Università degli studi di Genova

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Informatica

Anno Accademico 1999/2000

Tesi di Laurea

UML come linguaggio per specificare agenti: analisi, estensione ed integrazione in CaseLP

Candidato
Roberto Degl'Innocenti

Relatori
Prof. Maurizio Martelli
Dott. Viviana Mascardi

Correlatore
Prof. Maura Cerioli

A chi mi ha dato
una ragione per credere.
A chi mi ha insegnato
a credere nella ragione.

Una citazione, giusto per iniziare
"Il mio nome è Bond, James Bond."

Ringraziamenti

Eccoci alla parte in cui si tenta di chiudere i conti e saldare alcuni debiti, sia piccoli che grossi. E sento che ne ho un bel po‘ da chiudere...

Cominciamo con le persone che hanno reso possibile questo lavoro.

Ringrazio il professor Martelli per avermi mostrato l'eleganza e la potenza della programmazione logica.

La dottoressa Viviana Mascardi, oltre che per l'enorme aiuto fornitomi, per aver tentato di porre dei limiti al mio lavoro.

Il dottor Floriano Zini per avermi "preso in consegna" per buona parte di una lunga estate.

La professoressa Cerioli per aver avuto il coraggio, dopo aver corretto "L'infame" esercitazione di programmazione, di leggere la mia tesi.

Passiamo adesso alle persone che da lungo tempo si trovano ad avermi d'intorno.

Un grazie speciale va ai miei genitori, per avermi nutrito, vestito e soprattutto sopportato per tutti questi anni.

Un abbraccio ed un bacione va alla mia sorellina Ilaria per aver strenuamente cercato di ricordarmi che nella vita non esiste solo lo studio. Gran bel tentativo!

Un saluto ai miei ex-coinquilini: Max "Whistle" Bogliolo, Luca "Gunhead" Giorello, Fabio "Desparecido" Carrozzino, Ale "Pazuzu" DelBue, Andrea "Old Pic" Picarelli, Luca "Lemonface" Tricerri, Michele "Goodfella" Moretti ed i fratellini Berna & Berna. Ragazzi come è più dura la vita senza di voi. E quanti kilometri in piú!

Un abbraccio elettronico al mio compagno di deliri in rete: il mitologico F. (Anche se ancora adesso non so a quale dei gemelli mi riferisco).

Ed adesso veniamo alle persone con cui ho trascorso i miei ultimi anni di vita: i miei “compagnucci” di universitá, i mitici DISI-integrati. In stretto ordine sparso, cominciando ovviamente con le signorine:
Carla per il ruolo di interfaccia e per farmi sentire inopinatamente utile, Ileana per le liquirizie e per essere naturalmente predisposta alla follia, Marina per essere la mia amichetta pennuta, Daniela perchè mente in modo convincente, Milena per non avermi maltrattato troppo, Cristiano per la “cultura”, Emanuele per aver addolcito un pò la Ste, Emilio e Matteo per avermi presentato il signor C. M. Scoglio, Silvio per il delirio, Alberto ed il Ponta per le voci fuori campo.(se qualcuno è rimasto fuori è solo colpa della mia memoria, e non dell’affetto)
Grazie, ragazze e ragazzi, non ci crederete ma vi voglio bene!

Una nomina speciale, comunque, ai cinque che sono più al caldo nel mio “Quoricino”. Un grazie speciale a:
Lorella per essere stata sempre quella che mi trascinava coi piedi per terra e per avermi dato la forza, con l’esempio, di studiare sette giorni alla settimana.

Un saluto speciale a Stefania, amichetta ed ex-vicina di casa per le “ciattellate” e perchè sostiene che in fondo mi vuole bene, ma bisogna scavare a lungo!

Andrea: rigoroso, pedante, testardo e caparbio. Ma che mi sentirei defraudato se non ti avessi conosciuto. Ti voglio un sacco di bene, bastardo!

Riccardo per essere stato il mio contraltare in infinite, estenuanti discussioni. Spesso avevi ragione..... Forse.

Daniele ovvero il mio collega di progetti preferito (mi scuso con gli altri) e coautore di inenarrabili deliri informatici.

Una doverosa citazione va all’*umano* nominato dal LaTeX: Il nostro Guru Biagio.

Una parola a parte per le future sanguisughe sociali che sono ormai entrate a far parte della “società del DISI”: Daniela, dedita alle opere pie visto il ragazzo che si è scelta e Nicoletta, l’amica troppo figa. D’ora in poi chi vi verrà più ad importunare durante le sue pause?

Infine un doveroso ringraziamento lo devo fare alla Nostra Signora del Fato: nonostante tutto quello che mi ha fatto si merita un ringraziamento per il modo in cui ha tirato i dadi. Mi ha sempre circondato di persone stupende.

Indice

Introduzione

Esaminando le recenti evoluzioni nel campo dello sviluppo di sistemi software si possono notare due tendenze principali: da un lato l'attuale continua crescita dei sistemi software, sia in termini di complessità che in dimensione, mette a dura prova le capacità degli sviluppatori nel riuscire a padroneggiarla. Questa situazione viene addirittura resa critica dalla necessità di fornire ai prodotti sviluppati nuove capacità ormai ritenute necessarie come un comportamento intelligente, la sicurezza dei dati o l'accesso e la gestione di risorse distribuite. Si fa quindi sempre più sentire la necessità di un sistema in grado di poter suddividere questa difficoltà in termini maneggevoli.

Dall'altro si osserva l'adozione delle tecnologie informatiche in un numero sempre crescente di campi applicativi, con conseguente sviluppo di un gran numero di sistemi software che rispondono a svariate esigenze. Questi "componenti", grazie alll'esplosione della World Wide Web e di Internet che ha avuto l'effetto di rendere disponibili a chiunque una enorme mole di dati e di algoritmi, hanno causato un cambiamento delle competenze richieste ad uno sviluppatore software. Oggi infatti la bravura di uno sviluppatore software non si misura tanto in base alla capacità di *inventare* nuovi algoritmi ed *implementarli* efficientemente, quanto alla capacità di *integrare* componenti software e dati esistenti per creare nuove applicazioni in tempi brevi e minimizzando le risorse necessarie. Le caratteristiche delle componenti che verranno integrate nella nuova applicazione saranno solitamente

- distribuzione: saranno collocate non su una singola macchina ma su più macchine diverse collegate tra loro da una rete;
- eterogeneità: sono scritte in linguaggi di programmazione diversi e sviluppate su differenti piattaforme.

Prima di poter passare alla fase implementativa, lo sviluppatore software dovrà quindi affrontare i problemi di

- modellare una applicazione formata da componenti distribuite ed eterogenee; e

- verificare che il modello individuato soddisfi le specifiche della applicazione.

La *tecnologia ad agenti* [JSW98a], grazie all'alto livello di astrazione a cui si colloca (quello a livello di cooperazione della *conoscenza* tra sistemi autonomi), risulta particolarmente adatta a modellare applicazioni con le caratteristiche descritte sopra; essa consente infatti di modellare entità che comunicano (dotate quindi di *abilità sociali*), che monitorano l'ambiente in cui si trovano e sono in grado di reagire agli eventi che si verificano (*capacitá di reazione*), sono in grado di prendere l'iniziativa se la situazione lo richiede (*proattività*) senza l'intervento umano o di altri agenti(*autonomia*). Società composte da tali entità vengono definite *sistemi multi agenti (MAS)*. Esse contemplano la *distribuzione* degli agenti coinvolti e dei dati su cui operano e *l'integrazione* di software e dati eterogenei. Queste due caratteristiche sono fondamentali per il successo dei sistemi software e sono una delle ragioni per il consenso ricevuto dai MAS sia in campo accademico che industriale.

Data l'intrinseca complessità dei sistemi ad agenti, o sistemi multi-agente, sarebbe auspicabile l'adozione di metodologie standard e tecniche di ingegneria del software per seguire lo sviluppatore in tutte le fasi di sviluppo, dalla specifica informale alla implementazione.

Dovrebbe essere inoltre possibile l'utilizzo di linguaggi standard largamente diffusi per la specifica delle varie fasi di sviluppo, e l'utilizzo di sistemi di prototipazione rapida per un immediato riscontro delle scelte effettuate e delle strategie adottate.

Sfortunatamente però le metodologie, i linguaggi di specifica e gli ambienti di prototipazione rapida che supportino tali metodologie e linguaggi, restano per lo più circoscritti all ambito accademico, e non esiste nessuno standard *de facto* riconosciuto ed usato a livello industriale. La sperimentazione in questa direzione è quindi attuale e necessaria.

La presente tesi si propone quindi di verificare la utilizzabilità di alcuni linguaggi di specifica "tradizionali" e largamente usati in ambito accademico ed industriale, per la modellazione di sistemi ad agenti, e di integrare uno di essi in un ambiente di prototipazione di sistemi ad agenti esistente.

In particolare, la fase di analisi di linguaggi di specifica ha evidenziato che UML [UML00] è il più flessibile ed adatto a modellare agenti, pur non essendo nato con questo scopo. UML è un linguaggio di specifica ad oggetti che adotta un "vocabolario" grafico ed è quindi di facile comprensione anche per un non "addetto ai lavori". UML è stato standardizzato dall'OMG ed ha conosciuto un'ampia diffusione sia in campo industriale che accademico.

Il linguaggio è stato esteso con costrutti per supportare la modellazione di agenti che rispondano alla definizione data sopra. Infine, il linguaggio esteso è stato (parzialmente) integrato in CaseLP [MMZ97], un ambiente di prototipazione di sistemi ad agenti sviluppato presso il DISI di Genova. L'integrazione, una volta completata, consentirà di eseguire in CaseLP specifiche UML, e di sfruttare quindi tutti gli strumenti che CaseLP offre per la verifica ed il testing della applicazione modellata.

Struttura della tesi

Questo documento è suddiviso in tre parti: la prima parte si occupa di definire cosa viene inteso per sistema ad agenti ed esamina i linguaggi di specifica generici esistenti alla luce delle necessità di definire un sistema ad agenti; la seconda si occupa di introdurre il lettore agli ambienti di prototipazione rapida per MAS, in particolare a CaseLP; nella terza, infine, viene affrontata l'integrazione di UML all'interno di CaseLP.

Nel primo capitolo, dopo aver affrontato la definizione di MAS, e delle caratteristiche che un tale sistema impone ad un linguaggio di specifica, ci si occupa di esaminare le caratteristiche di alcuni sistemi di specifica algebrici, dandone una rapida valutazione.

Nel secondo capitolo, dopo una breve introduzione del paradigma ad oggetti, si fornisce una descrizione sintetica di UML, esamindanone rapidamente i diagrammi che lo compongono.

Nel terzo viene esaminato UML sulla base delle necessità emerse per specificare un sistema ad agenti e vengono proposte alcune estensioni del linguaggio per renderlo atto alla specifica di un MAS.

Nel quarto capitolo viene fornita una breve rassegna di alcuni ambienti di sviluppo e prototipazione per MAS, prodotti sia in ambito accademico che commerciale, per poter inquadrare le nozioni necessarie per lo studio di CaseLP.

Nel quinto si presenta CaseLP come sistema completo di specifica e prototipazione per sistemi ad agenti.

Nel sesto capitolo si presenta in dettaglio uno dei linguaggi che compongono il sistema CaseLP: HEMASL. HEMASL è un linguaggio di specifica per agenti basato sul paradigma imperativo ed è quindi particolarmente adatto agli scopi di questa tesi

Nel settimo viene illustrato un linguaggio (Text-UML) che è stato sviluppato per la rappresentazione testuale di alcuni diagrammi di UML, utile per una loro possibile traduzione ad opera di un sistema automatico.

Nell'ultimo capitolo vengono analizzati i legami che esistono tra il sistema di specifica proposto utilizzante UML ed il metodo di CaseLP e la loro integrazione. Vengono anche proposte alcune metodologie per passare da un sistema ad un altro.

In chiusura della tesi vengono presentate in appendice le sintassi formali dei linguaggi sviluppati nel corso del lavoro ed un breve esempio di definizione di un sistema ad agenti completo.

Parte I

Rassegna su linguaggi di specifica

Capitolo 1

Specifica di MAS mediante linguaggi "tradizionali"

In questo capitolo viene presa in esame la possibilità di specificare un MAS (Multi-Agent System) tramite linguaggi di specifica esistenti, non progettati per la specifica di MAS
Innanzitutto daremo una definizione di agente e esamineremo quali caratteristiche deve avere un linguaggio per poterlo specificare. Tale studio fornirà la base per estendere l'analisi a linguaggi per la specifica di un sistema multi agente. Tali linguaggi dovranno consentire la rappresentazione dell'esecuzione parallela e della concorrenza, tipiche di un MAS.
Verranno poi brevemente presentati una serie di linguaggi di specifica formale che verranno analizzati e confrontati a seguito dei requisiti che sono emersi dalle analisi precedenti.
Infine verranno suggeriti dei metodi generali per supplire ad eventuali manchevolezze presentate dai linguaggi sopra esposti.

1.1 Una definizione di agente

Prima di poter trattare un argomento è necessario avere ben chiari i concetti che verranno utilizzati. Nel campo degli agenti e dei MAS questo è, purtroppo, tutt'altro che banale. Uno dei primi problemi che siamo costretti ad affrontare è che nella comunità scientifica non è ancora stato raggiunto un accordo su una definizione di agente, una situazione peraltro similare a quella verificatasi per una delle nozioni di base nel campo dell'Intelligenza artificiale in cui non è ancora stato univocamente definito il concetto di intelligenza.

In questa tesi si è scelto di utilizzare la definizione data in [JSW98b].
Secondo tale definizione un agente software deve essere:

- **situato**: un agente è inserito in un ambiente e, soprattutto, deve ricevere input sensoriali dall'esterno ed effettuare delle azioni che mutano l'ambiente in cui si trova.
- **autonomo**: cioé
  - **i)** può agire senza l'intervento di nessuno (deve quindi essere in grado di funzionare in modo autonomo, deve avere una sua "intelligenza", in alcune definizioni più strette deve avere la capacità di imparare)
  - **ii)** deve avere il controllo delle sue azioni e del suo stato interno (a nessun altra entità del sistema deve essere permesso di interagire con le strutture interne dell'agente stesso).
- **flessibile**: che implica
  - **reattivo**: l'agente deve percepire l'ambiente che lo circonda e deve reagire rapidamente agli stimoli provenienti da esso;
  - **pro-attivo**: l'agente non deve semplicemente reagire in risposta all'ambiente ma deve poter mostrare un comportamento "opportunistico" diretto solo dai suoi scopi personali e poter prendere l'iniziativa quando serve. Deve quindi avere i suoi scopi e i suoi piani per realizzarli;
  - **sociale**: l'agente deve essere in grado di interagire con altri agenti (e con l'ambiente) quando necessario per poter risolvere i propri problemi e di aiutare eventualmente gli altri a fare altrettanto.

Nella nostra trattazione verranno considerati "agenti" anche entità che possiedono un sottoinsieme delle caratteristiche sopra elencate ma che possono essere utili ai fini pratici come, ad esempio, un agente solo proattivo o solo reattivo.

1.2 Requisiti necessari per la specifica di un agente

Analizzando la definizione di agente sopra esposta si possono fare delle considerazioni sui requisiti necessari ad un linguaggio di specifica (tralasciando in questo momento considerazioni metodologiche) per poter essere usato per definire agenti. Le necessità principali risultano essere

- **Comunicazione.**
  Per i requisiti di socialità (comunicare con altri agenti e/o entità esterne) e di essere situato (percepire eventi dall'esterno e manipolare l'ambiente che lo circonda) può essere sufficiente una buona modellazione della comunicazione verso l'esterno del sistema e verso gli altri componenti dello stesso. Infatti la ricezione di dati dai sensori e l'utilizzo di attuatori può essere modellata tramite comunicazioni da/verso l'ambiente. Sicuramente sarà necessaria una comunicazione asicrona, mentre, data la natura autonoma degli agenti (e quindi una minor necessità di sicronizzazione), passa in secondo piano quella sincrona.

- **Incapsulazione ed information hiding [1].**
  Dal punto di vista di autonomia come "riservatezza" dei dati viene molto comodo il concetto (e quindi un supporto dato dal linguaggio) di incapsulazione.

- **Modellazione dei tempi di risposta.**
  Per progettare un comportamento reattivo viene comoda la possibilità di poter fissare dei tempi di risposta a determinati stimoli.

- **Comportamento di default.**
  La proattività può venire modellata tramite un comportamento di default associato ad una serie mutevole di goals da realizzarsi. Quindi è necessario che il linguaggio fornisca una funzionalità per specificare il comportamento all'attivazione dell'agente (sul tipo di un costruttore di un linguaggio O-O, o di un main di una classe attiva Java). Viene altresì molto comodo poter disporre di una specifica dinamica del comportamento.

[1] Con incapsulazione si intende il concetto di "Black Box", una scatola nera nella quale si inserisce "qualcosa", che in qualche modo viene manipolata, ed, infine, viene restituito "qualcos'altro".
Con information hiding si intende che l'accesso ad alcune informazioni o servizi è limitato solo ad una particolare classe di utenti.

1.3 Una definizione di multi-agent system

Dopo la definizione di agente vediamo di estendere il discorso introducendo il concetto di MAS (Multi Agent System).
Un MAS può essere definito come un insieme di agenti (possibilmente eterogenei) che operano insieme per risolvere un problema che richiede abilità o conoscenze maggiori di quelle disponibili ad un singolo agente.
Gli agenti che lo compongono sono entità dinamiche, indipendenti, sociali, che possono essere di tipi dissimili, simili o uguali.
Un MAS non è semplicemente un gruppo di agenti ma un sistema complesso in cui sono utilizzati agenti che interagiscono tra loro.

I MAS sono dunque modelli ideali per rappresentare problemi che hanno più metodi risolutivi, più prospettive o che richiedono più entita necessarie per risolverlo.

Le sue caratteristiche sono

1. Un MAS è composto da agenti tra loro indipendenti.

2. All'interno del MAS ogni agente possiede/può possedere solo una visione parziale del problema, o solo un sottoinsieme delle capacità risolutiva necessaria.

3. In un MAS non esiste un sistema di controllo globale; dato che ogni agente deve essere libero di operare secondo la sua natura la coordinazione tra i singoli agenti è data da protocolli di comunicazione e negoziazione.

4. I dati sono decentralizzati, ogni singolo agente ha/può avere accesso solo ai dati a lui necessari per risolvere il sotto-problema di sua competenza.

5. Essendo composto di unità totalmente indipendenti tra di loro, la computazione è asincrona.

6. Richiede l'accesso, e la race condition risultante, a risorse condivise limitate.Gli agenti possono esibire un comportamento egoistico e tentare di accaparrarsi le risorse del sistema in cui si trovano. comunque con facilità (si potrebbe parlare di certezza se si modella un sistema reale) ci si trovera a modellare sistemi dotati di risorse finite.

Le ragioni che motivano l'interesse nei riguardi dei MAS (visti sia come modello che come sistema finale) sono molteplici:

- maggiore robustezza (il sistema è costituito da entità indipendenti);
- efficenza (da un punto di vista computazionale ciò deriva dal fatto che il sistema è distribuito; da un punto di vista progettuale si ha il vantaggio di concentrarsi su risolutori autonomi (gli agenti) facilmente riutilizzabili);
- facilità di integrazione con legacy system (un agente può funzionare da interfaccia con un pacchetto preesistente e le funzionalità aggiuntive richieste vengono fornite dagli altri agenti del sistema).

Ovviamente un tale nuovo approccio presenta anche delle difficoltà, che principalmente risiedono su un livello progettuale

- Come formulare, descrivere e decomporre i problemi o sintetizzare i risultati ottenuti dai singoli agenti?
- Come far comunicare ed interagire gli agenti di un sistema?
- Come coordinare gli sforzi di un gruppo di agenti?

Ma, forse, il problema principale è: *come ingegnerizzare e testare un MAS*?
Esaminiamo quindi cosa si rende necessario , o comunque auspicabile, per avere a disposizione un linguaggio di specifica utilizzabile per un sistema ad agenti.

1.4 Necessità per un sistema multi-agente.

Tra le caratteristiche necessarie ad un linguaggio per specificare un MAS si possono trovare:

- Specifica di processi concorrenti e computazione parallela: gli agenti devono essere visti come processi tra loro totalmente indipendenti, che si possono trovare su una o più macchine all'interno di una rete.

- Necessità di specificare oggetti dinamici: un agente è un entità dinamica di un sistema: può essere creato solo quando serve e distruggersi quando la sua utilità termina.

- Specifica del protocollo di comunicazione : per far sì che gli agenti comunichino è necessario specificare un linguaggio comune di comunicazione (es KQML), e specificare i protocolli utilizzati per le comunicazioni. Questo è di importanza fondamentale dato che andranno definite strategie di comunicazione complesse per ottenere cooperazione, coordinazione e negoziazione tra gli agenti stessi.

- Allocazione di risorse finite: dato che, come già detto, ogni agente è un'entità a se stante che può esibire un comportamento opportunistico, diviene auspicabile poter modellare l'utilizzo delle risorse del sistema in cui si opera per poter definire un sistema per il controllo, allocazione e condivisione delle risorse e poter definire risorse limitate e *consumabili*)

- Classi ed ereditarietà: tali concetti risultano utili per definire agenti con caratteristiche uguali o simili.

Riassumendo le necessità emerse dallo studio delle specifiche di agenti e MAS sono:

1. Modellazione della comunicazione.

2. Modellazione del parallelismo e della concorrenza.

3. Modellazione degli eventi provenienti dall'esterno del sistema.

4. Modellazione di un comportamento Environment e Goal driven.

5. Allocazione e condivisione di risorse finite.

6. Definizione di oggetti/agenti in modo dinamico.

7. Modellazione dell'incapsulation e dell'information hiding.

8. Supporto al paradigma Object Oriented (vedi 2.1), oltre a ciò sopra richiesto.

9. Modellazione dei tempi di risposta ad almeno alcuni messaggi.

Il nostro approccio alla valutazione di quali linguaggi di specifica esistenti, nati con una finalità diversa da modellare e specificare MAS, possono esere meglio riutilizzzati per questo scopo è il seguente: visto che il supporto della concorrenza è una delle caratteristiche più importanti, analizzeremo diversi linguaggi per la specifica di sistemi concorrenti e per ciascuno di essi valuteremo come supporta le caratteristiche individuate sopra (comunicazione, modellazione di eventi provenienti dall'esterno...).
Una tabellina al termine di ogni sezione descrive sinteticamente le caratteristiche supportate da ogni linguaggio.

1.5 Modelli per la specifica della concorrenza

Quando si vuole specificare un sistema concorrente è necessario pensare a che modello di calcolo concorrente si intende usare; considerando che non esiste un singolo modello in grado di catturare tutti gli aspetti rilevanti che formalizzazano un processo, è necessario effettuare una scelta in base alle necessità del sistema che dobbiamo specificare.

I principali modelli di specifica per sistemi concorrenti (vedi[AR91])

- Labelled Transition System: Proposti da Milner e Plotkin [Mil89, Hoa85] hanno le seguenti caratteristiche:

  - sono adatti ad un approccio composizionale;
  - non sono molto validi per l'astrazione (soffrono di una forte tendenza alla sopraspecificazione);
  - presentano una certa difficoltà nel definire una relazione di congruenza che sia semplice (relazione che è necessaria per poter definire il processo di raffinazione di una specifica);
  - non consentono (almeno in CCS) di passare una label come valore ad un sottoprogramma CCS (dato che le transizioni sono etichettate a priori): un problema classico è la chiamata di funzione all'interno

di 2 processi in parallelo: a quale dei due processi và inviata la risposta?

- Event Structures: modellano un processo considerando un insieme di fatti che possono avvenire durante l'attività chiamati *eventi* sulla base dei quali vengono definite delle relazioni che descrivono il comportamento del processo.
  - Non sono adatti ad un approccio composizionale
  - sono buoni per l'astrazione (permettono di concentrarsi solo sugli eventi di interesse)
- Petri Nets: le reti di Petri sono uno dei sistemi più conosciuti ed usati e hanno subito molte specializzazioni con il passare del tempo. Tramite esse si modella il sistema nel suo complesso e, normalmente, non il singolo processo.
  Come caratteristiche hanno:
  - buona astrazione e semplicità e chiarezza nelle operazioni di raffinamento;
  - assenza (quasi totale) della capacità di composizione.

