"*. Home Page: http://jade.tilab.com/.

[Wha] Whatis.com. *whatis.com*@. Home Page: http://whatis.techtarget.com/whome/0,289825,sid9,00.html.