  Sulla base delle PN sono state definite le Process Algebras; il potenziale espressivo di questa classe di algebre è minore a quello delle PN complete ma hanno il vantaggio innegabile di una miglior formalizzazione.
- Finite States Automata: Le FSA sono una sottoclasse delle reti di petri. Una forma particolare di esse è stata lungamente usata nella definizione di funzioni (ad esempio nella definizione di linguaggi).
  - Consentono una buona astrazione (ci si può limitare agli eventi di interesse) e raffinamento (se si utilizza la nozione di submachine).
  - Con la specifica di un elemento alla volta, ossia di una singola funzione per volta, permettono di comporre le specifiche.
  - Il principale difetto è che si definisce il singolo componente e che quindi è necessario una sorta di super-sistema per la composizione dei componenti nell'intero sistema.

- Dataflows: I dataflow usano l'astrazione della blackbox: un processo è una scatola in cui entrano dei valori lungo linee di input e ne fuoriescono altri lungo canali di output, il tutto in modo asincrono.
  - buoni per la astrazione
  - buoni per composizione
  - difetto principale: permettono di modellare semplicemente solo comunicazioni asincrone

I linguaggi di specifica della concorrenza (che adottano uno dei modelli sopra esposti) possono essere divisi in due grandi classi a seconda dell'approccio utilizzato per la specifica di un processo
I) Model Oriented: il linguaggio fornisce un modello del sistema specificato, e tramite esso si specificano tutti i modelli ad esso equivalenti.
Esempi di questo approccio: CCS, CSP, ...
II) Property Oriented: un modello viene specificato attraverso un insieme di proprietà, tutti i modelli che abbiano tali proprietà sono modelli corretti del sistema preso in esame (questo approccio presenta molte somiglianze con la normale specifica degli ADT tramite algebre con predicati)
Esempi di tale approccio: modal/temporal logic (usati con LTS,ES,PN), FOL (dataflow)

La seconda classe di linguaggi sembra essere più adatta per lo scopo preposto dato che un tale approccio viene naturalmente mappato in un modello/prototipo concentrandosi sulle proprietà ricercate

1.6 Linguaggi di specifica per sistemi concorrenti

I linguaggi di specifica presentati qui di seguito sono stati esaminati prendendo in esame:

- il linguaggio di definizione dei dati cioè come i dati possono essere specificati nel linguaggio.
- Il modello di comunicazione cioé se il linguaggio permette la comunicazione sincrona, asincrona od entrambi.

- Il modello di concorrenza e parallelismo del linguaggio.
- le eventuali facilities che il linguaggio offre:
  per modellare eventi (cioé comunicazioni con l'ambiente esterno);
  per definire una computazione attiva goal driven;
  meccanismi per l'encapsulation e l'object orientation
  sistemi per specificare transizioni temporizzate.

1.6.1 LOTOS

Il principale motivo di interesse di questo linguaggio è dato dal fatto che esso è stato standardizzato dall'OSI per la definizione di sistemi concorrenti.[Tur]

Il modello sottostante è essenzialmente fornito dal CCS (con qualche aggiunta presa dal CSP) e ActOne (usato come Data Definition Language).
ActOne è essenzialmente un linguaggio di definizione di algebre con predicati del I ordine (visti come equazioni booleane); sono ammesse equazioni condizionali. Permette l'overloading ed i tipi parametrici. La sua semantica di è quella delle algebre iniziali.

L'unità principale di Lotos è il processo ed il modello di comunicazione tra essi è dato dai punti di intersezione tra processi (GATES). Questi gates sono grosso modo canali di comunicazione, ma i parametri che sono passabili rappresentano un'estensione di quelli del CSP.
Dato che il linguaggio di definizione è un sovrainsieme del CSP il modello sottostante è quello degli alberi di transizione etichettati. Questo permette quindi di concentrarsi primariamente sulla sincronizzazione dei processi stessi.
È da notare che vi è una distinzione netta tra i processi ed i dati: i dati sono utilizzati dai processi ma non è possibile associare dei processi a degli ADT (*abstract data type*). Questo rende perlomeno difficile la definizione di una struttura simile ad una classe ovvero un ADT con un behaviour specificato.
Gli eventi in lotos rappresentano una sincronizzazione tra processi mentre un'offerta da parte di un processo di partecipare ad un evento è vista come una capacità (esattamente come in CCS).
Come in CCS lotos offre un ordinamento temporale degli eventi (visti come alberi).

Da queste sue caratteristiche si desume che:

- Lotos può gestire con facilità la comunicazione sincrona (la asincrona deve essere simulata/modellata per mezzo di buffer/code di messaggi)
- Il modello di parallelismo è quello dell'interleaving.
- Permette l'overloading ed i tipi parametrici.
- Non fornisce nessun supporto nativo per una definizione Object Oriented.

Data Utilities	overloading; tipi parametrici
Comunicazione	sincrona
parallelismo	interleaving
Eventi	SI
Encapsulation/info. hiding	??
Object Oriented	NO

1.6.2 DisCo

DisCo (Distribuited Cooperation) [JKS91] è un linguaggio di specifica semi-eseguibile nel senso che una specifica può essere direttamente usata in un interprete per testarla.
Il linguaggio permette di definire sia il sistema che l'ambiente in cui il sistema si troverà ad operare (tale ambiente viene chiamato creation), questo si rende necessario per il testing del sistema. All'interno del sistema è possibile definire funzioni ed asserzioni globali sia condizioni iniziali.
Le entità di base di DisCo sono gli oggetti e le azioni:

- Gli oggetti sono strutture dati con la capacità di partecipare a determinate azioni.
- Le azioni esprimono unità atomiche di esecuzione;

I cambiamenti dello stato di un oggetto sono permessi solo tramite partecipazione ad una azione, quindi tutte le transizioni sia interne che esterne sono viste come azioni.

DisCo permette la definizione di classi delle quali è necessario specificare:

1. **State**: gli insiemi degli stati raggiungibili dalle classi (visti come stati di una FSA).

2. **Vars**: le variabili utilizzate dalla classe.
3. **State extension**: i possibili sottostati di stati predefiniti.
4. **Inheritance**: le classi ereditate sono viste come un singolo stato che ha come sottostati gli stati della classe ereditata.
5. **Local assertion**: asserzioni (predicati che devono essere valutati veri durante l'esecuzione del "programma").
6. **Local initial condition**: specifica le condizioni iniziali della classe.

Per quanto riguarda le azioni esse definiscono i cambiamenti di stato che avvengono negli oggetti in seguito all'avverarsi di determinate condizioni.
Sono definiti tramite NOME, STATE TRANSITION, CONDIZIONE.
Può quindi essere vista come la definizione di una FSA suddivisa nelle due parti stati, transizioni.
La definizione di un sistema è quindi data dalla definizione delle sue classi e le azioni che intercorrono tra le classi stesse (vedendo le azioni interne come casi particolari).

Questo sistema evidenzia che la comunicazione è basilarmente sincrona, come una chiamata di metodo.
DisCo fornisce utilities per:

- valori di default,
- inheritance,
- Hiding/encapsulation.

Data Utilities	overloading; valori di default
Comunicazione	sincrona
parallelismo	interleaving
Eventi	SI
Encapsulation/info. hiding	SI
Object Oriented	SI

1.6.3 Z/objectZ

Object Z è basato su Z e quindi esaminiamo prima Z che fornisce la base per entrambi i metodi di specifica.

Z

Z [Spi87] è un linguaggio di specifica "tradizionale". Si focalizza quindi sulla definizione di ADT e non tratta la definizione di processi.
Il linguaggio è basato su algebre con predicati in FOL ed adotta un meccanismo denominato schema calculus che suddivide la singola specifica in "pezzi" definiti schemi e che vengono utilizzati per definire sia gli aspetti statici che dinamici di una classe.
Per definire il comportamento di una classe/metodo e per determinare il legame tra input-output viene utilizzato un sistema di tipo pre-post.
La definizione di funzione utilizza invece la normale definizione matematica di funzione (compresa lambda-astrazione).
Z fornisce ADT predefiniti.
Non ha nessun tipo di supporto nativo per la comunicazione-concorrenza ne per definire comportamenti pro-attivi.
Riassumendo in poche parole Z offre solo un linguaggio per definire algebre con predicati.

Object-Z

Object-Z [DKRS91]introduce i concetti di classe (visto come un insieme di schemi) ed oggetto e fornisce un supporto alle nozioni di

- ereditareità,
- aggregazione,
- polimorfismo,
- le classi sono dotate di una funzione init (che permette più di definire un comportamento iniziale) e
- supporta il concetto di riferimento (stile puntatore).

Le operazioni di una classe possono venire definite aggregando le operazioni preesistenti (della classe stessa o di oggetti contenuti) con:

- $\bigwedge$ (and) (le due operazioni procedono in parallelo con lo stesso input),
- $\bigvee$ (or),
- f1;f2 (operatore sequenziale),

- f1 || f2 (composizione parallela sincrona con l'output di f1 come input di f2),
- f1 [] f2 (scelta non deterministica).

Tramite questi operatori diventa possibile definire degli alberi di transizione.

È possibile definire degli invarianti sulle operazioni o sullo stato delle classi tramite un subset degli operatori di logica temporale.

Data Utilities	overloading; tipi parametrici
Comunicazione	sincrona
parallelismo	??
Eventi	SI
Encapsulation/info. hiding	SI
Object Oriented	SI

1.6.4 CafeOBJ

CafeOBJ [ATN] è un linguaggio semi-eseguibile basato sulla logica equazionale order sorted (estensione del many sorted con ordinamento sui sort), transizioni (intese come equazioni unidirezionali utilizzando la rewriting logic), hidden sort (basato sulle hidden agenda algebras).
La hidden sort porta a definire due forme di equivalenza: la term equivalence (la "normale" equivalenza) e la behaviour equivalence (2 termini sono uguali sse si comportano allo stesso modo in ogni contesto).
L'order sorted logic permette di definire in maniera naturale il polimorfismo e la relazione di sottoclasse (permette quindi di avere qualcosa di similare al "late binding").
Si può utilizzare sia una semantica design che una semantica loose.
L'unità base di cafeOBJ è il modulo che è composto dai seguenti blocchi:

- **Import**: dichiarazione di importazione (con tre possibili sistemi protecting, extending, using).
- **Signature**: composto da Sort (i tipi usati), subsort (le relazioni d'ordine tra sort), operator declaration (le dichiarazioni degli operatori), behavioural operator declaration (operatori sugli hidden sort), predicate declaration.

- **Assiomi**: equazioni, conditional equation, transition; ognuno di essi ha la sua controparte behaviouristica, ossia sui sort nascosti.

Per la specifica della concorrenza utilizza appunto la rewriting logic e la definizioni di transizione (questo permette di definire sistemi di transizioni che ricordano i LTS).

La comunicazione viene realizzata tramite la chiamata di metodo (questo permette di definire con facilità comunicazioni sincrone e sussiste il "solito" problema per le comunicazioni asincrone).

Il meccanismo di parallelismo non è chiaramente esplicitato.

Un'altra mancanza è che non sembra essere presente il concetto di oggetto vero e proprio ma solo di classe.

Data Utilities	overloading; tipi parametrici
Comunicazione	sincrona
parallelismo	interleaving (probabilmente)
Eventi	SI
Encapsulation/info. hiding	SI
Object Oriented	SI

1.6.5 FOOPS

FOOPS [BG95] è un estensione di obj realizzata per decrivere sistemi concorrenti ad oggetti. È basato su OBJ, quindi il sistema di tipo è quello delle order sorted logic.
In FOOPS la specifica dei dati avviene su due livelli:

- Il livello degli ADT (definito *functional level*) che utilizza essenzialmente delle algebre con predicati.

- Il livello degli oggetti (*Object level*) che permette di definire classi, considerate come collezioni di oggetti potenziali, ognuno unico, con un suo stato ed un suo comportamento.

Le classi si differenziano dagli ADT per la presenza di

1. Attributi (elementi di un sort, anche di sort funzionali) che possono essere dichiarati hidden e quindi non disponibili (visibili) all'esterno; gli attributi di tipo funzionale sono funzioni "pure" che quindi non modificano lo stato.
   Gli attributi a loro volta si dividono in *stored* (lo stato vero e proprio) e *derived* (cioè calcolati a partire dai valori di stato).

2. Metodi (ossia funzioni che modificano lo stato).

In una classe gli assiomi sono anch'essi divisi in due tipi:

- *direct* che assegnano direttamente un valore allo stato e
- *indirect* il cui risultato viene dall'elaborazione di altri metodi o attributi.

Viene permessa la definizione di classi/ADT parametrici.
Permette la creazione e distruzione dinamica di oggetti (i quali sono contraddistinti da un OID [2] unico)

La modellazione della concorrenza avviene tramite combinatori di metodi tra cui

- _ ; _ combinatore sequenziale;
- [_] operazione atomica;
- RESULT _ ; _ fornisce il risultato della computazione ed esegue il comando successivo;
- _ || _ composizione parallela;
- _ []_ scelta non deterministica;
- if _ then _ else _ fi composizione condizionale;

Il modello di comunicazione è quello della chiamata di metodo.
Il modello di parallelismo è quello dell'interleaving.

Data Utilities	overloading; tipi parametrici
Comunicazione	sincrona
parallelismo	interleaving
Eventi	SI
Encapsulation/info. hiding	SI
Object Oriented	SI

[2] Object IDentifier.

1.6.6 SMoLCS

SMoLCS [RP94] è un estensione rivolta verso la specifica di sistemi concorrenti del classico approccio delle specifiche algebriche con predicati del primo ordine ed uguaglianza.
La definizione dei tipi di dati (ADT) utilizza il classico approccio algebrico.
Un ADT è definito da:

- una famiglia di insiemi denominati *carriers*;
- alcune costanti, cioé elementi particolari appartenenti ai carriers;
- alcune funzioni per manipolare gli elementi dei carriers;
- dei predicati per definire condizioni sugli elementi dei carriers

La definizione è separata in una parte sintattica (la segnatura) e in una parte semantica , che fornisce l'interpretazione degli elementi della segnatura.
La semantica attribuibile alla specifica puo essere sia *loose* () che *design* (utilizzando un modello iniziale).
La specifica può essere sottoposta a raffinamenti successivi (sottoposto al vincolo algebrico *no junk, no confusion*) e permette di utilizzare un approccio modulare.
Esiste anche la possibilità di definire tipi parametrici, l'overloading delle funzioni ed una forma di controllo sulle funzioni esportate.

La parte dinamica della specifica utilizza come modello i sistemi di transizione etichettati (LTS).
Essa viene data definendo dei *dynamic sort* e delle etichette su di essi (*label*) su cui possono essere definiti predicati di transizione definiti come:
S-l→S' ove S,S' appartengono a Dsort e l appartiene a label.
Sui dynamic sort SMoLCS permette anche di usare operatori di logica temporale.
I dynamic sort rappresentano quindi gli stati utilizzati nelle transizioni. In tal modo si ottiene per ogni sort l'elenco della capabilities del sort stesso (esattamente come in una definizione CCS - CSP)

Non ha supporto per il paradigma object oriented e non supporta il polimorfismo.

Il modello di parallelismo è quella dell'interleaving e la comunicazione (data la derivazione dal CCS-CSP) è sincrona.

Data Utilities	overloading; tipi parametrici
Comunicazione	sincrona
parallelismo	interleaving
Eventi	SI
Encapsulation/info. hiding	SI
Object Oriented	NO

1.7 Metodi per sopperire alle eventali carenze del linguaggio

Si possono adottare alcuni metodi "standard" per sopperire alle eventuali carenze di un linguaggio di specifica per ottenere alcune delle caratteristiche ricercate.

- Comunicazione
  È sempre possibile modellare la comunicazione (sia sincrona che asincrona) mediante ADT/Classi mailboxes che ogni agente deve contenere al suo interno
  SVANTAGGI: non utilizzabile semplicemente per modellare sistemi event driven, non possibile modellare eventi interrupt (per ottenere il messaggio l'agente deve in qualche modo fare richiesta alla mailbox)

- Comunicazione sincrona/asincrona
  È sempre possibile modellare la comunicazione sincrona tramite l'asincrona e vice versa. La modellazione asincrona tramite la sincrona si modella tramite l'utilizzo di mailbox sincrone. Il caso inverso si ottiene immettendo cicli di attesa nel meccanismo di comunicazione.

- pro-attività
  La pro-attività in una visione imperativa si può implementare come un ciclo infinito in cui le azioni da svolgere vengono scelte in funzione di un database di fatti o di goals da raggiungere.
  Vi è però la necessità di definire qualcosa di simile ad un processo, un oggetto (dotato di costruttori) o di una classe attiva, cioé di qualcosa che

non viene attivato dall'esterno.
È necessario avere un ciclo infinito che viene attivato alla creazione dell'agente e che si occupa di implementare il comportamento pro-attivo. (in caso di necessità per funzionamenti complessi da eseguirsi in più fasi si può pensare ad una lista di goals che vengono eseguiti in sequenza con un puntatore a quello attualmente usato... usando una struttura dinamica si possono aggiungere, modificare, cancellare anche tali goal in modo tale da simulare un comportamento riflessivo)
VANTAGGI: sempre applicabile, particolarmente facile da usare in linguaggi ad oggetti.
SVANTAGGI: necessita di costruire strutture ad hoc, se il linguaggio non supporta i costruttori.

- allocazione dinamica degli agenti
  Se il linguaggio non permette strutture dinamiche si possono modellare tramite l'allocazione massima di agenti nel sistema e inserire costrutti come flag di on/off e similari per ottenere effetti simili.
  SVANTAGGI: aumenta la pesantezza della specifica poiché necessita di costrutti artefatti; si deve conoscere a priori il numero di agenti utilizzabili.

- risorse finite
  Le risorse finite sono modellabili tramite ADT/Classi che rappresentano la risorsa in questione con contatori, liste, array ... che rappresentano la finitezza della risorsa stessa.
  SVANTAGGI: appesantimento della specifica con classi non necessarie (addirittura che fanno parte dell'ambiente in cui il sistema verrà inserito)

Alla luce di queste considerazioni si può dire che i requisiti più importanti di un linguaggio di specifica da utilizzarsi per un sistema ad agenti siano la possibilità di definire (in ordine di importanza):

1. computazione parallela

2. comunicazione
3. risorse finite
4. dati/strutture dinamiche
5. object orientation

Capitolo 2

UML: lo standard “de facto”

In questo capitolo viene brevemente descritto lo Unified Model Language (UML). Ai fini della tesi verrà usato solo un sottoinsieme del linguaggio, con alcune aggiunte inserite per poter specificare un sistema ad agenti. Prima comunque di affrontare la definizione del linguaggio ricordiamo i concetti principali del paradigma ad oggetti.

2.1 Una breve definizione del paradigma ad oggetti

Ricordiamo le caratteristiche salienti del paradigma ad oggetti.
Il concetto base è quello dell’oggetto: un oggetto è contraddistinto da tre caratteristiche principali

- Identità: ogni oggetto deve essere un entità unica e distinguibile da ogni altra nell’universo del discorso.

- Stato : ovvero le caratteristiche proprie del singolo oggetto, in un preciso istante di tempo. Lo stato di un oggetto può essere visto anche come il risultato cumulativo del suo behaviour ovvero il risultato di tutte le azioni intraprese dall’oggetto stesso.

- Behaviour : ovvero come un oggetto agisce e reagisce, in termini di cambiamento di stato e di passaggio di messaggi. Il behaviour di un oggetto è funzione del proprio stato (e quindi della propria storia) oltre che degli stimoli (messaggi) esterni.

Per "proteggere" lo stato di un oggetto da "manipolazioni" esterne non consentite (e rafforzare cosìun concetto di "identità") si utilizzano le nozioni di **incapsulazione** (*encapsulation*) ed **information hiding** che permette di ottenere

- Modularità : La non trasperenza verso l'esterno dei meccanismi interni di un oggetto implica che i cambiamenti della specifica interna, o del comportamento, dell'oggetto non impongono modifiche al resto del sistema che interagisce con esso; quindi ogni singolo oggetto può essere specificato e studiato in modo indipendente dal resto del sistema. Questo consente una naturale suddivisione del lavoro.

- Riuso : dato che per ottenere i servizi di un oggetto si deve utilizzare un'interfaccia pubblica, a cui gli altri oggetti devono riferirsi per comunicare con esso, l'utilizzo di un oggetto è slegato dal programma attualmente in sviluppo. Quindi un oggetto può facilmente essere riutilizzato in un altro sistema.

Normalmente un oggetto da solo non è di grande utilità ma assume il suo ruolo una volta inserito in un sistema che comprende più oggetti, utilizzati come componenti di un sistema complesso. Attraverso le interazioni di tali oggetti si realizzano funzioni di alto livello e comportamenti complessi.
Tali interazioni, per avvenire, necessitano dello scambio di informazioni e nella richiesta di funzionalità di un oggetto da parte di un altro. Tale comportamento viene modellato tramite la metafora dello *scambio di messaggi*. In tale metafora ogni richiesta di informazioni o di "lavoro" da parte di un oggetto ad un altro viene considerato come un messaggio (o una serie di messaggi) che intercorre tra i due.

Nel mondo reale praticamente ogni oggetto appartiene ad un qualche genere. In una terminologia object oriented si dice che un oggetto (esempio: questa tesi) è un istanza di una *classe* (esempio: Tesi [1]). Tutti gli oggetti appartenenti ad un stessa classe hanno lo stato e il comportamento in comune (esempio: tutti le tesi hanno una bibliografia (stato) che può essere consultata (comportamento)). Ogni singolo oggetto della classe mantiene comunque un proprio stato indipendente dagli altri e tale stato può (entro termini prefissati)

[1] Seguendo una "tradizione" dei linguaggi ad oggetti le classi sono scritte con l'iniziale maiuscola.

differire da quello di qualunque altro oggetto della sua classe.

Proseguendo su tale metafora possiamo facilmente renderci conto che le classi possono a loro volta essere raggruppate in costrutti maggiormente generici. Tale tecnica viene chiamata **ereditarietà** (*inheritance*) e si dice che una classe A è superclasse di una classe B se la classe B è una specializzazione di A. Proseguendo l'esempio precedente le tesi sono sottoclasse di una classe Testo e tutti gli attributi di un testo (ed i suoi comportamenti) si ritrovano in una Tesi (e.g. Tutti i Testi sono composti di Parole e possono essere letti).

Questo porta ad una particolare forma di **polimorfismo**: tutti i comportamenti definiti da una superclasse devono essere validi per le classi figlie. Ma tali comportamenti devono essere applicati ad oggetti di tipo diverso ed avere, possibilmente, risultati diversi (Esempio: supponiamo l'esistenza di una classe Persona con il comportamento "fornisciInformazioni" definito, e tale comportamento fornisce il Nome e l'Età dell'oggetto; e l'esistenza di una sottoclasse Lavoratore in cui "fornisciInformazioni" deve essere definito e fornire in piú il lavoro svolto)

La relazione di ereditarietà introduce una relazione gerarchica tra le classi che sottende anche una relazione d'ordine tra i tipi: ogni sottoclasse è anche sottotipo della sua superclasse. Ciò implica che un istanza di una sottoclasse può essere sostituita a quella di una superclasse. Questa proprietà può portare ad avere un oggetto che appartiene a tipi diversi a compile time che a run time e la necessità , in caso di ridefinizione di metodi, di applicare quello dell'oggetto effettivamente utilizzato. esempio immaginiamo di aver definito la classe Shape dotata di metodo draw e la sottoclasse Rectangle in cui il metodo draw viene ridefinito per tracciare, perl'appunto, un rettagolo. supponiamo altresìsi avere una variabile di tipo Figure a cui viene assegnato un istanza di Rectangle. Se su tale variabile viene effettuata una chiamata a draw cosa accade? Quello che vorremmo è che venisse effettivamente tracciato un rettangolo. Il meccanismo per fare si che questo sia il risultato ottenuto viene denominato **late binding** ossia viene ritardato il legame della chiamata di metodo non viene effettuato in fase di compilazione ma a run time.

L'esempio fatto sopra permette di introdurre un altra utile nozione del paradigma ad oggetti: quella di **classe astratta**. Con classe astratta si indica una classe in cui non tutte le operazioni sono state definite ma che di esse ne è stata solo data la segnatura. Le classi astratte non possono essere ovviamente istanziate ma essere adoperate solo come superclassi di una classe istanziabile. Nell'esempio precedente la classe Shape sarebbe utilmente definita come classe astratta con il metodo draw non definito.

Un ultimo concetto che viene utilizzato più avanti è quello di classe parametrica o **template**: una classe parametrica è una classe che utilizza dei paramentri all'interno della sua definizione.
I parametri possono essere di due tipi

- parametri di tipo : ossia sono la dichiarazione di una variabile di tipo che deve essere instanziata in seguito per poter rendendere effettiva la classe, un esempio classico è List<T> che verrà istanziata come List<int>, List<char> ...

- parametri di valore: rappresentano invece ad esempio se si realizza la classe Limited_Stack<Limit:int = 10> si potrà utilizzare direttamente la classe Limited_Stack e il limite sarà quello di default. Se si dichiara Limited_Stack<20> il limite sarà posto a 20.

Un ultima definizione utile è quella delle **classi attive** (*active class*). Con tale termine si indicano quelle classi che possiedono un proprio thread e sono quindi considerabili come processi autonomi. Introdotte in Java esse dispongono di un metodo particolare (il *main* in una classe attiva Java) che viene attivato al momento dell'istanziazione di un oggetto di tale classe.

Se si ricorda la definizione di agente in 1.1 e le necessità che essa portava ad un linguaggio di specifica si puo quindi capire che una specifica ad oggetti fornisce quindi molti utili aspetti per lo scopo preposto.

2.2 Introduzione

Lo Unified Modeling Language (UML) è un linguaggio di specifica semi-formale ad oggetti, per visualizzare, costruire e documentare le specifiche di sistemi software, come anche di modelli commerciali e altri sistemi non software.
Il linguaggio ha avuto un grande successo commerciale e la sua diffusione è stata aiutata dalla presenza di strumenti C.A.S.E. (Computer Aided Software Engineering) che ne supportano l'utilizzo come:

- Visual UML (Tool di sviluppo che, oltre a fornire funzionalità per disegnare i diagrammi, supporta la progettazione orientata verso Visual Basic, e la generazione automatica di codice).
- Together (Strumento similare che, oltre a supportare la specifica in UML, genera e supporta sia codice Java che C++. Utilizza la totalità dei diagrammi di UML)
- Javelin (Tool che supporta solo la parte statica di UML; fornisce unicamente supporto verso Java)
- Rational Rose (Il Tool della Rational fornisce ambienti di sviluppo per i linguaggiC++, Java e Visual Basic; supporta UML al completo)

UML nasce dagli sforzi di Grady Booch, James Rumbaugh, and Ivar Jacobson per trovare un linguaggio che unificasse le metodologie di specifica da loro elaborate precedentemente (Booch, OMT, and OOSE).
Molti concetti non presenti nei linguaggi da cui deriva sono stati aggiunti,tra cui

- meccanismi di estensione (stereotipi, tagged values, and vincoli),
- threads e processi,
- distribuzione and concorrenza,
- patterns/collaborations,
- activity diagrams (per modellare processi economici),
- raffinamento (per trattare la relazione tra diversi livelli di astrazione),
- interfacce and componenti,
- un linguaggio per i vincoli.

Il progetto, iniziato in forma privata presso la Rational, venne in seguito preso sotto la tutela dell OMG che ha portato all'attuale definizione di UML.
Una parte del successo di UML è dovuto al fatto che si proponeva come successore di metodologie ampiamente affermate, cosicché è stato possibile per gli utilizzatori riutilizzare il know-how precedentemente acquisito nel nuovo linguaggio.

Un’altra delle motivazioni che hanno portato al successo UML è l’immediatezza di molti dei suoi diagrammi, che possono essere compresi anche da un non esperto. Questo è particolarmente utile nel campo dell’ingegneria del software in cui spesso le informazioni devono essere condivise da figure professionali molto diverse tra loro (come ad esempio il progettista ed il cliente).
Per realizzare l’astrazione necessaria ad un linguaggio di specifica, UML utilizza il concetto di dividere la progettazione secondo una serie di punti di vista quasi indipendenti, realizzabili a diversi livelli di dettaglio. Tali punti di vista sono esplicitati dai diagrammi che compongono una specifica UML:

- use case diagram
- class diagram
- behavior diagrams:
  - statechart diagram
  - activity diagram
- interaction diagrams:
  - sequence diagram
  - collaboration diagram
- implementation diagrams:
  - component diagram
  - deployment diagram

2.3 Uno sguardo ai diagrammi

Esaminiamo ora brevemente i vari diagrammi di cui è composta una specifica UML entrando in maggiori particolari per i diagrammi che verranno utilizzati nel seguito della tesi. Tutta questa parte è tratta da [UML00]

2.3.1 Use case diagram

Gli ***use case diagram*** mostrano attori e *use case* insieme con le relazioni che tra loro intercorrono. Gli use case rappresentano le funzionalità di un sistema, inteso come un sistema harware-software completo, o di una classe, ed esprime come tali funzionalità si manifestano nei riguardi di utenti esterni.

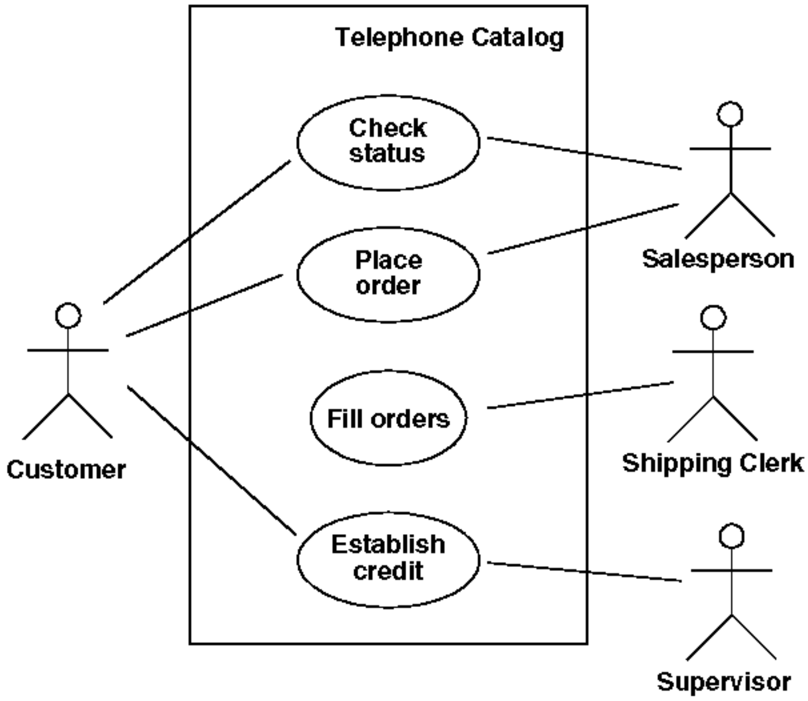

Figura 2.1: Uno use case diagram.

Come definizione uno *use case* è un tipo di *classifier* che rappresenta un insieme coerente di funzionalità fornite da un sistema, o da una classe.
Un **Classifier** è il metamodello superclasse[2] di classe, tipo di dato, ed interfaccia. Tutte queste entità sono denotate da una sintassi similare (il rettangolo) raffinato da parole chiavi dove necessario.
Data la loro maggior frequenza le classi saranno indicate dal rettangolo senza specificatori.

[2] Con ciò si intende che è il modello ad un livello di astrazione superiore il cui raffinamento porta ad ottenere i modelli voluti

Un attore, indicato con un omino stilizzato, definisce un insieme coerente di ruoli che un utilizzatore puó assumere quando interagisce con una entitá. Esistono alcune relazioni standard tra *use case* o tra attori ed *use case*:

- **Association**. La partecipazione di un attore ad uno *use case*, cioé le istanze dell'attore e le istanze dello *use case* comunicano l'uno con l'altro. Questa è l'unica relazione possibile tra attori e *use case*.

- **Extend**. Una relazione di *extend* da uno *use case* A ad uno *use case* B indica che B può essere arrichito dai comportamenti specificati in A.

- **Generalization**. Tale relazione (che può essere tra due *use case* come tra due attori) indica che A è una specializzazione di B.

- **Include**. Una relazione di questo tipo dallo *use case* A allo *use case* B indica che un istanza di A contiene anche i comportamenti specificati da B.

2.3.2 Class diagram

Un ***class diagram*** è un grafo di classificatori (*Classifier*) connessi tra loro da relazioni statiche. E da notarsi che in un *"class" diagram* possono apparire anche interfacce, package, relazioni ed anche istanze dei suddetti. Un nome più corretto sarebbe forse diagramma delle strutture statiche.

Una **classe** rappresenta un concetto all'interno del sistema che si sta modellando ovvero il descrittore di una serie di oggetti con struttura dati, comportamento e relazioni simili.
In UML il modello viene descritto intensionalmente, cioé tramite le regole che lo definiscono, mentre l'esecuzione provvede la sua estensione cioé le sue istanze. UML fornisce la notazione per dichiarare le classi e le loro proprietà, cosi come l'utilizzo delle stesse all'interno di un sistema.
Notazionalmente una classe è rappresentata tramite un rettangolo suddiviso in tre compartimenti separati da linee orizzontali. Il compartimento superiore (detto name compartment) contiene il nome e altre proprietà generali della classe; il compartimento di centro contiene la lista degli attributi; l'ultimo compartyimento contiene la lista delle operazioni definite sulla classe.

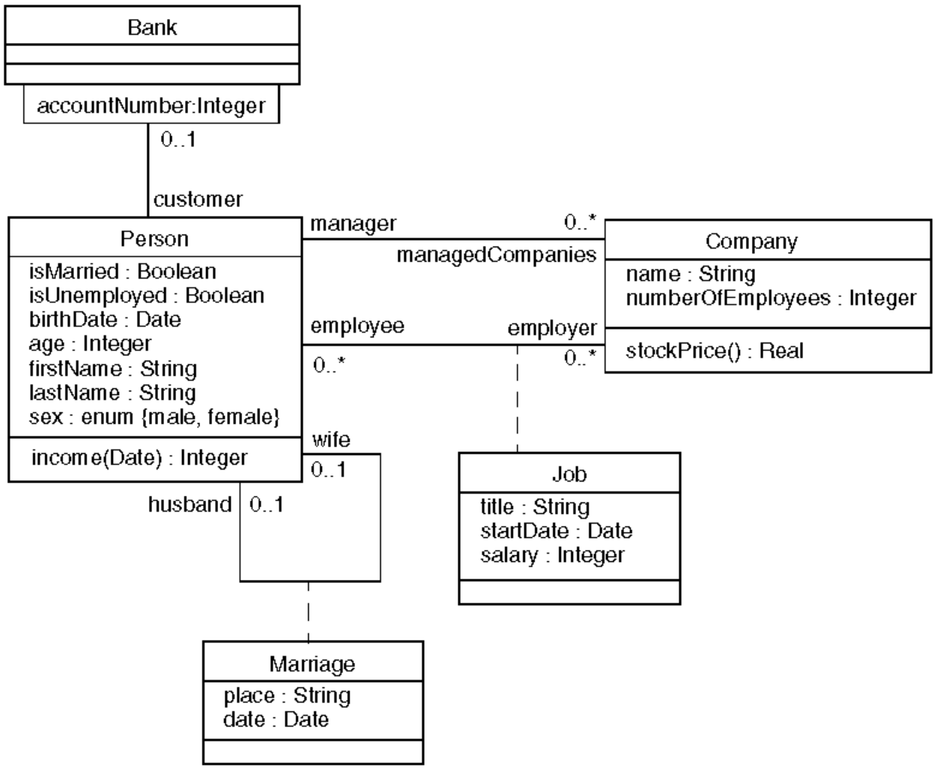

Figura 2.2: Un esempio di class diagram.

Un **oggetto** rappresenta una particolare istanza di una classe. La principale differenza è che esso ha un'identità e valori assegnati ai suoi attributi. Un oggetto è rappresentato da un rettangolo separato in due compartimenti.
Nel compartimento superiore appare il nome dell'oggetto nella forma "objectname : classname".
Nel secondo compartimento vengono posti gli attributi dell'oggetto con il loro valore tramite la sintassi "attributename : type = value".

Un'**interfaccia** è uno specificatore delle operazioni, spesso solo di una parte di esse, visibili esternamente di una classe, componente, o sistema che non specifica la struttura interna.
Le interfacce non sono implementate direttamente. Esse non hanno attributi, stati o associazioni; possiedono solo operazioni.Un'interfaccia può essere legata ad una o più classi tramite relazione di realizzazione o partecipare a relazioni di generalizzazione. Formalmente un'interfaccia è equivalente ad una classe

astratta senza attributi o metodi ma solo operazioni astratte.

Una **classe parametrica** (*Template*) è il descrittore di una classe con uno o più parametri formali non legati. Esso definisce una famiglia di classi, ogni classe difinita tramite il legame di un parametro formale ad un valore attuale. I parametri possono essere tipi come anche valori o operazioni. Gli attributi e le operazioni all'interno del template sono definiti in termini dei parametri formali cosìda risultare legati una volta che il parametro è legato.
Un template non è utilizzabile come classe fino a che tutti i suoi parametri non risultano legati. Quindi non può partecipare a relazioni intercorrenti tra classi fino a che non viene legato (può comunque essere sottoclasse di una classe ordinaria).
Tali parametrizzazioni possono applicarsi anche ad entità che non sono classificatori che seguiranno le stesse regole qui enunciate.

Le **associazioni** sono gli archi che legano i vari classificatori presenti in un class diagram. Esse sono rappresentate graficamente come linee che possono essere adornate per mostrare le loro proprietà. Un'associazione in sé non rappresenta altro che un legame (mono o bi-direzionale) tra i classificatori che unisce. Alle associazioni possono essere associate informazioni nei punti di aggancio ai classificatori (**association end**) come molteplicità, direzione ...

Alcune associazioni di particolare interesse sono

- **Aggregazione**: l'aggregazione rappresenta la proprietà di un'istanza della classe da parte di un altra. Essa viene rappresentata come un rombo sul terminale della classe contente.

- **Composizione**: è una forma di aggregazione forte che rappresenta un possesso forte e durata di vita coincidenti della parte con l'intero. Viene rappresentata come un rombo pieno.

- **Generalizzazione**: è la relazione tassonomica tra un elemento più generico e una sua specializzazione che sia pienamente consistente con il primo elemento e che aggiunga solo addizionali servizi e/o informazioni. La Generalizzazione è rappresentata come una relazione con un triangolo vuoto alla fine dell'arco verso l'elemento più generico.

2.3.3 Behavior diagrams

Questi diagrammi vengono usati per definire il comportamento (*behavior*) di una classe o sistema.

Statechart diagram

Uno **statechart diagram** può essere utilizzato per descrivere il comportamento di un elemento del modello, tipicamente una classe. Nello specifico viene descritta la possibile sequenza di stati e le azioni attraverso le quali l'elemento puo procedere durante la sua vita come risultato della reazione ad eventi discreti.
La semantica e la notazione utilizzati sono sostanzialmente quella di David Harel [Har87, HG96] con modifiche verso l'object orientation.

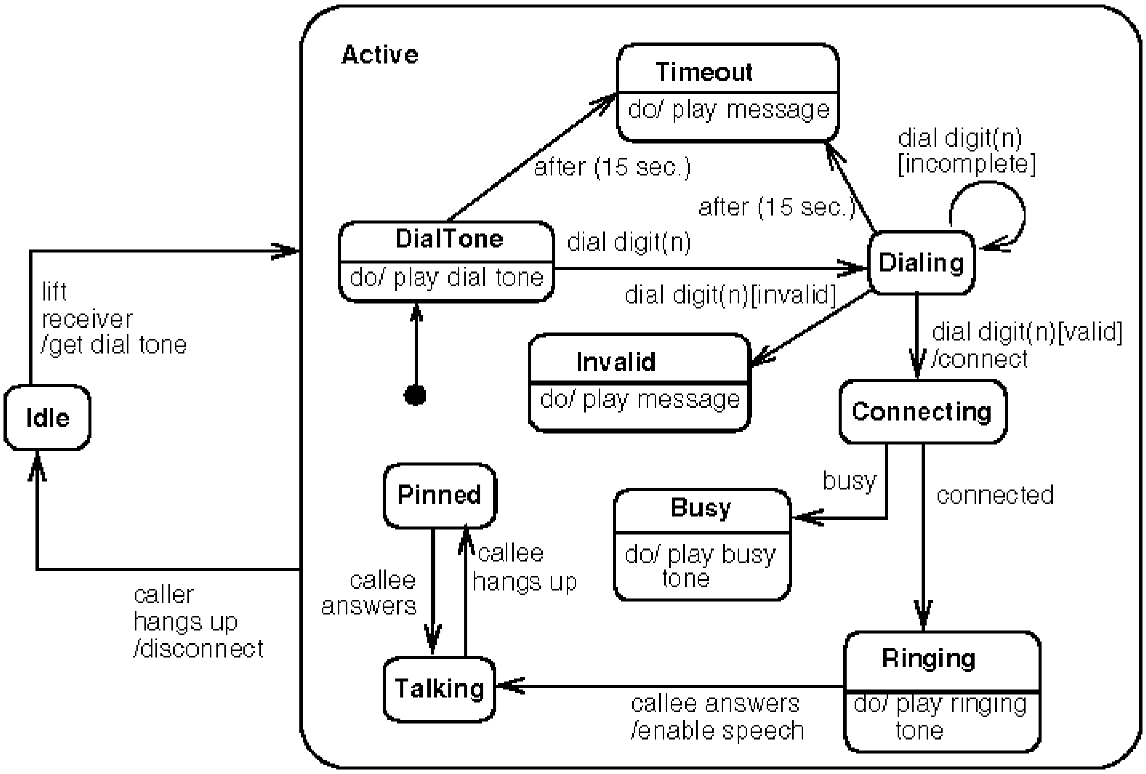

Figura 2.3: Un esempio di state chart diagram.

Uno statechart diagram contiene degli stati legati da delle transizioni.

Uno **stato**, rappresentato da un rettangolo con gli angoli arrotondati, è una condizione durante la vita di un oggetto, o di un interazione, durante la quale sono soddisfatte determinate condizioni, vengono effettuate delle azioni, o si attendono determinati eventi.

Un'**azione** puo essere associata ad uno stato o ad una transizione. Se associata ad uno stato può essere preceduta da un'etichetta che ne specifica o il tipo o il momento dell'esecuzione. Se associata ad un transizione verrà eseguita quando la transizione è attivata.
Un'azione può essere scritta in termini di operazioni, attributi o link [3] dell'oggetto che la possiede o ogni altra feature visibile nel suo scope. I dettagli dell'operazione dipendono al linguaggio selezionato per il modello.

Un **evento** è un accadimento degno di nota. In uno state diagram un evento è la causa che può attivare una transizione.
Gli eventi possono essere di diversi tipi (non necessariamente mutuamente esclusivi)
-Una condizione designata diventa vera: evento di tipo *change event*. L'evento avviene nel momento in cui la condizione passa da falsa a vera. Non è la guardia di una condizione (la guardia viene valutata dopo che la transizione può essere attivata da un evento, e se falsa la transizione non viene attivata).
-La ricezione di un segnale esplicito da un altro oggetto: evento di tipo *signal event*. Sul diagramma esso viene rappresentato tramite la segnatura dell'evento utilizzato come *trigger* dell'etichetta di una transizione di una transizione.
-La ricezione di una chiamata di metodo implementata come transizione da un oggetto (*call event*).
-Il passaggio di un determinato periodo di tempo dopo un particolare evento (e.g. un timeout dopo l'invio di un messaggio), spesso l'ingresso in un detrminato stato o l'occorrenza di una determinata data/ora (*time event*).

Una **transizioni Semplice** è una relazione tra due stati che indica che un oggetto nel primo stato entrerà nel secondo stato quando si verificherà un evento specificato sempre che certe condizioni siano sodddisfatte. Quando questo cambiamento di stato avviene si dice che la transizione “spara”. Il

[3] Un link è l'istanza di una relazione.

trigger per una transizione è l'occorrenza dell'evento che etichetta la transizione stessa. Se un evento non attiva nessuna transizione è scartato; se attiva più transizioni non concorrenti (e non vi sono livelli di priorità stabiliti o tali livelli sono uguali) una di queste è selezionata non deterministicamente ed attivata.

Activity diagram

Un **Activity Diagram** ha lo scopo di la focalizzare i flussi interni al programma e non gli eventi esterni. è quindi particolarmente adatto a modellare un flusso di controllo procedurale.

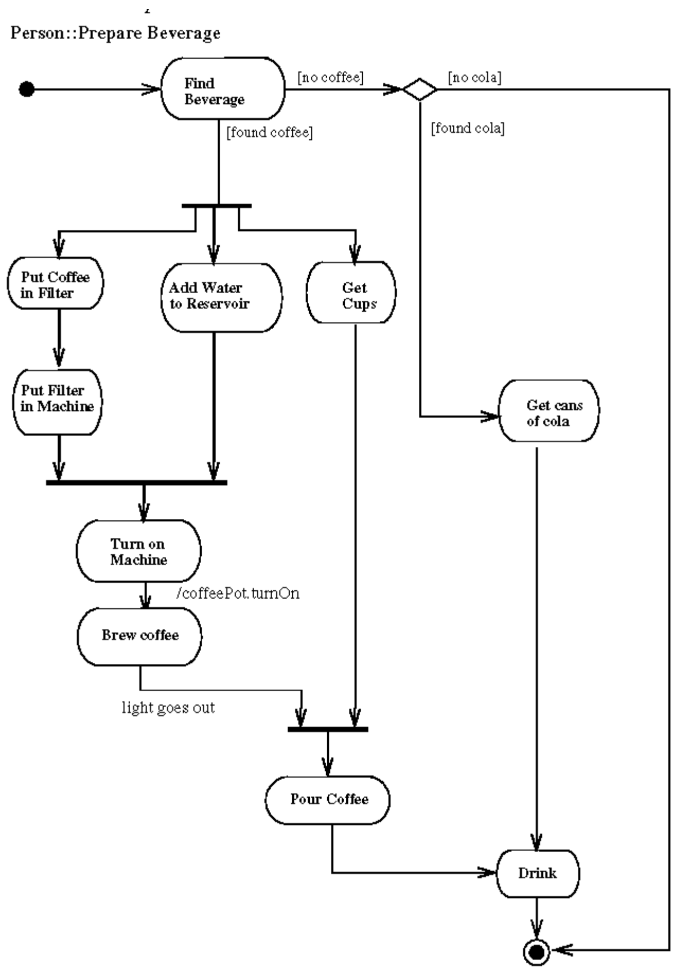

Figura 2.4: Un esempio di activity diagram.

Un **Activity graph** è una variazione di una macchina a stati in cui gli stati rappresentano l'esecuzione di un azione o sottoattività e le transizioni sono attivate solo al completamento dello stato. Essi rappresentano una macchina a stati della procedura stessa. Il grafo è composto da *action state* e transizioni.

Un **Action state** è un tipo di stato con un *entry action*, ossia un'azione da eseguirsi all'entrata dello lo stato, ed almeno una transizione uscente (o più di una se hanno una condizione di guardia) che viene attivata dal completamento dell'azione.
Un action state non può avere transizioni uscenti basati su eventi espliciti, ossia su eventi diversi dal completamento delle azioni previste all'interno dello stato.

Le transizioni possono essere tutte le transizioni utilizzate nelle state chart con l'unica eccezione che la loro attivazione è dovuta al completamento dell'attività dello stato di partenza.

2.3.4 Interaction diagrams

Un **interaction diagram** mostra una possibile struttura di interazioni tra istanze.
Gli interaction diagram hanno due possibili aspetti, entrambi basati sulle stesse informazioni di fondo, ma ognuno dei due pone l'enfasi su di un particolare aspetto.
I due aspetti che gli interaction diagram possono assumere sono: il sequence diagram e il collaboration diagram.
I sequence diagram enfatizzano la traccia temporale dell'interazione mentre i Collaboration diagram mettono in evidenza le relazioni tra le istanze.
In entrambi i grafici vengono definite delle collaborazioni.
I partecipanti ad esse definiscono dei ruoli che le istanze interpretano quando interagiscono le une con le altre.

Sequence diagram

Un **sequence diagram** mostra un'interazione, cioé un insieme di messaggi tra *ClassifierRoles* all'interno di una collaborazione per ottenere un determinato

risultato, ordinata in una sequenza temporale.
In particolare mostra le istanze che partecipano nell'interazione con le loro durata di vita e gli stimoli che vengono scambiati in una sequenza temporale. Data l'enfasi che si dà ai tempi sono particolarmente adatti per modellare sistemi real time.

Un sequence diagram ha due dimensioni:

1. la dimensione verticale rappresenta il tempo;

2. la dimensione orizzontale rappresenta i differenti oggetti.

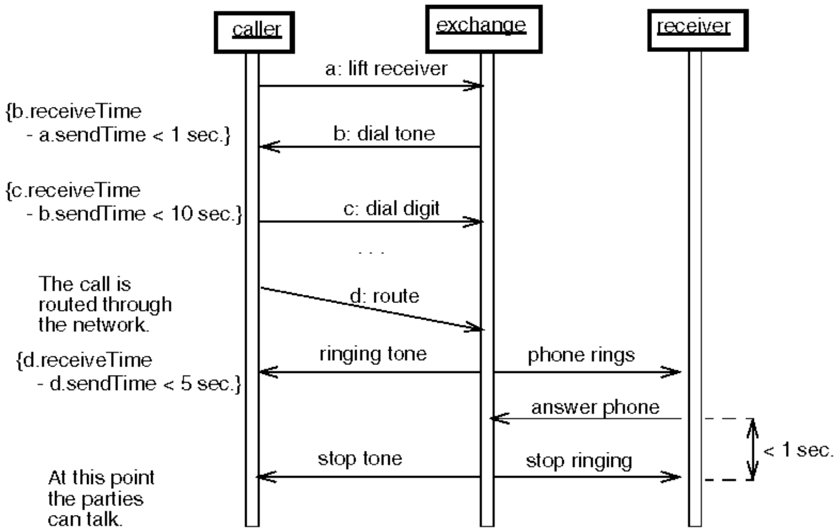

Figura 2.5: Un esempio di sequence diagram.

La durata di vita di un oggetto (**Object Lifeline**), denota che un oggetto recita un determinato ruolo per un determinato lasso di tempo. Delle frecce tra diverse lifeline denotano delle comunicazioni tra gli oggetti che assumono tali ruoli.
All'interno di un sequence diagram l'esistenza e la durata di un oggetto vengono denotati con una linea verticale che inizia con la creazione dell'oggetto e termina con la sua distruzione (se tali eventi avvengono durante l'interazione).

Un'**attivazione**, o focus di controllo, mostra il periodo durante il quale un oggetto sta eseguendo un'azione sia direttamente che tramite una chiamata ad

un oggetto subordinato.
Essa è rappresentata tramite un rettangolo sulla lifeline con una base allineata con l'attivazione e l'altra sulla terminazione dell'attivitá.

Messaggi e **stimoli**: uno stimolo è una comunicazione tra due oggetti che trasporta informazioni con l'aspettativa che un azione seguirá. Uno stimolo causerá l'invocazione di un'operazione, genererá un messaggio o causerá la creazione o la distruzione di un oggetto.
Un messaggio è la specifica di uno stimolo, quindi uno stimolo è un istanza di un messaggio.
Entrambi sono rappresentati da frecce che partono dalla lifeline di un oggetto e terminano nella lifeline dell'oggetto ricevente. Sulle frecce è apposta un'etichetta che ne specifica il nome ed i suoi argomenti (opzionalmente anche un numero di sequenza o una condizione di guardia).

Collaboration diagram

Un collaboration diagram mostra un'interazione organizzata attorno ai ruoli presenti nella stessa e le relazioni che intercorrono tra essi, ed è utile per comprendere tutti gli effetti di una singola istanza e per la progettazione procedurale. A differenza dei sequence diagram il tempo non è più mostrato in maniera esplicita, ma è dato implicitamente dal numero d'ordine dei messaggi.
Un collaboration diagram può essere realizzato a due differenti livelli:
A livello di specifica (mostrerà quindi ClassifierRoles, AssociationRoles e messaggi) o a livello di istanza (con oggetti, link, stimoli).

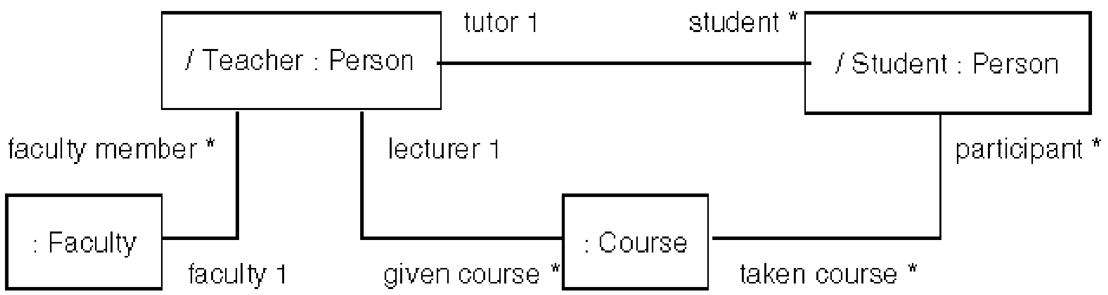

Figura 2.6: Un esempio di collaboration diagram.

Un comportamento può essere implementato tramite un insieme di oggetti che

si scambiano stimoli all'interno di un'interazione con lo scopo di ottenere un risultato. È spesso utile concentrarsi solo sugli oggetti, e le interazioni tra di loro, che collaborano per ottenere tale scopo.
Se una collaborazione è usata per descrivere un operazione o uno use case tale collaborazione viene detta realizzazione dell'operazione o use case.

2.3.5 Implementation diagrams

Gli implementation diagram mostrano aspetti dell'implementazione, inclusa la strutturazione del codice e l'implementazione a run time.
Essi sono di due tipi:

1. component diagram che mostrano la struttura del codice;

2. deployment diagram che mostrano la struttura a run time del sistema.

Component diagram

Un diagramma dei componenti *component diagram* mostra le dipendenze tra i componenti software, inclusi i componenti del codice sorgente. Nel caso di modellazione di un commercio il termine "componente software" prende il significato di "procedure commerciali e documenti".

Un diagramma delle componenti è un grafo di componenti connessi da relazioni di dipendenza.
I componenti possono essere connessi ad altri tramite "inscatolamento" per rappresentare delle relazioni di composizione.

Un **componente** (*component*) rappresenta un pezzo di implementazione distribuibile di un sistema come del codice (binario o sorgente) ma anche elementi commerciali in un sistema umano. Un componente è un classificatore e come tale può avere operazioni o realizzare interfacce.

Deployment diagram

I **deployment diagram** mostrano la configurazione a run time dei processi e delle componenti software.
Un **deployment diagram** è un grafo di nodi (*nodes*) connessi da associazioni

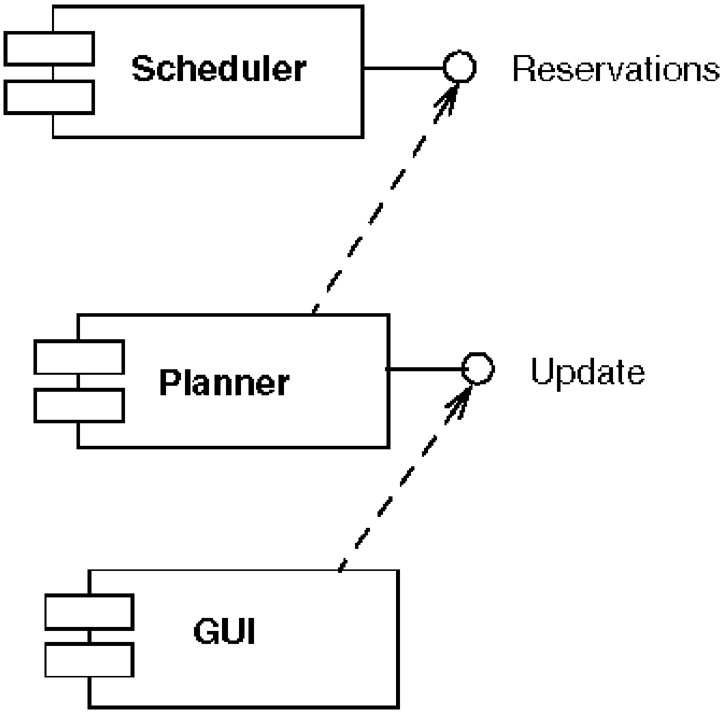

Figura 2.7: Un esempio di component diagram.

di comunicazione.

Un **nodo** è un oggetto fisico che rappresenta un risorsa di calcolo, su di esso possono risiedere sia oggetti che componenti. Se un nodo contiene componenti significa che esse "girano" su quel nodo. La migrazione di componenti da un nodo ad un altro può essere mostrata tramite lo stereotipo << *become* >> su una relazione di dipendenza.

2.4 Considerazioni finali

UML permette:

- La definizione di tipi di dato e di classi.
- Si possono definire classi parametriche ed interfacce.

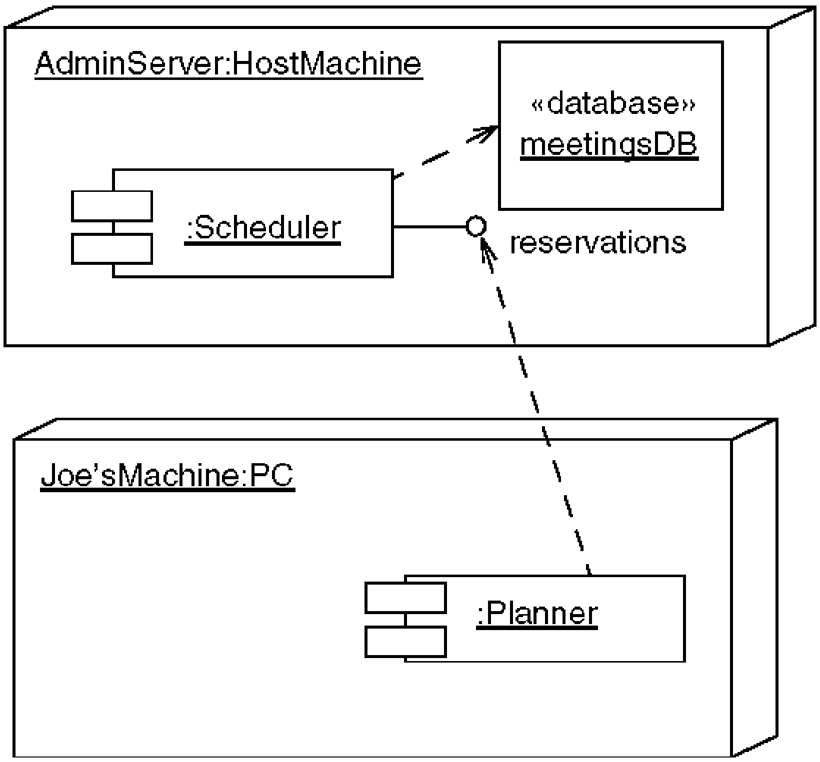

Figura 2.8: Un esempio di deployment diagram.

- Nel class diagram, oltre alla segnatura, vengono stabilite anche le relazioni tra le classi.Tra le relazioni vale la pena di citare: aggregation, inheritance, composition, template instantiation.
- La parte dinamica viene definita tramite gli state diagrams che utilizzano un modello esteso degli FSA (automi a stati finiti).
- Il meccanismo di comunicazione di UML è quello della chiamata di metodo: esso è sostanzialmente sincrono (almeno per quanto riguarda la ricezione). Tale limite è comunque superabile tramite il “solito” buffer.
- I canali di comunicazione sono rappresentati tramite associazioni nel class diagram, ed è possibile specificare “protocolli” di comunicazione tramite gli interaction diagram.
- Il modello di parallelismo non è esplicitamente definito, ma permette di definire sia multi-treading che multitasking vero.

- Il fatto che ad ogni classe è associabile un FSA rende la definizione di comportamenti pro-attivi possibile (basta definire un FSA pro-attivo) e non solo reattiva.

- Sono previsti costrutti specifici per la creazione e la distruzione dinamica di oggetti.

- È possibile allegare ai metodi commenti (con ad esempio la loro implementazione, gli eventuali limiti temporali, etc..) come note allegate. Esse tuttavia non hanno significato semantico in UML.

Capitolo 3

Utilizzo di UML per la specifica di un MAS

3.1 Utilizzo di un linguaggio di specifica ad oggetti per specificare un MAS

Il paradigma ad agenti può essere "implementato" tramite il paradigma ad oggetti, ma i due appartengono a livelli di astrazione diversi (esattamente come il paradigma ad oggetti può essere implementato in assembler ma il potere espresssivo ed il livello di astrazione di un linguaggio assembler ed uno ad oggetti sono però ben diversi).
Ed è proprio questa differenza di livelli di astrazione che rende utile il paradigma ad agenti: esso consente di astrarre da alcuni dettagli a basso livello per concentrarsi sulla descrizione di comportamenti complessi.
Un linguaggio di specifica ad oggetti fornisce meccanismi utili per la specifica di un MAS.

- Ogni agente è contraddistinto da un suo stato e da un suo comportamento; questo è modellato facilmente tramite la nozione di oggetto.

- Il concetto di identità degli oggetti fornisce un valido metodo per ottenere la nozione di identità di agente.

- Sia gli agenti che gli oggetti non rendono visibile all'esterno altro che una interfaccia mantenendo celate le informazioni circa i meccanismi di funzionamento. Per un oggetto l'interfaccia è obbligatoria ed è la segnatura formale delle operazioni (servizi) che l'oggetto mette a disposizione. Nel caso di un agente l'interfaccia non è obbligatoria (si pensi ad un agente puramente pro-attivo) e non è semplicemente una segnatura; essa può essere un complesso di comunicazioni (un protocollo di comunicazioni) o addirittura non essere ulteriormente specificata oltre ad un canale di comunicazione aperto verso l'esterno (ad esempio se per caso l'agente si occupa di riconoscimento di linguaggio naturale). Nonostante tutto questo avere a disposizione un concetto similare semplifica la modellazione.

- Il fatto che ogni oggetto abbia un proprio comportamento ed uno stato privato si adatta a modellare l'autonomia dell'agente, utilizzando i meccanismi di incapsulation ed information hiding, sia pure con le limitazioni discusse nel seguito.

- La nozione di ereditarietà è molto utile poiché permette di definire una tassonomia di classi di agenti. In un MAS spesso possono esistere agenti appartenenti a tipologie consimili: sia architetturalmente (BDI [1], proattivi, reattivi) sia come comportamento (agenti interfaccia verso programmi "standard", agenti che forniscono servizi di pagine gialle, ...). Il concetto di ereditarità rende modellabili questi concetti e fornisce una valida metodologia per il riutilizzo del codice.

Esistono tuttavia molte differenze tra i due paradigmi che li collocano a due livelli diversi

1. **Astrazione**. I due paradigmi si trovano a diversi livelli di astrazione: il paradigma ad agenti si concentra su entità autonome in grado di risolvere problemi semplici, che combinate tra di loro possono risolverne di più complessi; non si occupa di strutturazione del codice ma di come, e con *chi*, affrontare il problema all'origine.
   Il paradigma ad oggetti si concentra sulle entità che formano un sistema, sull'individuazione dei singoli componenti che formano il totale; l'attenzione è quindi sulla struttura del sistema risolutivo.

2. **Autonomia**. L'autonomia di un agente può essere molto maggiore di quella di un oggetto: un agente ha una propria agenda di cose da fare mentre un oggetto solitamente no. Tenendo conto che un agente può addirittura ignorare una richiesta ricevuta mentre un oggetto di

[1] BDI sta per Beliefs, Desire, Intention definito in [RG95] ed è una architettura classica per agenti.

cui venga chiamato un metodo non può non eseguirlo, si comprende immediatamente quanto sia maggiore l'autonomia di un agente.

3. **Coordinazione**. Gli agenti perseguono i propri scopi quindi, per poter arrivare ad un obbiettivo comune, è necessario stabilire dei protocolli di cooperazione, coordinazione e negoziazione; gli oggetti invece necessitano di un controllo esterno per eseguire i propri metodi ed è quindi già il sistema stesso che li coordina.

4. **Attività**. Gli oggetti sono generalmente passivi: reagiscono agli stimoli esterni ma non esibiscono comportamenti goal directed, laddove gli agenti agiscono per ragioni che emergono dalle proprie personali caratteristiche e dalle condizioni in cui l'agente si trova ad operare.

5. **Eterogeneitá**. Gli agenti sono normalmente eterogenei e necessitano quindi di un linguaggio comune di comunicazione. La comunicazione tra oggetti è molto più a basso livello. Si presuppone che un oggetto conosca la sintassi del metodo di un altro oggetto per poterlo invocare. Questo significa che la comunicazione non può prescindere dalla implementazione degli oggetti e che non esiste un linguaggio di comunicazione uniforme.

6. **Ruoli.** Il concetto di ruolo nel paradigma ad agenti è molto più forte che quello del paradigma ad oggetti, come espresso nella metodologia booch ed in UML [Boo83]. Peraltro un agente può assumere piú ruoli contemporaneamente e passare da un ruolo ad un altro dinamicamente.

7. **Messaggi.** Le nozioni di messaggi e di interfaccia sono profondamente diverse tra i due paradigmi. Da un punto di vista ad oggetti un agente distribuito può non esportare un'interfaccia e non rispondere ad alcun messaggio (nel senso di chiamata di metodo). Esso esplorerà il suo ambiente e, contemporaneamente, potrà ricevere messaggi asincroni (ove con il termine messaggi si intende una comunicazione "volontaria", molto più vicina ad un concetto umano, come un e-mail, che ad una chiamata di metodo) ed in base alla situazione contingente stabilirá il suo comportamento. Da un altro punto di vista, l'interfaccia di un agente si puo definire come i servizi che esso metterà a disposizione in seguito a delle richieste, ed il protocollo che è necessario seguire per effettuare queste richieste.

Queste considerazioni portano a dover realizzare delle estensioni dei linguaggi di specifica ad oggetti fino ad ora utilizzati, ed a elaborare un diverso approccio metodologico per la specifica di agenti.

3.2 Perchè UML

Tra i possibili linguaggi di specifica ad oggetti è stato scelto UML.
La scelta di UML deriva da diversi fattori

1. La principale ragione che ha motivato la scelta è che, attualmente, UML é *il* linguaggio di specifica ad oggetti, almeno in un ambito non accademico. Dopo l'adozione di UML da parte dell'OMG la sua diffusione, come linguaggio "standard" è ancora aumentata.

2. UML non ha una semantica definita in maniera formale, ciò rende più facile definire nuovi costrutti che si rendono necessari per supportare il paradigma ad agenti. In più nella stessa descrizione di UML viene suggerito di adottare nuovi *template* per estendere il linguaggio qualora sia necessario.

3. All'interno dei gruppi di ricerca sul paradigma ad agenti vi è molto interesse e lavoro per l'adozione di UML (o meglio di estensioni di UML) nella specifica di sistemi ad agenti: questo porterebbe innegabili vantaggi per quanto riguarda l'adozione del paradigma ad agenti da parte dell'industria. Utilizzare l'estensione di un linguaggio che già si conosce (e che è supportato) rende molto più facile adottarlo col minor numero di traumi possibili (basti pensare che la reale adozione del paradigma ad oggetti da parte dell'industria la si è avuta col C++). Tra i lavori interessanti in questo senso vale la pena citare

    - MaSE [MAS99]: MaSE è l'acronimo di Multiagent System Engineering ed è una proposta di estensione di UML fatta dall' Air Force Institute of Technology. In esso vengono proposti nuovi costrutti e diagrammi per estendere UML con lo scopo di modellare le Agent Class e le comunicazioni tra agenti.
    - Zeus [NNLC99]: Zeus è un ambiente di sviluppo grafico che per sistemi ad agenti con architettura BDI. In Zeus riveste particolare interesse il concetto di *ruolo* ed infatti vi è proposta un estensione di UML per modellarlo. Zeus, oltre a fornire l'ambiente di sviluppo,

contiene librerie utili per un sistema ad agenti programmato tramite linguaggio Java e classi di agenti predefinite. Una più ampia trattazione verrà fatta in 4.1.1.

- Il protocollo di interazione tra agenti proposto da James Odell [JO99]: In collaborazione con la FIPA Odell ha proposto un estensione di UML orientata agli agenti ed in particolare un diagramma di definizione dei protocolli di comunicazione tra agenti. Una versione semplificata di tale proposta viene utilizzata in 3.6.
- AUML (Agent UML) [AUM]: gruppo di lavoro che si propone lo sviluppo e l'utilizzo di una versione estesa di UML per la specifica di sistemi ad agenti. Del gruppo fà parte anche James Odell.

3.3 Mancanze di UML

Alcune carenze di UML che evidenziano la necessità di estendere il linguaggio sono:

- modellazione dei messaggi : come in ogni metodo di specifica ad oggetti lo scambio di messaggi è assimilato alla chiamata di metodo. Questo è inadatto per modellare lo scambio di messaggi in un MAS. In primo luogo la comunicazione in un MAS solitamente non è sincrona, inoltre l'eterogeneità degli agenti in un MAS costringe ad adottare sistemi di comunicazione più complessi della chiamata di un metodo (es KQML).

- modellazione dei protocolli di comunicazione : UML fornisce gli interaction diagram per stabilire il flusso delle comunicazioni tra 2 o più istanze di classe. Quello che a noi serve è modellare il protocollo di comunicazione tra 2 o più ruoli che possono essere ricoperti da agenti.

- modellazione dei ruoli : i ruoli in UML seguono la definizione di Booch, quindi lo stato ed il comportamento di un oggetto defiscono il ruolo che un oggetto può ricoprire. A sua volta un ruolo adempie le responsabilità dell'oggetto (con responsabilità si vuole indicare tutti i servizi che l'oggetto fornisce per tutti i "contratti" che esso supporta).
  A volte un oggetto può ricoprire più ruoli durante la sua esistenza: e i ruoli da lui ricoperti influiscono sulla semantica delle operazioni offerte. Comunque un oggetto non cambia i servizi offerti col cambiare dei ruoli, l'interfaccia che esso esporta rimane statica per tutta la durata della sua

esistenza. Un agente è un entità maggiormente dinamica. il mutare dei suoi ruoli muta anche i servizi offerti: un agente addetto alla compravendita di materiali può comprare e vendere contemporaneamente ma anche solo comprare (e rifiutarsi di vendere) o viceversa. L'assunzione di ruoli da parte di un agente modifica l'insieme dei servizi forniti.

- assegnazione dei ruoli a una specifica classe di agente : in UML non esiste un "diagramma dei ruoli" ed una progettazione basata sui ruoli non è quindi direttamente supportata. I ruoli sono altresìun utile astrazione per gli agenti (e per i protocolli che tali agenti utilizzeranno) quindi si sente la necessità di una tale modellazione.

Dalle precedenti considerazioni risulta quindi chiaro che, per modellare agenti o un sistema ad agenti, UML si rivela insufficiente. Per ovviare a questi problemi viene qui proposta un estensione di UML che tenga conto delle particolarità degli agenti, in particolare i ruoli, l'interazione tra agenti e ruoli e i protocolli di comunicazione.

3.4 Role Diagram

La prima estensione da noi realizzata che viene proposta in questa tesi si è in parte ispirata al metodo proposto in ZEUS per la specifica degli agenti, modificato per una sua integrazione in CaseLP (si veda capitolo 5).

il Role diagram mostra i ruoli che vengono ricoperti dagli agenti presenti in un MAS. I ruoli emergono solitamente nella fase di esame del dominio applicativo in cui il MAS dovrà operare. In particolare evidenzia i servizi richiesti ed offerti da un ruolo e le relazioni intercorrenti tra i ruoli stessi. Da un certo punto di vista esso presenta alcune somiglianze con il class diagram, dato che si occupa di relazioni tra strutture; la differenza fondamentale è che un ruolo è una struttura dinamica e non statica come una classe.

Un role diagram è un grafo di Ruoli, Actors ed Agent Class connessi dalle relazioni di contenimento, raffinamento o utilizzo intercorrenti tra essi.
La nozione O-O standard più vicina ad un ruolo è quella di interfaccia. Le differenze tra le due nozioni sono comunque abbastanza ampie da necessitare qualche commento: un'interfaccia rappresenta la segnatura di una serie di servizi messi a disposizione dalle classi che la realizzano.
Un ruolo rappresenta una serie di servizi non solo forniti ma anche richiesti dall'agente che lo ricopre ed il totale dei servizi offerti varia col variare dei ruoli assunti dall'agente.

3.4.1 Role

Un ruolo è un insieme di servizi (forniti e richiesti) che viene ricoperto da uno o più agenti nel corso della loro esistenza. I ruoli forniscono un vocabolario per descrivere un sistema ad agenti, con ogni ruolo usato per descrivere una posizione ed un insieme di responsabilità all'interno di un contesto. Tale approccio incoraggia a pensare al problema da risolvere nei termini di ruoli che devono essere assunti dai singoli elementi software che dovranno risolvere il problema in esame, in termini di responsabilità e capacità, in modo da portare ad una corretta modellazione secondo il paradigma ad agenti.Tale approccio rappresenta un estensione del role modelling dei sistemi O-O.

Per capire cosa puo essere un ruolo, e cosa non può esserlo, è utile uniformarsi a questi criteri:

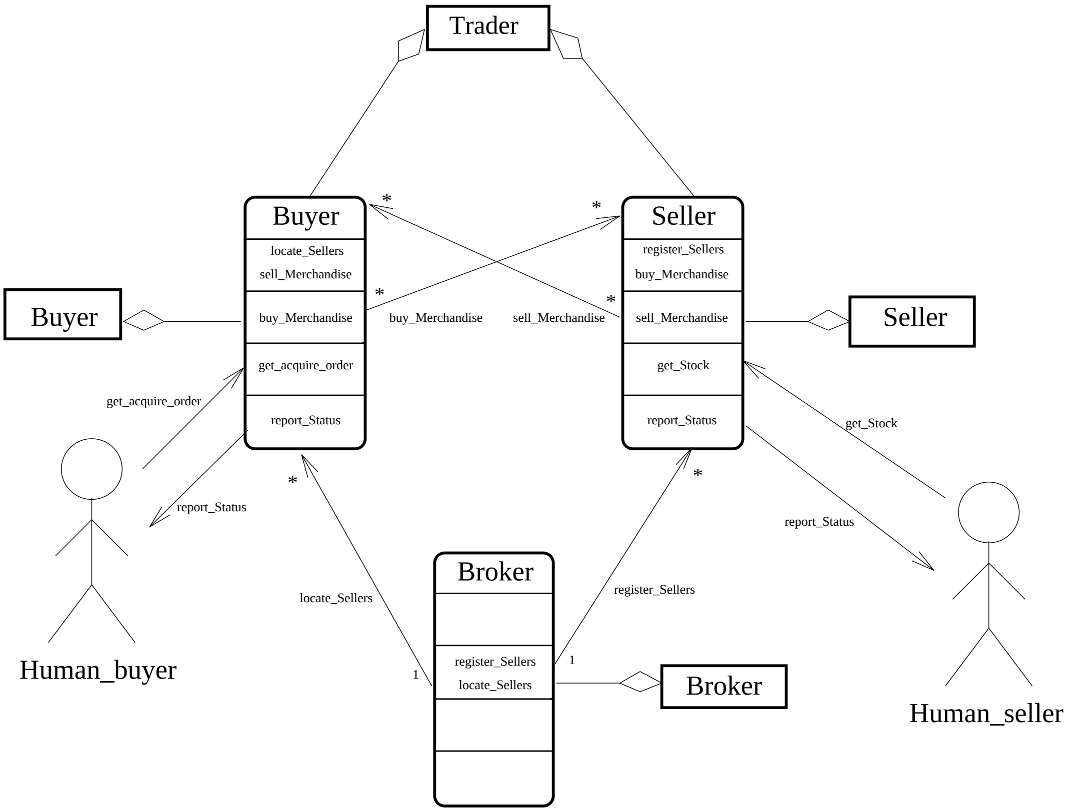

Figura 3.1: Un esempio di Role Diagram : un mercato distribuito.

- Modularità: un ruolo descrive un insieme di entità che si troveranno ad occupare la stessa posizione in una struttura ricorrente, perciò un ruolo deve essere sufficientemente modulare da permettere di aggiungere nuovi "interpreti" senza dover ridefinire il sistema di ruoli in cui è inserito.

- Alta coesione: per promuovere la modularità un ruolo deve possedere un insieme ben definito di responsabilità correlate ed avere una funzione ben definita; le responsabilità devono formare un'unità coesiva.

- Parsimonia: un ruolo non deve avere responsabilità estranee.

- Completezza: un ruolo non deve essere banale; è errato fare di ogni servizio un ruolo; ruoli troppo piccoli devono essere fusi tra loro per creare ruoli più grandi.

- Accopiamento lasco: le dipendenze tra i ruoli devono essere minimizzate.

La modellazione dei ruoli è rilevante in molti aspetti dello sviluppo di un sistema ad agenti, dal livello di analisi fino all'implementazione e manutenzione degli agenti. Una valida scelta dei ruoli aiuta anche nell'identificazione e specifica dei protocolli di comunicazione e coordinazione necessari al funzionamento del MAS stesso. Tale fase quindi dovrà essere eseguita con particolare attenzione. Tipicamente un agente ricoprirà molti ruoli nel corso della sua esistenza, cosicché la validità dell'agente stesso dipende dalla sua abilità nel supportare i servizi che deve offrire.

Figura 3.2: Un esempio di Role.

Semantica I servizi definiscono funzionalità che l'agente fornisce o richiede per raggiungere il suo scopo. I servizi tipicamente dipendono dal dominio applicativo del sistema. Alcuni servizi forniti possono essere esportati ad utenti esterni o ad altri MAS. Un agente che ricopre un ruolo deve essere in grado di fornire (magari in momenti diversi della sua esistenza) tutti i servizi offerti da tale ruolo.

Come si vede dalla figura 3.2 nella modellazione dei ruoli qui proposta un ruolo viene visualizzato tramite un rettangolo dagli angoli arrotondati che contiene 5 compartimenti.Essi sono

- **Name Compartment**: In tale compartimento è riportato il nome del ruolo e, opzionalmente, una breve descrizione dello stesso.
- **Require Compartment**: In questo compartimento sono elencati i servizi che un ruolo richiede da altri ruoli ricoperti da agenti presenti nel MAS.

- **Provide Compartment**: In questo compartimento sono elencati i servizi che l'agente che ricopre tale ruolo fornisce ad altri agenti presenti nel MAS.

- **Import Compartment**: In questo compartimento sono elencati i servizi che un ruolo richiede da entità esterne al MAS, come un operatore umano o altri MAS presenti nell'ambiente in cui il sistema verrà inserito.

- **Export Compartment**: In questo compartimento sono elencati i servizi che un ruolo offre ad entità esterne al MAS, come, ad esempio messaggi o risultati per un operatore umano o ad altri MAS presenti nell'ambiente in cui il sistema verrà inserito.

I **servizi** sono contenuti nei compartimenti sopra definiti. Un servizio è indicato tramite un nome assegnatogli. Opzionalmente è possibile aggiungere una breve descrizione del servizio.

3.4.2 Agent Class

Una agent class viene vista come un contenitore di ruoli. Essa sarà connessa mediante le relazioni di Contains ai ruoli che assumerà. In essa è presente un singolo compartimento che riporta il nome della classe.
Secondo l'approccio proposto in Zeus, dei principi validi per il passaggio da ruoli ad agenti, ossia su come accorpare i ruoli in classi d'agenti, possono essere i seguenti:

1. autonomia e

2. socialità.

Criterio di autonomia Dato che una delle caratteristiche contraddistintive di un agente è quella dell'autonomia; il partizionamento dei ruoli deve avvenire in modo tale da massimizzare l'autonomia degli agenti. Un agente deve essere autonomo e quindi deve essere responsabile per il controllo delle risorse e la realizzazione dei servizi. Tale area di controllo (ossia ogni gruppo di servizi, e le risorse necessarie per fornirle, messi a disposizione da un agente) viene definito sfera di responsabilità. Quando lo sviluppatore deve decidere quali agenti esisteranno all'interno del MAS bisognerà considerare come le responsabilità saranno partizionate all'interno del dominio di applicazione cioé la quantità di legami tra i ruoli esistente. Se, per esempio due ruoli saranno correlati dal fatto, per esempio, di essere sempre presenti sulla stessa macchina o di dover accedere agli stessi dati, essi dovranno essere uniti in un unico agente. In breve: ogni sfera di responsabilità dovrà possedere un unico agente. È chiaro che questo approccio non è univoco e non esisteranno scelte "giuste" e "sbagliate" ma modi diversi di affrontare il problema.

Un altro parametro importante è la percezione dell'ambiente esterno: conviene che ad un singolo ambiente sia assegnato un singolo agente.
Esempio: se si deve gestire la temperatura in più stanze diverse conviene che ad ogni stanza sia assegnato un singolo agente.

Criterio di socialità Gli agenti devono essere sociali. Questo significa che gli agenti presenti in un MAS devono interagire costruttivamente ed intelligentemente con le altre entità presenti nel MAS. Questo implica che le informazioni e i servizi possono non essere direttamente accessibili (per motivi di sicurezza o di efficienza) ma devono essere richieste all'agente responsabile del loro controllo. Tramite questo criterio si separano le risorse dell'applicazione dalle entità che le devono utilizzare per fornire servizi. Per esempio una società non desidererà certo mettere a disposizione il suo database ma utilizzarà un agente interfaccia che filtrerà le richieste ed evaderà solo quelle leggittime. Concettualmente gli agenti hanno effetto sulle risorse, le interazioni hanno effetto sugli agenti; quindi ogni punto di accesso ad informazioni, conoscenze e servizi è un buon candidato ad essere un agente.

Questi due criteri possono essere frequentemente in conflitto tra loro. Sta alle scelte progettuali trovare un buon compromesso tra queste necessità.

3.4.3 Relations

Gli elementi del Role Diagram possono essere collegati tramite delle relazioni che costituiscono gli archi del grafo. Esse possono essere

- **Contains Relation**: Questa relazione, se lega due o più ruoli, indica che un ruolo contiene un altro ruolo al suo interno e che in un qualche momento della sua esistenza richiederà e fornirà i servizi del ruolo contenuto. Se invece lega un ruolo ad una Agent Class sta a significare che la classe assumerà quel ruolo nel corso della sua esistenza. La contains relation è simboleggiata tramite la notazione standard di UML cioé una linea con un diamante vuoto ad una delle due estremità; l'estremità col diamante è collegata all'agent class o ruolo che contiene l'altro.

- **Generalization Relation**: Questa relazione indica che un ruolo è il raffinamento di un altro ruolo, in un senso molto simile alla relazione di sottoclasse. La nozione é diversa dalla nozione di contenimento da un punto di vista concettuale dato che questa rappresenta una specializzazione del ruolo. La rappresentazione è quella standard di UML cioé una freccia vuota.

- **fulfill relation**: Questa relazione indica che il ruolo A soddisfa una richiesta del ruolo B. È indicato tramite una freccia con la parola chiave << fulfill >> che parte da A ed arriva a B. Se ritenuto utile si possono inserire degli attori, che rappresentano entità esterne al MAS, e collegarli ai servizi esportati e importati tramite le relazioni di soddisfacimento.

3.5 Agent Diagram

Il secondo diagramma che è stato sviluppato per questa tesi si occupa di una definizione architetturale del sistema. Nel caso di un MAS l'architettura assume un'importanza ancora maggiore rispetto a quella di un sistema sviluppato con altre tecniche ed è quindi sembrato interessante mettere in rilievo quest'aspetto.

Un agent diagram specifica il tipo, l'archittettura, l'eventuale interprete di un agente, come i servizi che caratterizzano i ruoli ricoperti da un agente vengono realizzati e le relazioni che gli agenti all'interno di un MAS hanno tra di loro. Lo scopo è comunque ancora una descrizione ad alto livello delle classi di agenti comprese in un MAS.

Le Agent Class verranno poi ulteriormente specificate mediante i normali diagrammi di UML come si vedrà in seguito.

3.5.1 Agent Architecture

Una agent architecture rappresenta il descrittore di una architettura per agenti. Esempi di tali architetture possono essere BDI, reattiva, pro-attiva... Una architettura si deve considerare come una metaclasse di agente ossia l'istanziazione di una architettura è una agent class.
Una agent architecture, come ogni altro classificatore, può essere specificata mediante raffinamento. In relazione al lavoro svolto su CaseLP una generalizzazione di particolare interesse è quella che porta da un'architettura astratta ad un'architettura concreta o tra piú architetture astratte, ulteriormente specializzate ad ogni passo. Con architettura astratta si intende una architettura in cui i tipi siano parametrici ed almeno una parte delle sue funzioni siano astratte. La relazione di generalizzazione che lega quindi un architettura astratta ad una concreta è quindi similare a quella che lega una classa astratta ad una classe concreta con in più la nozione di istanziazione dei parametri. Un'architettura viene rappresentata tramite un rettangolo che contiene:

- Name Compartment: ove si indica il nome dell'architettura che viene definita.

- Architecture Kind Compartment: viene indicato il grado di astrazione dell'architettura. I due livelli attualmente utilizzati sono *abstract* e *concrete*.

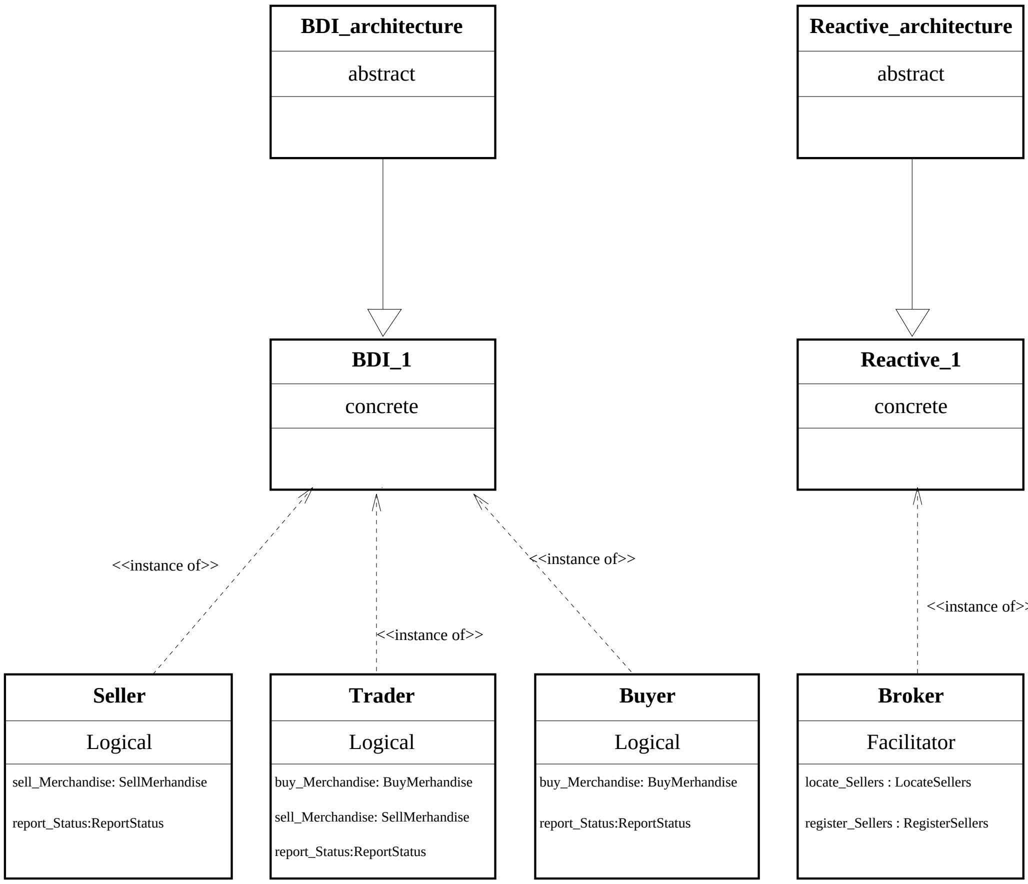

Figura 3.3: Un esempio di Agent Diagram. Da notare che non sono presenti istanze di agenti

- Operation Compartment: opzionalmente possono essere elencate le operazioni definite all'interno dell'architettura, ad esempio *percept_event()*.

3.5.2 Agent Class Architecture

Una agent class rappresenta un descrittore per un insieme di agenti aventi caratteristiche comuni. Il concetto è molto simile a quello di una classe di UML all'interno del Class Diagram ma, in questo caso, non è visualizzata la struttura interna della Agent Class ma una sua visione architetturale. Una Agent Class Architecture è definita come un rettangolo che contiene all'interno

- **Name Compartment** In tale compartimento è riportato il nome della Agent Class e, opzionalmente, una breve descrizione della stessa.

- **Kind Compartment** Specifica il tipo generico della classe presa in esame: esso è opzionale ed usato in seguito per uniformità col metodo di progettazione di CaseLP (ed in particolare con MAS-Adl).

- **Service Realization Compartment** Esplicita da che operazioni della classe che implementa la Agent Class vengono realizzati i servizi forniti ed esportati dai ruoli che vengono ricoperti dall'agent class. Essi verranno denotati con la sintassi
  Service : MethodName
  Questo compartimento non è obbligatorio. Tale rappresentazione è utile per stabilire un legame tra la specifica più ad alto livello (ruoli ed architetture) e la parte a più basso livello.

Nel caso in cui l'architettura sia parametrica, i parametri saranno eventualmente evidenziati tramite la normale sintassi UML.

3.5.3 Agent

È possibile inserire l'instanziazione di una Agent Class, cioé un agente vero e proprio, all'interno del grafico per stabilire links o dipendenze che si devono estendere a livello di singolo agente. Un agente viene disegnato come un rettangolo con un singolo compartimento. Il compartimento mostra il nome dell'oggetto e la classe di appartenenza, il tutto sottolineato, usando la sintassi:
objectname : classname
Dato che si è ancora ad un livello di architettura nulla viene detto dello stato specifico del singolo agente, come invece accade nei diagrammi "standard" di UML.

3.5.4 Relations

Le relazioni che si sono volute introdurre in questo diagramma sono le seguenti

- **Contains Relation**: La relazione di contenimento viene utilizzata per esprimere che alcune classi che risultano particolarmente importanti sono contenute all'interno dell'Agent class. Un esempio può essere, nel caso di un agente interfaccia, la classe che realizza l'interprete verso un programma esterno.

- **Generalization Relation**: Esprime la relazione di ereditarietà (singola) esistente tra le Agent Class; essa viene usata per esprimere generalizzazione di agent class effettive o di agent architecture. Nel caso delle agent class la relazione di generalizzazione è più concettuale che pratica: dato che un agente non esporta un'interfaccia in senso O-O è più da considerarsi come un arricchimento di funzionalità fornite che altro. Nel caso invece delle Agent Architecture essa rappresenta una una normale relazione di generalizzazione.

- **Fulfill Relation**: Nel caso che si renda necessario specificare (o mettere in risalto) particolari relazioni di soddisfacimento di servizi tra agent class, si può usare un indicatore di Fulfill Relation che unisce le Agent Class presenti nel diagramma.

- **Instance of Relation**: Indica che un classificatore è l'istanziazione di una classe o meta-classe. Viene utilizzato per unire le architetture alle Agent Class o gli agenti alle agent class. Viene indicata tramite una freccia tratteggiata con la punta sulla classe e la coda sull'istanza; sulla freccia si troverà la parola chiave <<instance of>>.

3.6 Protocol Diagram

Tale diagramma è fortemente mutuato dalla proposta di Odell [JO99] per la FIPA, sia pure con alcune semplificazioni. I protocolli (o conversazioni) vengono mappati in un diagramma strutturalmente simile ad un interaction diagram (anche la semantica è similare). L'eccezione più rilevante è la presenza non di classificatori qualunque ma dei ruoli: un ruolo, ribadiamolo una volta di più, è qualcosa di dinamico ed una classe può implementare più ruoli sia contemporaneamente che sequenzialmente durante la sua esistenza. Una Conversazione (o protocollo di comunicazione) specifica un possibile scambio di messaggi tra due o più ruoli od agenti, all'interno di una collaborazione per ottenere un comportamento desiderato. Un protocol diagram ha la forma di un sequence diagram: cioé un diagramma con due dimensioni:
1) la verticale rappresenta il tempo
2) l'orizzontale rappresenta i diversi ruoli che partecipano alla comunicazione.

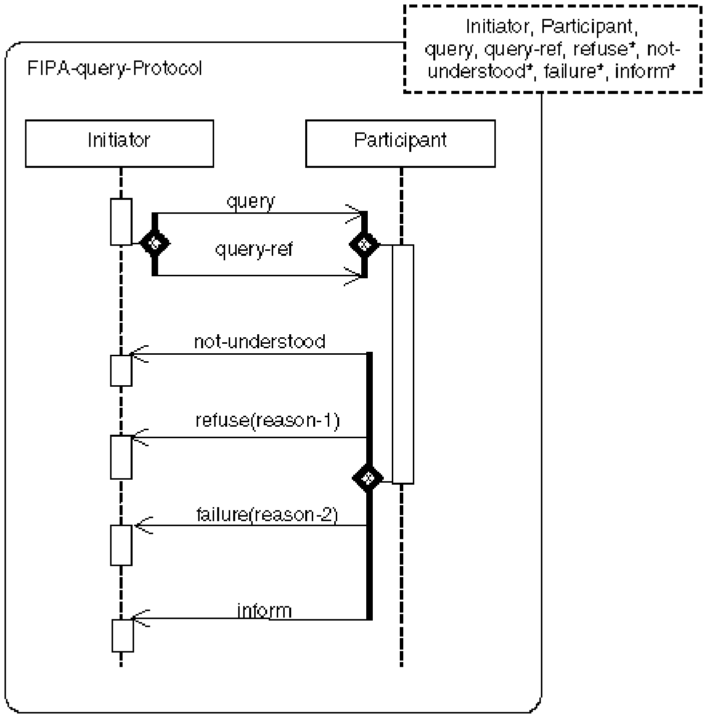

Figura 3.4: Un esempio di Protocol Diagram.

3.6.1 Ruoli

In un protocol diagram i Ruoli sono ruoli che saranno assunti dagli agenti che partecipano all'interazione. Possono essere anche variabili di Ruolo che verranno istanziati con ruoli reali al momento della realizzazione del protocollo di comunicazione stesso. Un ruolo, in questo caso può essere visto come un insieme di agenti che soddisfano una determinata interfaccia, descrizione di servizio o comportamento. Quindi l'implementazione di un agente può soddisfare multipli ruoli.

3.6.2 Role lifeline

A partire da ogni ruolo si traccia una barra verticale che rappresenta la durata in cui un agente assume un determinato ruolo all'interno dell'Interazione. Per esempio un agente "User Interface" viene creato quando un utente si collega al sistema ed è distrutto quando l'utente si scollega. Una lifeline può prendere "strade" diverse in seguito ad eventi o comunicazioni ricevute dall'agente. Tali cammini alternativi possono essere suddivisi tramite:

- Parallelismo in AND. Rappresentata come una barra orizzontale spessa.
- Parallelismo in OR (or inclusivo). Rappresentata come una barra orizzontale spessa con un rombo di decisione vuoto.
- Decisione (or esclusivo). Rappresentata come una barra orizzontale spessa con un rombo di decisione con all'interno una "X".

3.6.3 Thread d'interazione

Similare all'activation (o focus di controllo) degli interaction diagram di UML il thread d'interazione viene necessariamente raffinato in questo diagramma. Dato che il comportamento associato ad un ruolo dipende dai messaggi ricevuti e che possono verificarsi diverse possibilità di risposta in seguito ad un messaggio ricevuto, il thread d'interazione cioé l'attività del ruolo in risposta ai messaggi ricevuti deve poter essere suddiviso in modo rappresentare azioni differenti. Esso viene mostrato come un rettangolo con la cima allineata con l'inizio dell'operazione e il fondo con la terminazione dell'attività posto sulla lifeline del ruolo. Opzionalmente può essere adornato con l'indicazione dell'attività svolta.

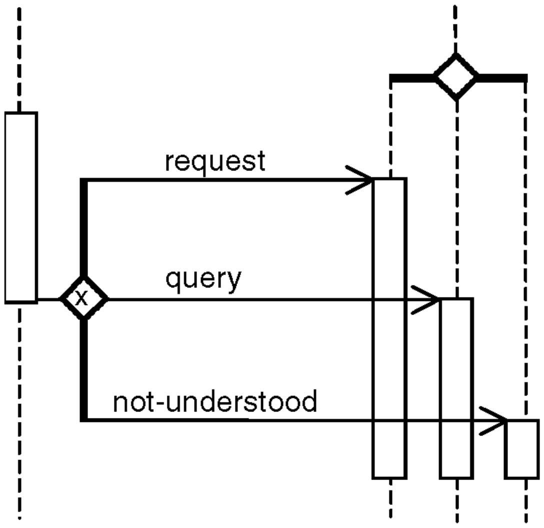

Figura 3.5: Un esempio di lifeline suddivisa con thread d'interazione su di essa.

Il thread d'interazione mostra il periodo durante il quale il ruolo sta effettuando qualche azione come reazione ad un messaggio ricevuto. Esso rappresenta solo la durata dell'azione nel tempo, ma non la relazione di controllo tra il mittente ed il ricevente del messaggio. Un thread d'interazione è sempre associato con la lifeline di un ruolo.

3.6.4 Messaggi

La ragione di esistere di un protocollo è la definizione di una schema di comunicazione ed in special modo regolare l'invio di messaggi tra un ruolo ed un altro. Tale invio può essere svolto in modi differenti a seconda di alcuni vincoli o decisioni da parte degli agenti che ricoprono i ruoli attualmente in comunicazione. Un messaggio è una comunicazione da un ruolo ad un altro che trasporta con sé informazioni con l'aspettativa che il ruolo ricevente reagirà in accordo con la semantica del messaggio. La specifica del protocollo nulla dice circa l'implementazione del messaggio stesso.

I messaggi possono essere suddivisi in due classi: messaggi semplici e messaggi complessi.

- Messaggi semplici. L'invio di un messaggio è mostrato come una freccia che parte dal thread d'interazione di un ruolo (il ruolo che invia il messaggio) e termina sul thread d'interazione di un altro ruolo (il ricevente).

Ogni freccia è etichettata con un etichetta di messaggio.
L'etichetta di messaggio ha la seguente sintassi:

predecessor guard-condition sequence-expression communicative-act argument-list

predecessor. Il predecessor è costituito da un singolo numero naturale seguito da (/) e definisce la sequenza in un costrutto parallelo, cioé serve per differenziarne i diversi rami. Questa clausola è omessa se vuota.

condizione di guardia. La condizione di guardia è una espressione a valori booleani come usuale in UML. La sua semantica è che il messaggio viene inviato se e solo se la guardia è vera.

espressione di sequenza. L'espressione di sequenza rappresenta un vincolo sul numero di volte o sui vincoli temporali legati all'invio di un messaggio. In particolare n..m denota che un messaggio viene inviato da n a m volte con $n \in N$, $m \in N \cup \{*\}$, lo star rappresenta un numero arbitrario di volte. Può essere indicato un limite temporale posto sul tempo di ricezione.

atti di comunicazione. Il communicative-act è una stringa riportante il nome dell'atto di comunicazione o il nome dell'istanza dell'atto di comunicazione ed il tipo dello stesso espresso nella forma
nome:communicative-act.

lista di parametri. È una lista di parametri separati da virgole racchiusi tra parentesi tonde. Le parentesi possono essere omesse se la lista è vuota.

Nel caso di un messaggio che il ruolo invia a se stesso, la freccia può partire e terminare sulla stessa lifeline o thread. Tale tipo di thread annidato è denotato dal fatto di essere leggermente spostato a destra rispetto al thread attuale.

- Messaggi complessi. I messaggi complessi sono messaggi composti da più messaggi uniti tramite un operatore logico. Esistono tre tipi di messaggi complessi. Tutti i tipi sono rappresentati tramite una freccia che parte dal ruolo che invia il messaggio e termina in
  - una spessa linea verticale (per il parallelismo in AND),

– una spessa linea verticale con un rombo vuoto (per il parallelismo in OR; or inclusivo) oppure
– una spessa linea verticale con un rombo vuoto con una "X" all'interno (per le decisioni; or esclusivo)

Da queste barre verticali nuove frecce si dipartono ad angolo retto e terminano sui possibili thread d'interazione dei ruoli ricettori. L'unione di differenti messaggi è indicato in modo analogo, cioé i messaggi arrivano ad una barra e si uniscono in una singola freccia.

3.6.5 Protocolli parametrici

I protocolli definiti possono essere parametrizzati. Un protocollo parametrico è la descrizione di un protocollo con uno o più parametri non istanziati. La sintassi è esattamente quella di UML. Tipicamente i parametri non istanziati sono ruoli, vincoli, o le istanze di atti di comunicazione. Un protocollo parametrico non è utilizzabile fino a che non viene istanziato.

3.7 Collegamento tra i nuovi diagrammi e UML standard

Quella fino ad ora fornita è la specifica architetturale degli agenti e delle loro classi ed è stata fornita la specifica delle comunicazioni intercorrenti nel MAS. Per completare la definizione è necessario:

- Stabilire, mediante classi UML, l'ontologia del MAS. Ogni singolo concetto verrà mappato in un una classe, le quali, a loro volta, verranno racchiuse in un package che rappresenterà l'ontologia definita.
- Fornire la struttura statica delle architetture e delle loro istanziazioni in classi di agenti mediante i class diagram di UML.
- definire tramite i behaviour diagram il funzionamento delle operazioni introdotte.
- nel caso si reputi importante ai fini della specifica, fornire mediante gli implementation diagram, le informazioni sulla localizzazione dei processi e sulle caratteristiche hardware del MAS .

Parte II

Ambienti di prototipazione rapida per MAS

Capitolo 4

Alcuni ambienti di prototipazione rapida per MAS

In questo capitolo viene presentata una breve panaromica di alcune proposte esistenti per ambienti di prototipazione rapida/sviluppo/simulazione di MAS. Sono stati presi in esame sia strumenti sviluppati in ambito accademico che commerciali, a conferma dell'enorme interesse che esiste su questo argomento.

Con questa sintetica rassegna si vuole introdurre il lettore al mondo degli ambienti per modellazione e sviluppo di agenti, per consentirgli di comprendere le problematiche affrontate solitamente da uno strumento del genere, le soluzioni proposte e le diverse scelte architetturali affrontate. Questo capitolo vuole quindi fornire le basi per la comprensione di CaseLP descritto in 5, tramite la presentazione di sistemi analoghi e la discussione degli approcci seguiti da tali sistemi.

4.1 Ambienti di sviluppo e prototipazione di sistemi ad agenti

4.1.1 Zeus

Zeus ([zeu]) è un ambiente di sviluppo per agenti collaborativi sviluppato dal British Telecommunications Labs. Esso comprende una libreria di componenti scritta in Java ed una serie di strumenti per la definizione degli agenti.
Ogni agente in Zeus consiste in un livello di definizione, un livello organizzativo ed un livello di coordinazione.
Il livello di definizione rappresenta le capacità di apprendimento e di ragiona-

mento, i suoi scopi, le risorse disponibili, le abilità, le credenze, etc.
Il livello organizzativo descrive le relazioni di un agente con gli altri agenti presenti nel MAS.
il livello di coordinazione descrive le tecniche di coordinazione e negoziazione che l'agente possiede. I protocolli di comunicazione sono costruiti a partire dal livello di coordinazione ed implementano le comunicazioni inter-agente.
Lo sviluppo di un MAS con Zeus segue le fasi illustrate in figura 4.1. Gli *Activity Stage* sono le fasi di studio e codifica del sistema; i *Results* rappresentano i risultati ottenuti dalle attività precedenti

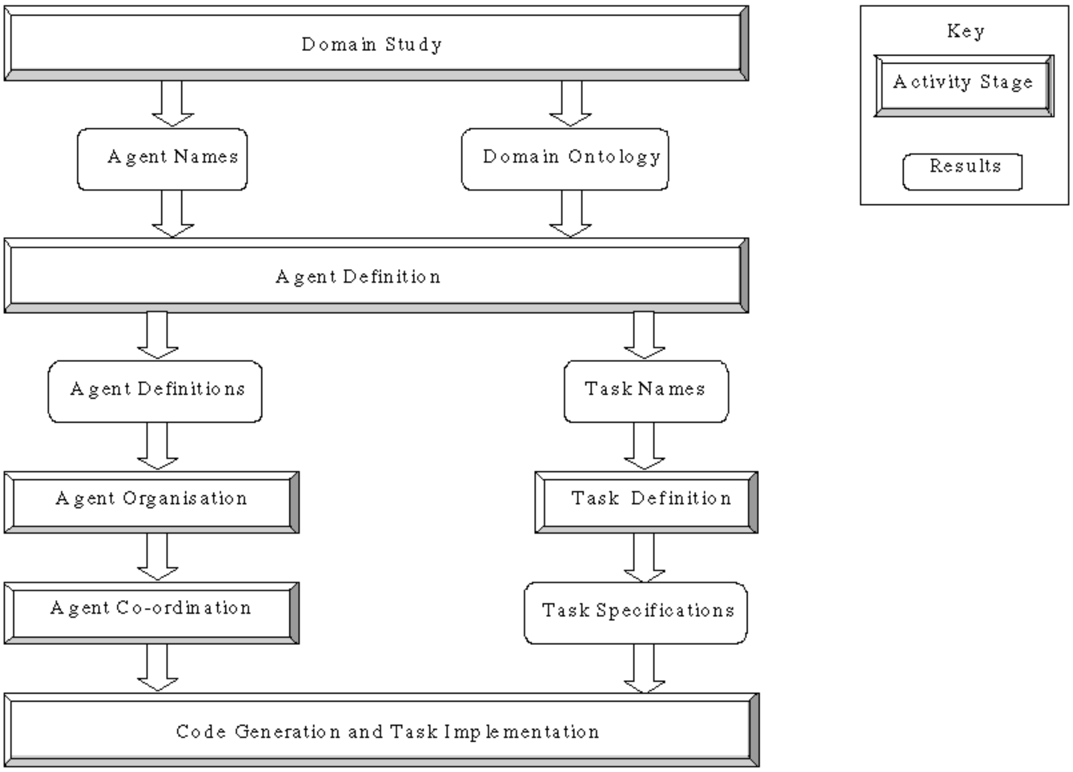

Figura 4.1: La procedura di sviluppo di Zeus.

Descriviamo le fasi dello sviluppo.

- **Studio del dominio** (*Domain Study*): In questa fase iniziale, lo sviluppatore analizza il dominio applicativo per identificare i potenziali agenti e per creare un'ontologia per i concetti del dominio.

- **Definizione degli agenti** (*Agent Definition*): Una volta che tutti i possibili candidati ad essere agenti sono stati identificati, lo sviluppatore deve identificare gli attributi significativi di ogni agente. Questa fase puo essere suddivisa nei seguenti passi

  1. identificare i compiti che l'agente può svolgere.
  2. identificare il numero di compiti che l'agente può svolgere concorrentemente.
  3. identificare con quanto anticipo l'agente può pianificare le sue attività. Questo rende l'agente più o meno reattivo a seconda dei tempi di pianificazione, anche se una pianificazione rapida può portare a risultati inferiori.
  4. identificare le risorse iniziali disponibili all'agente.

  La fase di definizione dell'agente continua fino a che tutti gli agenti sono stati considerati e i loro compiti e le loro risorse non sono state identificate.

- **Definizione dei compiti** (*Task Definition*): In questa fase i compiti che sono stati identificati precedentemente vengono definiti in termini di precondizione, effetti, costo, durata e vincoli.

  – Precondizione: è l'elenco delle risorse necesserie per svolgere il compito.
  – Effetti: sono le risorse che verranno prodotte dallo svolgimento del compito.
  – Costo e durata: rappresentano il costo del lavoro svolto ed il tempo impiegato per svolgere il compito.
  – Vincoli: sono condizioni (od effetti) addizionali che devono venire rispettate per poter svolgere il compito.

- **Organizzazione degli agenti** (*Agent Organisation*): Vengono definiti i rapporti tra gli agenti del MAS. Tali rapporti possono essere definiti come tra pari, stabilire una gerarchia di comando, etc.

- **Coordinazione degli agenti** (*Agent Co-ordination*): In questa fase vengono stabiliti i protocolli di comunicazione tra gli agenti e le loro strategie di coordinazione. Esempi di protocolli e strategie sono master-slave per delegare compiti ai subordinati, contract-net per decidere a chi affidare un compito in un rapporto paritario, etc.

- **Generazione del codice ed implementazione dei compiti** (*Code Generation and Task Implementation*): Nella fase finale tutte le informazioni necessarie per generare il codice sono disponibili. L'unica cosa che rimane da definire sono le procedure che descrivono come un compito può essere svolto.

Queste fasi sono supportati dagli strumenti di design grafici di Zeus. Attualmente essi comprendono:

- Un editor di ontologie. Si definiscono gli oggetti che compongono l'ontologia in stile O-O, stabilendone i tipi, i valori di default, etc. (Figura 4.2)

- Un editor per i fatti e le variabili per descrivere le singole istanze dei concetti definiti con l'editor di ontologie.

- Un editor di definizione degli agenti usato per descrivere gli stessi con un approccio logico. Questo significa specificare i compiti , le risorse iniziali, e le dimensioni dell'agenda dei piani. (Figura 4.3)

- Un editor per la descrizione dei compiti per specificarene gli attributi e per comporre graficamente i compiti compositi (Figura 4.4).

- Un editor per definire le organizzazioni tra gli agenti.

- Un editor per selezionare le strategie di coordinazione dall'insieme delle strategie disponibili in Zeus (Figura 4.5).

Una volta definito il sistema tramite questi strumenti il codice corrispondente può venire automaticamente generato.

Una nota importante riguarda l'approccio e la notazione utilizzata per descrivere i ruoli: questa infatti ha fornito le idee di partenza per il diagramma di modellazione dei ruoli presentato in questa tesi.

Figura 4.2: L'editor di ontologie di Zeus.

4.1.2 Sodabot

Sodabot ([sod]) è un ambiente di sviluppo per agenti *general-purpose* sviluppato presso MIT Artificial Intelligence Lab.
Il suo componente principale è il *basic software agent (BSA)*, una struttura computazionale per costruire agenti, essenzialmente un sistema operativo dedicato agli agenti.
Nel corso del progetto è stato sviluppato un linguaggio di definizione per agenti (SodaBotL) le cui primitive sono vicine ad una descrizione in linguaggio naturale dell'attività dell'agente.
Utilizzando questo linguaggio l'utente può sviluppare con una certa facilità applicazioni classiche del paradigma ad agenti come dei personal assistant o

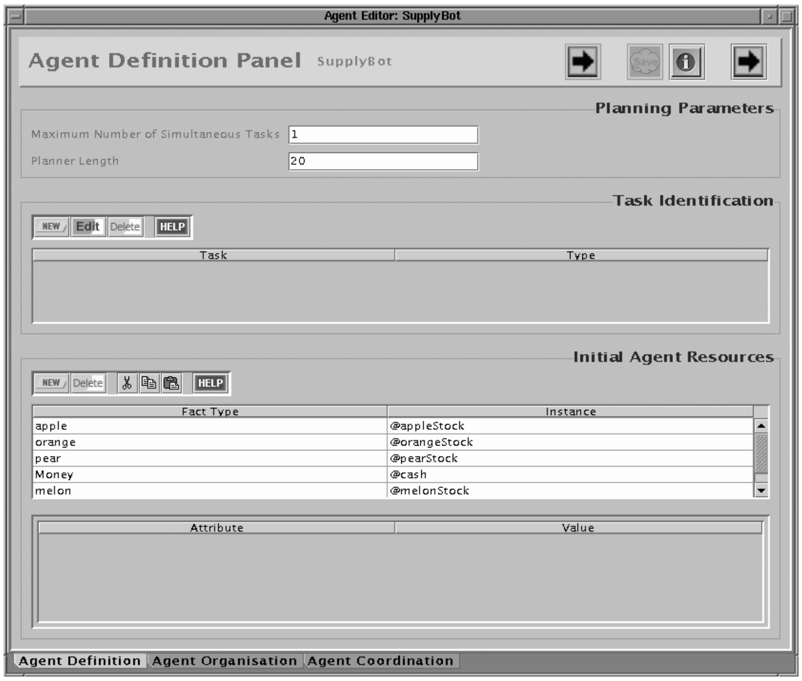

Figura 4.3: L'editor di agenti di Zeus.

degli agenti per la pianificazione degli appuntamenti.

Alcune caratteristiche particolarmente interessanti sono

- Gli agenti sono automaticamente distribuiti tra i BSA costituenti il sistema, semplificando notevolmente il lavoro dell'utente.

- Gli agenti scritti in SodaBotL possono essere ispezionati tramite una GUI del BSA prima di permettere il loro utilizzo nel proprio sistema.

- Le risorse disponibili agli agenti sono filtrate dal BSA aumentando il livello di sicurezza del sistema.

Un agente in SodaBot è caratterizzato dalle seguenti componenti:

- Dichiarazioni globali: accessibili a tutti i BSA su cui si trovano gli agenti.

- Il programma principale dell'agente: ovvero le istruzioni che sono eseguite dall'agente dal momento in cui viene attivato. Necessita della definizione dell'input richiesto alla sua attivazione ed il corpo del programma.

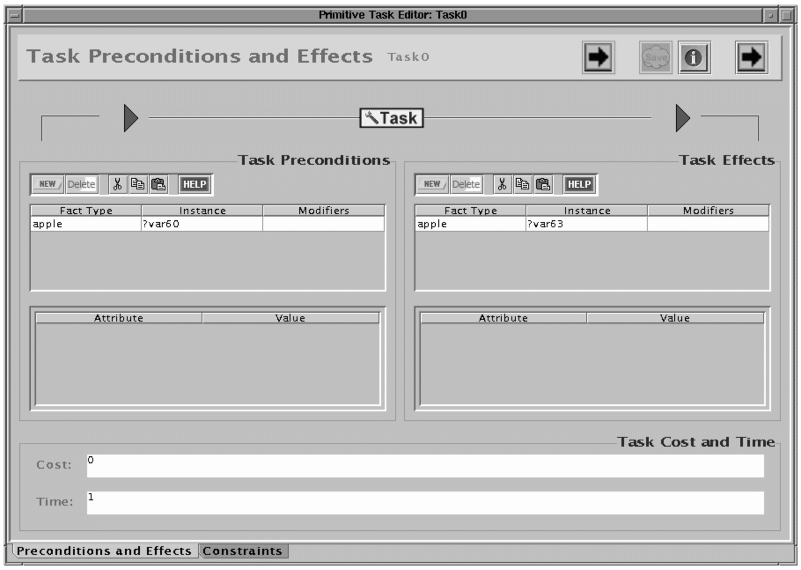

Figura 4.4: L'editor di compiti di Zeus.

- Le richieste: sono le sezioni atomiche dell'agente, che vengono distribuite nel sistema nell'ambiente adatto. Ogni richiesta viene caratterizzata da tre concetti base

  1. L'insieme dei messaggi che attivano la richiesta, e che possono quindi essere gestiti dall'agente ricevente.
  2. Cosa fare una volta ricevuto un messaggio, e come manipolare le informazioni ottenute.
  3. Come continuare, eventualmente, la conversazione, generando altre richieste per ottere lo scopo prefissato.

  In breve l'attività di un agente viene descritta dando una traccia delle conversazioni a cui prende parte, specificando come avviene la trasformazione di un input in un eventuale output.

- Le subroutines: le subroutine sono porzioni di codice chiamato unicamente dalle richieste per risolvere particolari problemi o effettuare computazioni.

SodaBot risolve affronta la definizione ed implementazione di agenti in modo innovativo, definendo un ambiente-sistema operativo che garantisca le prin-

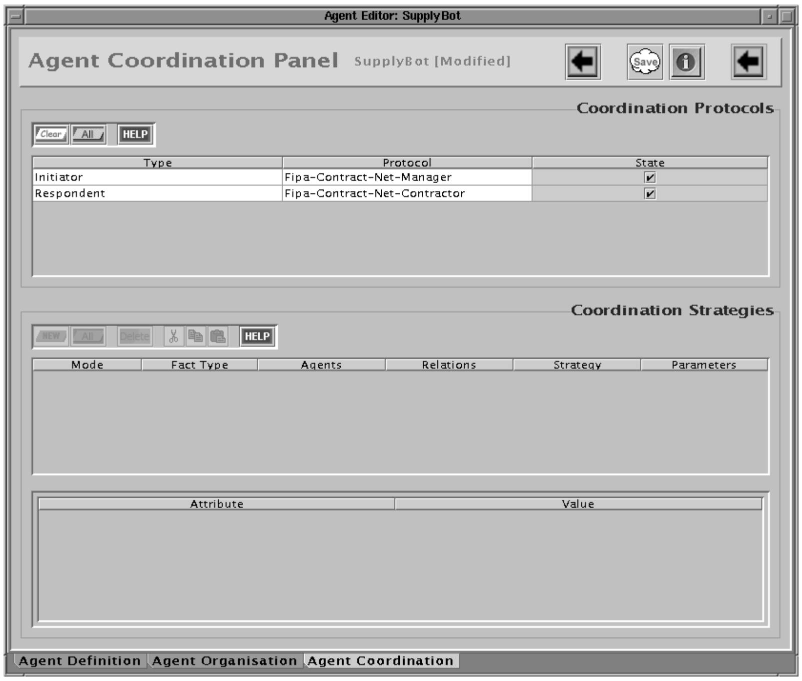

Figura 4.5: L'editor di strategie di coordinazione di Zeus.

cipali caratteristiche richieste da un agente, ed un linguaggio (una sorta di equivalente del C per Unix) per poter definire e programmare gli agenti.

4.1.3 Desire

La specifica di un sistema multi-agente attraverso l'uso di DESIRE ([des]) è basata sull'analisi dei task che il sistema deve realizzare.

I task vengono decomposti in una struttura gerarchica e ad ogni task vengono associati uno o più agenti. Nella specifica vengono descritti la struttura gerarchica dei task, l'interazione tra i livelli di tale struttura, la conoscenza che un agente ha dell'ambiente in cui è collocato e quella riguardante gli altri agenti del sistema.

I task trattati in DESIRE sono di due tipi:

- task complessi: sono descritti da uno o più sub task che possono assumere ancora la forma di task complessi;

- task primitivi: non possono essere decomposti in sub task, possono solo essere eseguiti.

I sistemi descritti con DESIRE sono detti sistemi composizionali : il sistema è formato da diverse componenti ognuna in relazione con i task individuati.

Le componenti sono descritte attraverso un linguaggio logico basato su un insieme di predicati che possono assumere tre valori di verità: vero, falso, indefinito. Esistono due tipi di componenti:

- componenti composte: descrivono la conoscenza necessaria per eseguire i task complessi del sistema, l'interfaccia di input-output della componente e le informazioni riguardo i sub task;

- componenti primitive: sono in relazione con i task primitivi, specificano cioé la conoscenza necessaria per eseguire i relativi task.

Ogni componente ha uno stato interno che descrive sia le informazioni generate durante l'esecuzione dei task che quelle ottenute dall'interazione con le altre componenti. La componente può inoltre trovarsi in uno tre stati di attivazione: active, idle, awake. Una componente passa dallo stato active a quello idle quando termina il processo per cui è stata attivata, mentre è nello stato awake quando attende informazioni.

Le componenti interagiscono tra loro utilizzando interfacce di input-output descritte da due segnature:

- segnatura di input: descrive i fatti che la componente può ricevere come input;

- segnatura di output: descrive i possibili fatti di output che la componente può computare.

Lo scambio di informazioni tra le interfacce delle componenti è basato su strutture chiamate collegamenti di informazione. I collegamenti di informazione specificano quali valori di verità un atomo usato in una componente assume in un'altra componente. Possono assumere tre stati: idle, up_to_date, awake.

Quando un collegamenti di informazione è nello stato up_to_date trasferisce l'informazione e ritorna nello stato idle, mentre è nello stato awake se attende informazioni. In un sistema composizionale la possibilità di cambiare sia lo stato interno delle componenti che lo stato dell'ambiente in cui sono collocate, è ottenuta dall'interazione tra differenti livelli di ragionamento : livello-oggetto, meta-livello, meta-meta-livello eccetera. Il livello-oggetto ragiona sullo stato dell'ambiente ed il meta-livello ragiona sul livello-oggetto. Questa struttura di livelli di ragionamento può essere ripetuta: in questo modo il meta-livello ha il ruolo di livello-oggetto per il meta-meta-livello e così via. Il risultato del ragionamento al meta-livello può essere utilizzato per guidare il ragionamento al livello-oggetto. Il meta-livello utilizza come informazione i risultati ottenuti al livello-oggetto, sia per il proprio ragionamento, sia per fornire al livello-oggetto le informazioni che necessitano per computare i propri risultati.

La specifica di un sistema composizionale descrive sia la struttura interna delle componenti che le relazioni tra queste. La struttura interna delle componenti è caratterizzata da:

- conoscenza riguardo alla struttura dei task;
- conoscenza riguardo alla sequenza in cui vanno eseguiti i sub task ;
- conoscenza riguardante lo scambio di informazione tra i sub task ;
- conoscenza riguardante la conoscenza della struttura dei task e la conoscenza sulla decomposizione dei task;
- conoscenza riguardante la delega dei ruoli;

mentre le relazioni tra componenti riguardano:

- decomposizione dei task: descrive la gerarchia dei task e le segnature di input e output per ogni sotto task;
- conoscenza relativa al controllo dei task: specifica l'attivazione della componente e delle sotto-componenti;
- collegamenti di informazione;
- allocazione della conoscenza per i task;
- allocazione dei task (tra gli agenti).

Gli agenti descritti da DESIRE sono chiamati agenti composizionali e sono modellati come componenti composte con l'aggiunta della conoscenza riguardante l'ambiente e gli altri agenti del sistema. La comunicazione tra agenti è quindi modellata allo stesso modo della comunicazione tra componenti, cioè tramite l'utilizzo di collegamenti di informazione. Ad esempio se un agente necessita di alcune informazioni per completare il proprio task trasferisce la richiesta ad un secondo agente tramite un collegamento di informazione. L'informazione richiesta è inviata, tramite un collegamento di informazione, all'agente richiedente, e può essere ottenuta sia come informazione presente nelle conoscenze dell'agente sia dal completamento di un task. Lo scambio d'informazione tra agente composizionale ed ambiente avviene in modo simile a quella tra agente e agente. L'osservazione dell'ambiente può essere modellata come una richiesta di informazione trasferita all'ambiente attraverso un collegamento di informazione; la risposta viene quindi inviata al richiedente tramite un apposito collegamento. Un altro tipo di comunicazione tra agente e ambiente è l'esecuzione di una azione da parte dell'agente. L'effetto di tale esecuzione trasferisce informazione dall'agente all'ambiente modificandone lo stato. La conoscenza relativa al controllo dei tasks rende possibile la specifica del comportamento degli agenti: riguarda le interazioni richieste per assicurare i fatti in input agli agenti e l'insieme dei fatti di cui vengono cercati i valori di verità (insieme degli obiettivi). Una componente ha successo se riesce a completare tutti i propri tasks, altrimenti fallisce. L'attivazione di una componente non sempre dipende dal completamento di un'altra: essa può essere attivata da un appropriato input.

Per concludere si possono quindi trarre alcune considerazioni.

In DESIRE il comportamento dell'agente e la strategia utilizzata nell'architettura per realizzarlo sono aspetti entrambi descritti all'interno della componente composta relativa all'agente, ma specificati in sotto-componenti separate che interagiscono tramite canali di comunicazione. Canali di comunicazione che a loro volta devono essere descritti nella specifica dell'agente.

Nel sistema multi-agente descritto attraverso l'uso di DESIRE gli agenti sono rappresentati come sotto-componenti della componente composta che rappresenta l'intero sistema multi-agente. Questa componente interagisce, attraverso canali di comunicazione, con la componente rappresentante l'ambiente. La componente che rappresenta un agente del sistema, è a sua volta una componente composta, in cui ogni sotto-componente ha una propria scala temporale.

La comunicazione tra agenti in DESIRE viene realizzata tramite canali di comunicazione ognuno dei quali viene definito nella specifica dell'agente. Tale comunicazione viene realizzata utilizzando un protocollo point-to-point : ogni agente sa con quale altro agente comunicare ed invia i messaggi solo a quell'agente.

In DESIRE il meta-ragionamento viene modellato usando una componente per il livello oggetto ed una per il livello meta. Ad esempio una componente può ragionare sul processo di ragionamento di un'altra componente.

4.1.4 AgentBuilder

AgentBuilder della Reticular Systems, Inc. ([age]) è uno strumento per la costruzione di agenti intelligenti. Agent builder è composto da due componenti principali

- Il toolkit include strumenti per sviluppare un software basato su agenti: tali strumenti consentono l'analisi del dominio applicativo, la progettazione e lo sviluppo del sistema di comunicazione, la definizione del comportamento degli agenti ed infine il debugging e il test del sistema.

- il sistema run-time comprende un *agent engine* che fornisce un ambiente per l'esecuzione del software ad agenti.

Gli agenti costruiti con AgentBuilder comunicano utilizzando KQML e supportano le performative in esso definite. Nuovi comandi di comunicazione possono comunque esssere aggiunti in caso di necessità.

Tutte le componenti di AgentBuilder sono state sviluppate in Java. Questo permette una più facile migrazione degli agenti stessi, e la possibilità di sviluppo su piattaforme eterogenee. Anche gli agenti sviluppati sono programmi Java cosicchè possono essere eseguiti su ogni piattaforma che dispone di una Java Virtual Machine.

L'architettura di agente utilizzata in AgentBuilder è una derivazione dei *Mentalistic Agents* sviluppati da Shoham [Sho91] per AGENT-0. Oltre alle quattro caratteristiche utilizzate in AGENT-0, ossia Beliefs, Capabilities, Commitments e Commitments rules vengono aggiunte le Behavioral rules.

- **Beliefs**: rappresentano le conoscenze dell'agente e sono aggiornate ad ogni nuova informazione che l'agente riceve.

- **Capabilities**: sono costrutti usati dall'agente per associare un azione con le precondizioni ad essa necessarie. Le capacità sono statiche ed immodificabili per tutta la durata di vita di un agente. Le azioni eseguibili sono a loro volta divise in due categorie: le azioni private e le azioni di comunicazione.

- **Commitments**: sono accordi (solitamente presi con un altro agente) per compiere una determinata azione ad un determinato tempo. Queste non garantiscono comunque che l'azione avrà luogo. Se nel frattempo le precondizioni dell'azione vengono a mancare l'azione non verrà eseguita.

- **Commitment rules**: sono le regole stabilite per poter attivare dei commitments; solitamente si presentano sotto forma di uno scambio di comunicazioni.

I Reticular Agent Mental Model aggiungono le **behavioral rules**. Esse estendono il modello di Shoham, in cui le azioni vengono solamente attivate da un commitment, con la possibilità da parte dell'agente di avere un insieme di regole che, sulla base dell'ambiente percepito, attivano determinate azioni. Le regole sono della forma When-If-Then ossia quando accade un determinato evento, si controllano delle condizioni che, se verificate, attivano determinate azioni.

Il motore dell'agente segue il seguente schema:

- processa i nuovi messaggi;
- determina che regole sono applicabili nella condizione corrente;
- esegue le azioni previste dalle regole;
- aggiorna il modello mentale in accordo con le regole attivate;
- esegue una fase di pianificazione.

Capitolo 5

CaseLP

CaseLP (Complex Application Specification Environment Based on Logic Programming) [MMZ97] è un ambiente sperimentale di prototipazione rapida per sistemi multi-agente basato su linguaggi logici. Esso è stato sviluppato presso il dipartimento di informatica e scienze dell'informazione dell'università di Genova.
CaseLP fornisce un metodo di prototipazione ed un insieme di strumenti e linguaggi che supportano la realizzazione del prototipo.

- Per la specifica architetturale statica del sistema viene fornito un linguaggio di descrizione dell'architettura, utile per descrivere il prototipo in termini di ruoli, servizi offerti e richiesti, classi d'agenti, istanze delle stesse ed i link di comunicazione tra essi (MAS-adl).
- per definire l'ontologia, ovvero i concetti principali del dominio del discorso, viene proposto un linguaggio ODL (Ontology Definition Language).
- Per la specifica di agenti un linguaggio non eseguibile basato su regole può essere usato per definire semplicemente agenti reattivi e proattivi (ACLPL).
- Un linguaggio eseguibile, basato sulla logica lineare, può essere usato per definire agenti maggiormente sofisticati ed il sistema in cui essi operano (Ehhf)
- Un linguaggio di definizione per agenti, dall'architetttura fino alla specifica di ogni singolo agente presente nel MAS, basato sul paradigma imperativo (HEMASL).

- A livello di implementazione, vengono fornite nuove primitive per estendere un linguaggio prolog-like ed un compilatore che traduce le specifiche da tale linguaggio in SICStus prolog ed in ECLiPSe.

- In ultimo vengono forniti dei tool di simulazione integrati all'interno di CaseLP per visualizzare l'esecuzione del prototipo e per raccogliere statistiche sul suo funzionamento.

La sezione 5.1 introduce gli agenti di CaseLP. La sezione 5.2 descrive a grandi linee il metodo di prototipazione utilizzato. Le sezioni seguenti focalizzano sui linguaggi e gli strumenti che vengono utilizzati nelle varie fasi di specifica e sviluppo del prototipo. Nello specifico la sezione 5.3 si occupa della fase di modellazione, la sezione 5.4 della fase di implementazione e la sezione 5.5 descrive gli strumenti usati nella fase di esecuzione del prototipo.

5.1 Agenti in CaseLP

Un agente in CaseLP può essere rappresentato da due diversi punti di vista.
Da un punto di vista descrittivo, un agente è caratterizzato da un *tipo*, un'*architettura*, un *interprete*, i *ruoli* che esso ricopre e, di conseguenza, un insieme di *servizi*.
Il *tipo* identifica un agente come logico, interfaccia o facilitatore:

- gli agenti logici forniscono il controllo e la coordinazione tra i componenti del MAS utilizzando le loro capacità di ragionamento;

- gli agenti interfaccia forniscono un interfaccia tra moduli software esterni al sistema e gli altri agenti che compongono il sistema stesso;

- I facilitatori forniscono agli altri agenti servizi di pagine gialle;

L'*architettura* si riferisce alla struttura interna dell'agente. Un'architettura specifica sia la struttura dati che ne forma le componenti interne che il flusso di controllo che guida l'attività delle componenti stesse.
L'*interprete* dipende dal tipo di modulo esterno a cui un agente interfaccia è collegato.
I *ruoli* possono essere considerati da due punti di vista:
come un insieme coerente di servizi o come le "impersonificazioni" che un agente attua nel corso della sua esistenza. Ad esempio un agente atto alla

compravendita ricoprirà alternativamente il ruolo di acquirente e di venditore. I *servizi* definiscono funzionalità che l'agente, o meglio un ruolo, fornisce o necessita per svolgere il proprio compito. I servizi dipendono tipicamente dal dominio dell'applicazione del prototipo.

Alcuni dei servizi offerti da un ruolo possono essere esportati ad utenti esterni al sistema o ad altri MAS, d'altra parte alcuni servizi possono essere importati dall'esterno del MAS. In tal modo i servizi esportati ed importati rappresentano un interfaccia tra il MAS nel suo insieme ed il mondo esterno. Essi sono funzionalità che il MAS fornisce o richiede ad utenti esterni, sia umani che altri MAS. I servizi esportati ed importati sono un sottoinsieme dell'insieme dei servizi offerti e richiesti. Ad esempio se CaseLP è usato per prototipare un sistema di supporto decisionale, almeno un servizio deve essere esportato per far si di consentire l'interazione con l'utente umano. Se invece il sistema prototipato è un simulatore l'insieme dei servizi esportati ed importati potrà essere vuoto.

Nel seguito della tesi considereremo un agente come una entità caratterizzata da

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

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

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

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

La applicazione delle istruzioni dell'engine allo stato ed al programma dell'agente ne determinano il comportamento. L'architettura di un agente deve consentirne la rappresentazione nei termini sopra descritti. Essa sarà quindi caratterizzata da componenti che contengono lo stato, componenti che contengono il programma, e da un engine che opera su queste componenti. Due agenti possono avere le stesse componenti, contenenti gli stessi dati, ma engines diversi, oppure possono avere lo stesso engine e le stesse componenti, contenenti però dati diversi. In entrambi i casi il comportamento degli agenti risulterà in generale differente.

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

Agenti che interagiscono tra di loro con le caratteristiche dette sopra possono essere descritti in CaseLP a differenti livelli di astrazione, usando i differenti linguaggi e strumenti forniti dall'ambiente. Un Metodo guida l'utente nei vari passi della prototipazione, dalla descrizione informale dell'applicazione allo sviluppo di un prototipo funzionale. In ogni fase l'utente può adottare il linguaggio ed lo strumento più adatto tra quelli forniti.

5.2 Metodologia di prototipazione

La realizzazione di un prototipo di un sistema software basato su MAS può essere svolta in accordo ai seguenti passi:

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

   (a) determinare quali ruoli sono necessari all'interno del sistema studiato. Tali ruoli sono insiemi logici di servizi, che suggeriscono funzioni complesse che gli agenti presenti nel sistema si troveranno a ricoprire. Per un trattamento più dettagliato vedere 3.4.1;

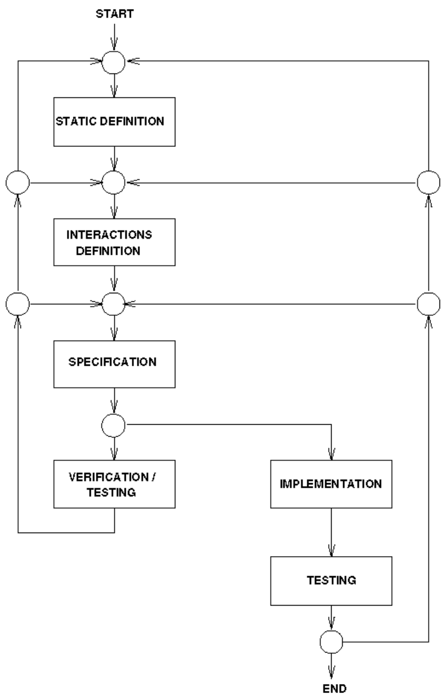

Figura 5.1: Una rappresentazione grafica della metodologia di sviluppo.

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

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

(d) Raggruppare i ruoli in classi di agenti, tenendo conto della metodologia vista in 3.4.2, stabilendo così quali classi di agenti sono necessarie all'interno dell'applicazione;

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

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

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

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

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

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

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

   (c) Specifica del comportamento dell'agente.

4. **Verifica e testing del sistema.**
In questo passo viene effettuata la verifica e/o la validazione preliminare della specifica astratta effettuata al punto 3. La validazione è possibile se il linguaggio di specifica è eseguibile. La verifica è invece possibile se il linguaggio di specifica usato mette a disposizione un meccanismo di prova. Se la specifica è stata fatta utilizzando un linguaggio logico, un'esecuzione goal-directed è un meccanismo di prova valido. In questo caso, data una specifica del prototipo ed una configurazione iniziale del

sistema, è possibile seguire l'evoluzione di un MAS in dettaglio. È altresì possibile verificare se una particolare computazione del sistema può essere eseguita o, cosa ancor più importante, che ogni computazione che inizia in particolare stato porti ad uno stato finale che soddisfi le proprietà ricercate.

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

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

Possiamo quindi distinguere tre fasi principali all'interno del metodo adottato da CaseLP. I passi dall'1 al 4 formano la fase di modellazione. Il passo 5 rappresenta la fase dell'implementazione e il passo 6 quella dell'esecuzione. Le sezioni successive tratteranno in maggior dettaglio i linguaggi e gli strumenti messi a disposizione da CaseLP per affrontare ognuna delle fasi sopra esposte.

5.3 Specifica del prototipo

In questa sezione vengono descritti i linguaggi e gli strumenti utilizzabili durante la fase di modellazione del prototipo.

5.3.1 MAS-adl

Nella primo passo del sistema di specifica viene utilizzato MAS-adl. MAS-adl è un semplice linguaggio per la definizione dell'architettura di MAS. Il

nome MAS-adl stà infatti per Multi Agent System Architectural Description Language.
Esso serve per dare una specifica ad alto livello dei componenti di un MAS e delle relazioni tra loro intercorrenti.
Esso viene usato per descrivere

- I ruoli che sono necessari all'interno del MAS, ed i servizi offerti e richiesti da ciascun ruolo. Tali ruoli vengono definiti secondo questo schema

```
role <RoleName> {
  requires <ServiceList>
  provides <ServiceList>
  import <ServiceList>
  export <ServiceList>
}
```

- I legami di soddisfazione tra i servizi offerti e richiesti dai ruoli. Ogni servizio offerto da un ruolo è collegato con uno o più servizi richiesti da un altro ruolo, o più ruoli possono offrire a d un singolo ruolo. Questo fornisce il canale di comunicazione (*link*) più generale.

- Stabilire una relazione gerarchica tra i ruoli tramite la le relazioni di contenimento e raffinamento.

- Definire le classi di agenti tramite i ruoli che essi contengono, *le eventuali relazioni di sottoclasse,* l'architettura, il tipo e, nel caso di agenti interfaccia, il programma verso cui fornisce l'interfaccia.

- le eventuali relazioni di soddisfacimento dei servizi oltre a quelle definite dalle relazioni tra i ruoli (o una loro maggior specifica). La struttura di comunicazione tra tutti gli agenti del MAS viene a formarsi tramite questa specifica di collegamenti.

- stabilire e dichiarare il numero di agenti esistenti nel MAS, ossia quante istanze delle classi di agenti esistono per ciascuna classe.

Una sintassi formale di MAS-adl viene riportata nell'appendice A.

5.3.2 Strumenti per il modello di conversazione

Il secondo passo del metodo descrive il modello di conversazione del MAS tramite un linguaggio che è una controparte testuale del diagramma dei protolli presentato in 3.6.
Il linguaggio di comunicazione prescelto è un sottoinsieme di KQML [FFMM94]. Il modello è stabilito scegliendo per ogni link stabilito antecedentemente una sequenza complessa di messaggi detta conversazione o protocollo, così come le performative ed il contenuto di ogni messaggio. All'interno della sequenza possono essere inserite scelte tra messaggi diversi o differenziazioni nel comportamento in seguito a particolari messaggi. Per poter specificare comportamenti complessi sono state messe a disposizione diverse primitive

- **Goto e label**: spesso alcune comunicazioni assumono un aspetto ciclico fino a quando non viene realizzata una condizione: ad esempio nelle trattive per trovare un accordo tale comportamento può facilmente presentarsi. Un altro fattore che si può presentare sono condizioni che fanno terminare o passare ad un'altra fase del protocollo. Al fine di definire tali cicli e salti è stato inserito il meccanismo del Goto e delle label.

- **connettivi logici tra messaggi**: spesso non viene seguito un percorso univoco all'interno di un protocollo: può esserci la possibilità di scelta tra alcuni messaggi (*Xor o Or esclusivo*), più messaggi possono essere spediti in parallelo (*And o parallelismo*) oppure è possibile che si verifichi l'invio di solo alcuni messaggi in parallelo (*parallelismo in Or o Or inclusivo*).

- **switch e different lifepath**: il comportamento di un agente varia a seconda dei messaggi ricevuti dalle altre entità presenti in un MAS. Per poter modellare queste situazioni serve poter stabilire possibili cicli di vita del protocollo a seconda di situazioni differenti. Per tale scopo è stata definita una direttiva *Switch*

- **subconversation**: è possibile che da una conversazione ne scaturisca un'altra, la conversazione che viene a nascere viene definita subconversazione.

- **guard condition**: a volte un messaggio viene inviato solo se alcune condizioni dello stato dell'agente sono verificate. Per poter modellare queste situazioni è utile stabilire una guardia all'interno della specifica del messaggio. Tale messaggio verrà inviato solo se la condizione di guardia è verificata.

Le definizioni di protocolli possono essere parametriche, in modo da stabilire protocolli generici da specializzarsi in seguito. Tali protocolli generici possono essere utilizzati come una libreria di protocolli.

5.3.3 Ontology Definition Language

Dato che uno dei requisiti fondamentali per gli agenti è la capacità di comunicare informazioni è evidente la necessità di adottare un vocabolario comune all'interno del MAS.
Tale vocabolario viene definito ontologia.
Un ontologia è un insieme di conoscenze che rappresentano la specifica di un concetto chiave all'interno di un dominio applicativo. Per isolare tali concetti chiave basti pensare se è possibile far interagire degli agenti senza che abbiano conoscenza del concetto stesso. Se tali interazioni risultano prive di senso allora il concetto è significativo. La definizione di un'ontologia segue lo schema in fig 5.2.

```
ontology ontology_name{
  concept_name_1 abstract {
    name_1: type [constraints]
    ...
    name_n: type [constraints]
  }
    concept_name_j effective contains:concept_name_h{
    name_1: type [constraints]
    ...
    name_x: concept_name_h [constraints]
  }
    concept_name_m effective refine:concept_name_1{
    name_1: type [constraints]
    ...
    name_w: type [constraints]
  }
}
```

Figura 5.2: Uno schema per la definizione di ontologie.

La parola chiave *ontology* introduce la definizione dei concetti definiti all'interno del dominio preso in esame, denominato, nello schema, *ontology_name*. ogni concetto può essere astratto o concreto.

- *astratto* significa che il concetto non è un tipo reale del dominio ma un tipo generico (es. mercanzia). Tale concetto deve venire in seguito raffinato.
- *concreto* significa che il concetto è un entità reale del dominio (es. pere).

Un concetto viene definito tramite una lista di attributi. Ogni attributo verrà specificato tramite

- il *nome*: necessario per distinguere gli attributi tra loro.
- il *tipo*: i tipi ammessi sono : stringhe, interi, reali , booleani oppure un concetto definito antecedentemente (in tal caso il concetto utilizzato deve venire evidenziato nella clausola contains).
- gli eventuali *vincoli*:i *constraints* sono formule di logica equazionale del primo ordine e stabiliscono vincoli sui valori che possono essere assunti dall'attributo.

Tra i concetti può essere definita una gerarchia tramite la clausola *refines* che stabilisce una relazio di sottoclasse tra i concetti.
Una BNF completa si trova nell'appendice C.

5.3.4 $\mathcal{E}_{hhf}$

Il linguaggio logico lineare $\mathcal{E}_{hhf}$ [Del96] è un linguaggio eseguibile per modellare sistemi concorrenti con gestione di risorse finite basato sul linguaggio Forum [Mil96]. $\mathcal{E}_{hhf}$ è un linguaggio basato su logica multiset che combina caratteristiche di linguaggi logici estesi come λProlog, cioé goals con implicazione e quantificazione universale con la nozione di formule come risorse che è alla base della logica lineare. Riferendosi al passo 3 della metodologia di sviluppo di CaseLP $\mathcal{E}_{hhf}$ può essere usato per specificare tutti e tre i livelli di modellazione.

5.3.5 HEMASL

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

5.3.6 LIbreria di specifiche architetturali

La specifica dell'architettura di un agente è sicuramente uno degli aspetti più complessi della specifica di un MAS. Per aiutare lo sviluppatore in questa fase CaseLP mette a disposizione una libreria di architetture specificate in $\mathcal{E}_{hhf}$ da cui attingere il modello più utile. Attualmente la libreria comprende quattro specifiche:

- reattiva
- proattiva
- reattiva-proattiva e
- Procedural Reasoning System (PRS) []
- BDI

5.3.7 ACLPL

ACLPL è un linguaggio a regole che viene utilizzato per programmare il comportamento di agenti reattivi, proattivi, reattivi-proattivi.
Il comportamento degli agenti reattivi viene definito mediante regole del tipo *evento-condizione-azione*; gli agenti proattivi tramite regole del tipo *condizione-azione*.

Per eseguire un *azione* un agente deve essere (eventualmente) attivato mediante la percezione di un particolare *evento*, correntemente implementato come un messaggio ricevuto dall'agente, ed una *condizione* deve essere soddisfatta dallo stato dell'agente. Le azioni possono essere o aggiornamentio dello stato dell'agente o l'invio di un messaggio ad altre entità del sistema. Il comportamento degli agenti reattivi-proattivi viene specificato utilizzando entrambi i tipi di regole viste sopra. La sintassi di ACLPL è brevemente illustrata qua sotto.

Behavior ::= **behavior** RulesList **endbehavior**
RulesList ::= Rule | Rule RulesList
Rule ::= ReactiveRule | ProactiveRule
ReactiveRule ::= **on message** Msg **check** Condition **do** ActionsList
ProactiveRule ::= **check** Condition **do** ActionsList
Msg ::= Performative {**content:** Content; **sender:** Sender; **receiver**: Receiver;}
Condition ::= StateCondition **and** AuxiliaryCondition
StateCondition ::= **true** | Goal
AuxiliaryCondition ::= **true** | Goal
Goal ::= A | A **and** Goal | A **or** Goal
Action ::= **assert_state**(Fact) | **retract_state**(Fact) | **send**(Receiver, Msg)

Ad ogni agente viene assegnato uno stato iniziale composto da un insieme (possibilmente vuoto) di formule atomiche ground. Una regola reattiva è attivata da un messaggio prelevato dalla mailbox dell'agente e, se la condizione è soddisfatta, l'azione corrispondente viene eseguita. Se il prototipo contiene solo agenti reattivi, proattivi o agenti proattivi-reattivi ACLPL può essere utilizzato nella fase 3 al posto di $\mathcal{E}_{hhf}$ o HEMASL. ACLPL è comunque limitato alla specifica del comportamento dell'agente.

5.4 Implementazione del prototipo

Il quinto passo del metodo proposto riguarda l'implementazione di una versione "concreta"del prototipo. Il linguaggio è il **Prolag** (*Prolog per agenti*), una versione estesa multipiattaforma del Prolog. L'ambiente CaseLP è infatti stato realizzato sia in ECLiPSe che in SICStus Prolog: due compilatori effettuano la traduzione da Prolag nel linguaggio di implementazione prescelto.

5.4.1 Estendere il Prolog per sviluppare agenti

Prolag è un Prolog standard esteso con predicati di comunicazione e di aggiornamento di stato sicuri (cioé con possibilità di backtrack). Due predicati vengono utilizzati per aggiornare lo stato.

- assert_state(Fact): asserisce il fatto Fact nello stato dell'agente. La sua implementazione è sicura poiché, in fase di backtracking, il fatto antecedentemente asserito viene rimosso dallo stato.

- retract_state(Fact): Rimuove il primo fatto dello stato dell'agente che unifica con Fact. Se avviene un backtrack, il fatto viene ri-asserito.

Tre predicati sono stati definiti per permettere ad un agente di ispezionare la sua mailbox e leggere i messaggi in essa contenuti.

- sync_receive(Message): blocca l'agente che effettua la chiamata finché non arriva nella mailbox un messaggio che unifica con Message.

- async_receive one(Msg_input, Msg_output): Ricerca nella mailbox dell'agente un messaggio che unifica con Msg_input. Se tale messaggio esiste Msg_output viene unificato con esso, altrimenti Msg_output unifica con l'atomo no_message.

- async_receive_all(List_of_messages) prende tutti i messaggi dalla mailbox dell'agente e li unifica con List_of_messages.

Un predicato viene infine fornito per permettere all'agente di inviare messaggi.

- send(Receiver, Message): Invia il messaggio Message dall'agente che effettua la chiamata di predicato all'agente Receiver. Il predicato Send è implementato utilizzando i predicati di aggiornamento sicuri assert_state e retract_state.

5.4.2 Implementare un prototipo funzionante

Per implementare un prototipo funzionante, deve essere fornito il codice Prolag di ogni agente presente nel sistema. Tale codice deve

1. implementare tutte le caratteristiche computazionali che caratterizzano l'agente: il suo stato iniziale, il suo comportamento ed il suo engine; e

2. rispettare la specifica dell'agente e del sistema data nei passi da 1 a 3 della metodologia di specifica.

Per gli agenti interfaccia sarà necessario definire i predicati dell'interprete.

5.5 Esecuzione del prototipo

Dopo che tutti gli agenti sono stati implementati come codice Prolag, essi possono essere compilati per poter essere utilizzati per la simulazione. la primitiva di compilazione compile_agent è stata implemantata in modo da poter prendere più file in input, questo per permettere il riuso dello stesso file Prolag per più agenti diversi che differiscono solo per pochi particolari del loro stato o comportamento. L'output di questa primitiva è un modulo SICStus o ECLiPSe. Dopo che tutti i moduli sono stati generati, lo sviluppatore del sistema deve decidere quali agenti caricare nel sistema, come inizializzare le loro mailbox e decidere se far partire la simulazione seguendo la sua esecuzione tramite un interfaccia grafica (solo per la versione ECLiPSe) o dando comandi tramite una shell SICStus.

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

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

La figura 5.3 descrive la finestra che consente l'inizializzazione della mailbox di un agente; la traccia on-line della esecuzione è mostrata in figura 5.4 dove gli eventi e lo stato relativi ad ogni agente vengono mostrati. La figura 5.5 mostra la traccia della esecuzione off-line, con eventi e messaggi mostrati sulle linee orizzontali (agenti), in corrispondenza dell'istante in cui accadono (dimensione verticale, tempo). I dettagli di ogni evento possono essere ottenuti cliccandovi sopra. La finestra che si apre è mostrata in figura 5.6.

Durante la simulazione vengono mostrati dati riguardanti gli agenti "Instrumentati".Al termine della simulazione un traccia completa di tutti gli eventi che sono stati selezionati può essere visualizzata. La "Instrumentazione" è completamente indipendente dall'esecuzione, e quindi non influenza futuri cambiamenti del supporto di esecuzione.

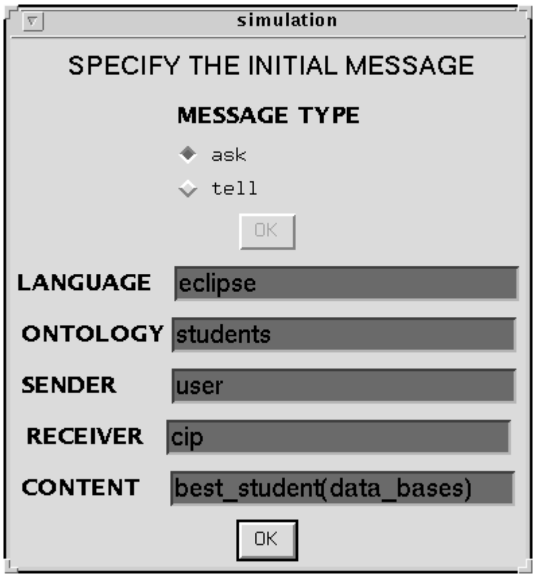

Figura 5.3: Finestra per l'inizializzazione delle mailboxes.

5.5.2 Comandi di shell per l'esecuzione della simulazione

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

- `compile_agent(List_of_modules_and_dynamic_predicates_files, List_of_state_files, List_of_behaviour_files, Agent_name, Compiled_file_name)`:
  Il primo argomento è una lista di files nei quali sono contenute le dichiarazioni dei moduli esterni che l'agente dovrà usare e la lista dei predicati dinamici.
  Il secondo argomento è la lista dei files nei quali sono contenuti i fatti che rappresentano lo stato iniziale dell'agente.
  Il terzo argomento è la lista dei files che contengono le regole che governano l'esecuzione dell'agente (è necessario definire il predicato "interpret" per ogni agente, e tutti i predicati ausiliari).
  Il quarto argomento è il nome dell'agente (le primitive per inviare messaggi hanno un argomento che è il nome dell'agente a cui mandare il messaggio; tale nome deve essere unico).

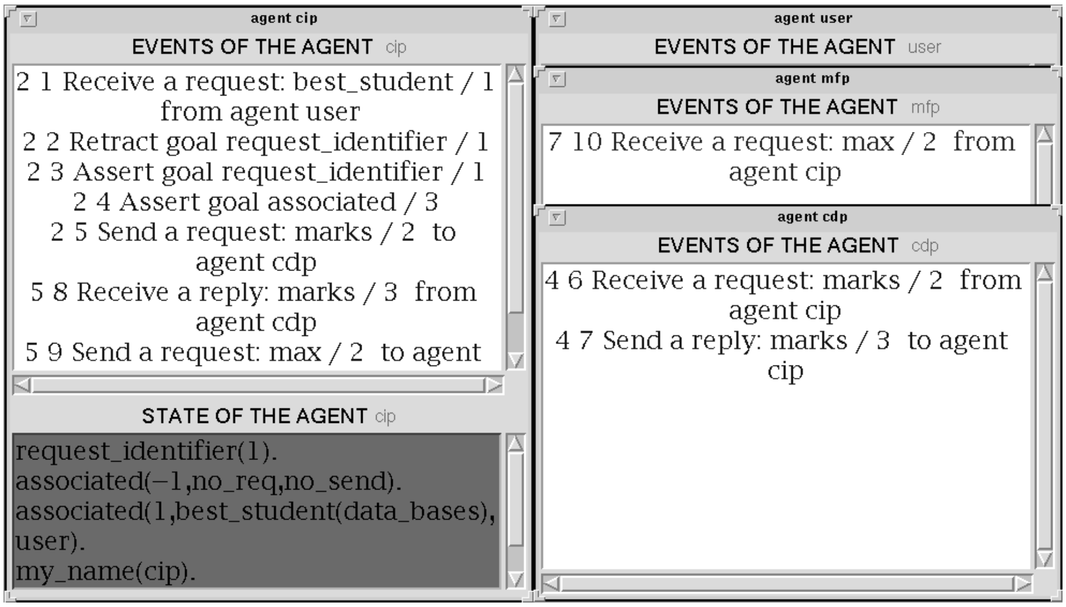

Figura 5.4: finestra che descrive l'esecuzione on line.

Il quinto argomento, infine, è il nome del file che verrà generato a seguito di questo processo di compilazione. Tale file contiene il codice completo dell'agente, e verrà usato dal predicato **load** quando l'agente verrà caricato nel sistema.

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

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

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

- `net_parameters(From_ag, To_ag, Failure_rate, Min_latency, Max_latency, Unit)`: seleziona i parametri di comunicazione tra ogni

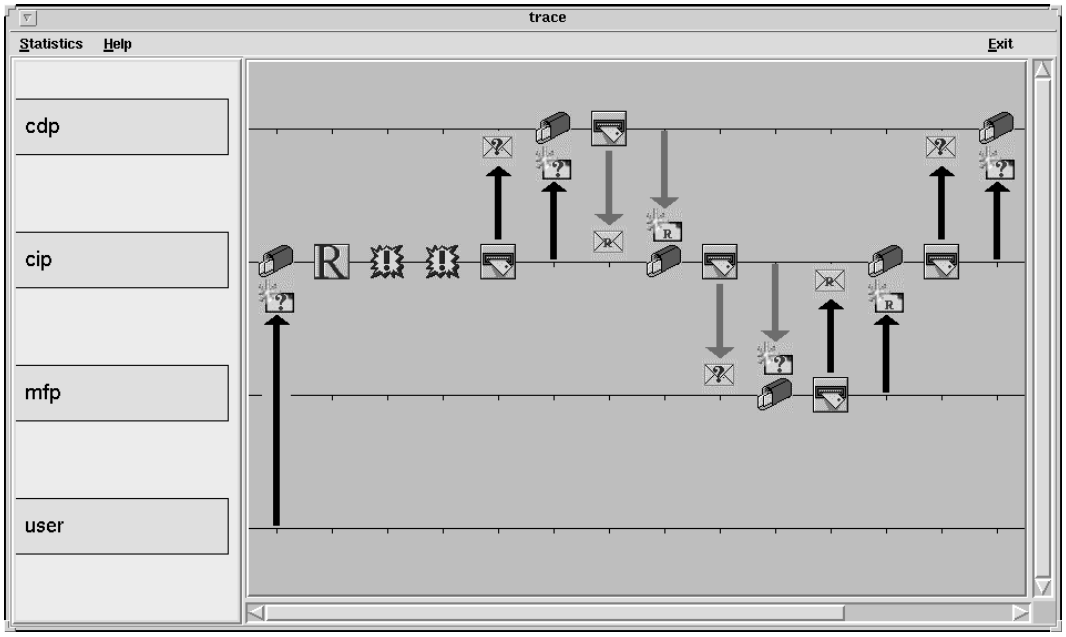

Figura 5.5: Traccia off-line delle esecuzioni.

coppia di agenti presenti nel sistema; From_ag e To_ag specifica la coppia di agenti, Failure_rate specifica quanti messaggi su 100.000.000 sono persi per problemi di comunicazione, Min_latency e Max_latency e Unit specificano il minimo ed il massimo tempo di ritardo sul canale espresso in Unit. Se per una coppia di agenti non sono specificati parametri la comunicazione è immediata e totalmente affidabile.

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

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

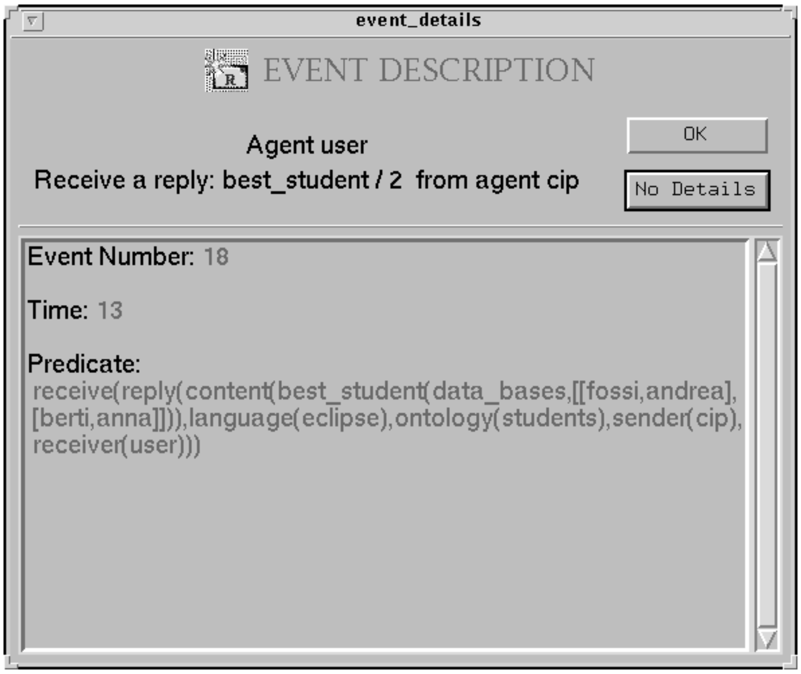

Figura 5.6: Dettagli di un evento tracciato.

Capitolo 6

HEMASL

Diamo la descrizione del linguaggio prescelto per la specifica degli agenti all'interno del metodo di CaseLP per gli scopi di questa tesi. La scelta è ricaduta su questo linguaggio per due ragioni:

- un più facile utilizzo da parte dello sviluppatore piú abituato al paradigma imperativo che logico;

- una più facile mappatura da e verso UML che, essendo basato sul paradigma ad oggetti, è anch'esso imperativo.

Questo linguaggio era stato presentato precedentemente nella tesi di Simone Marini [Mar99], ma alcuni cambiamenti sono stati fatti per renderlo più vicino al paradigma O-O. I cambiamenti principali sono:

- attualmente il linguaggio è fortemente tipato;

- la sintassi dei comandi si avvicina maggiormente a quella di linguaggi O-O;

- sono stati inseriti concetti come tipi template, interfacce, ed il concetto di sottoclasse tra le classi di agenti; è stata inoltre introdotta la possibilità di importare classi definite in UML;

- in accordo con le modifiche introdotte in MAS-adl sono stati esplicitati i ruoli ed i servizi svolti dagli agenti;

6.1 Concetti di base del modello proposto

Al fine di permettere una maggiore modularità e flessibilità nella definizione di agenti eterogenei introduciamo una gerarchia di livelli di astrazione, illustrata in figura 6.1.

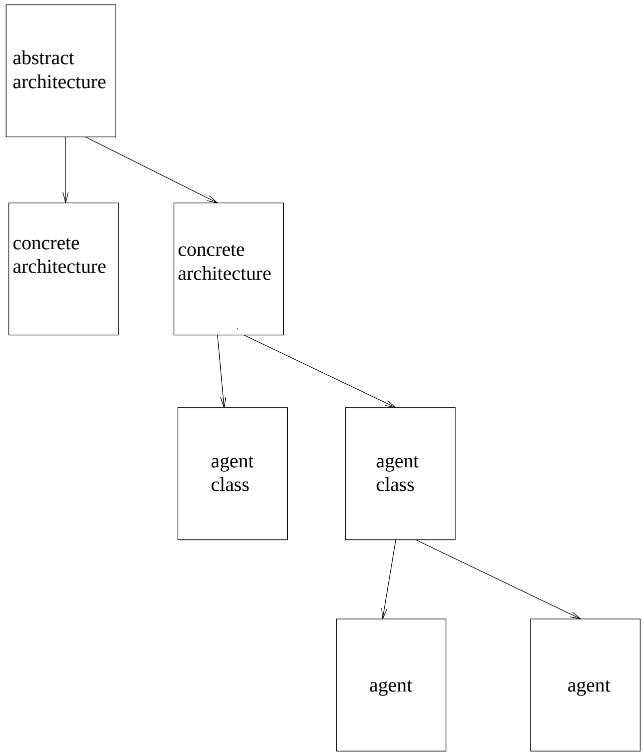

Figura 6.1: La gerarchia architetturale di HEMASL.

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

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

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

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

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

  1. scegli a caso una regola;
  2. verifica che la testa della regola sia uguale all'evento;
  3. verifica che le condizioni sulle credenze espresse nella regola siano soddisfatte;
  4. se entrambe le verifiche hanno successo restituisci la regola scelta, altrimenti ritorna al punto 1.

  oppure, un altra scelta potrebbe essere

  1. costruisci l'insieme di tutte le regole applicabili;
  2. restituisci la regola che, tra queste, prevede il minor numero di azioni da eseguire.

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

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

  - ad ogni componente si associa un tipo tra quelli previsti dal linguaggio;

  - per ogni istruzione ad alto livello contenuta nell'engine si sceglie una delle possibili indicazioni su come realizzarla (se tali indicazioni sono fornite) e se ne dà una implementazione concreta.

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

- Classe di agente. Una classe si definisce a partire da una architettura concreta istanziando le componenti contenenti il programma dell'agente. Tali componenti saranno definite a livello astratto. Se Hemasl viene utilizzato per raffinare una specifica data in MAS-adl seguendo il metodo di progettazione di CaseLP, si specificano i ruoli ricoperti dalla agent class. Vi è anche la possibilità di estendere una classe precedentemente definita tramite un rapporto di sottoclasse.

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

Per illustrare meglio le potenzialità della gerarchia introdotta consideriamo uno scenario in cui vogliamo definire agenti con architettura BDI.
L'architettura BDI è stata definita in [RG95] è basata su tre concetti principali

- **Beliefs**: le credenze dell'agente ovvero ciò che l'agente conosce in un determinato momento.

- **Desires**: i desideri ovvero ció che l'agente vuole fare of ottenere.

- **Intentions**: ovvero ciò che l'agente stà facendo, i piani che l'agente stà portando avanti.

La modalità di funzionamento è la seguente

1. Generare una lista di opzioni. Tali opzioni sono i mezzi per soddifare le intenzioni correnti e nuove opzioni vengono generate sulla base delle credenze e dei desideri dell'agente. Le opzioni a disposizione dell'agente

che sono i mezzi per soddisfare le sue intenzioni correnti sono generate tramite raffinamento; similmente alla funzione di raffinamento di un pianificatore gerarchico (la parte in figura 6.2 denominata refinement). Le opzioni disponibili dell'agente che sono alternative alle intenzioni correnti sono generati sulla base delle sue credenze correnti e i suoi desideri (nella figura la parte denominata alternatives).

2. Si seleziona un sottoinsieme di tali opzioni per essere adottate. La selezione viene normalmente guidata dalle credenze, desideri ed intenzione dell'agente. Le opzioni selezionate sono aggiunte alle intenzioni dell'agente.

3. se esistono azioni atomiche all'interno delle intenzioni, esse devono essere eseguite

4. infine, se l'agente ha soddisfatto un intenzione o gli risulta che un intenzione non sia più realizzabile, tale intenzione viene scartata.

5. Si aggiornano le credenze dell'agente e si torna al punto 1.

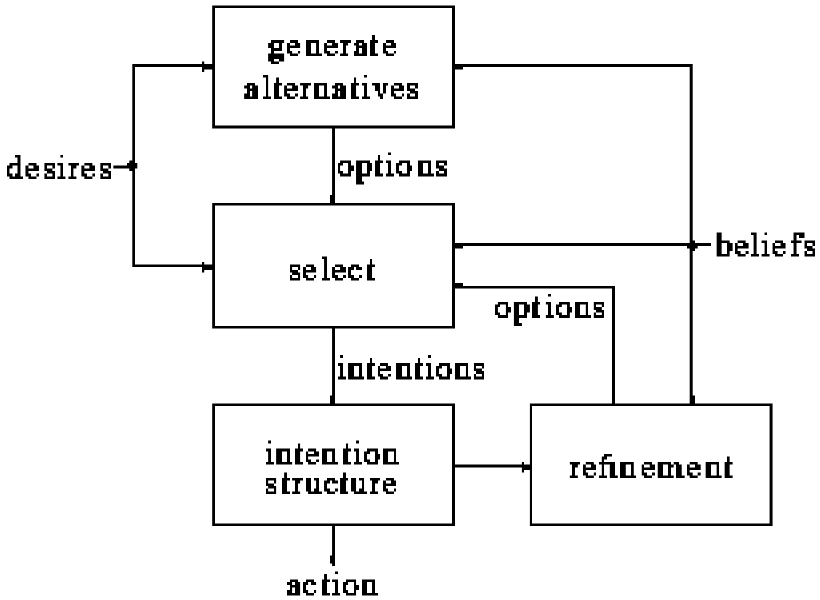

Figura 6.2: L'architettura BDI

L'architettura dell'agente è quindi costituita da un engine del tipo

```
repeat
  opzioni := generatore-opzioni(coda-eventi);
  opzioni-estese := deliberatore(opzioni);
  aggiorna-intenzioni(opzioni-estese);
  esegui-intenzioni();
  percepisci-evento-esterno();
  elimina-intenzioni-eseguite();
  elimina-intenzioni-impossibili();
  elimina-opzioni-impossibili();
end repeat
```

e da tre componenti: la prima componente mantiene le credenze riguardanti l'ambiente in cui l'agente si trova; la seconda mantiene la conoscenza procedurale dell'agente costituita dai piani che l'agente può intraprendere quando un evento viene percepito; infine, la terza componente mantiene le intenzioni dell'agente, cioè i piani scelti per essere eseguiti. La prima e la terza componente contengono pertanto lo stato dell'agente, mentre la seconda contiene il programma. Se inizialmente la terza componente contiene delle intenzioni, esse vengono eseguite all'attivazione dell'agente. Il comportamento di due agenti con uguale architettura differisce a causa dell'informazione contenuta nelle loro componenti.

Le componenti dell'architettura corrispondono alle strutture dati dell'agente. Quando le componenti vengono implementate bisogna assegnare loro un tipo, al fine di poterle accedere e manipolare utilizzando le istruzioni appropriate. Ad esempio la componente che mantiene le intenzioni può essere implementata come uno stack o può essere una lista. Nelle architetture concrete che implementano una architettura astratta, una stessa componente può essere realizzata utilizzando tipi diversi. Questo non snatura l'architettura degli agenti, che a livello astratto rimane invariata.

L'engine dell'architettura è un programma composto da istruzioni che agiscono sulle componenti. Facendo riferimento all'esempio precedente, l'attivazione di un piano è subordinata alla sua scelta all'interno della componente conoscenza procedurale . Il modo in cui viene effettuata questa scelta è codificato nell'engine della architettura concreta, e può differire da una architettura concreta ad un'altra. Ad esempio, un agente può scegliere i piani in modo diverso da un altro agente: il primo può scegliere il primo piano che può essere utilizzato, mentre l'altro agente sceglie il migliore, rispetto ad un certo criterio, tra tutti quelli che possono essere utilizzati. Questo comporta una diversa definizione della procedura che descrive la scelta di un piano all'interno della componente conoscenza procedurale . Anche in questo caso l'architettura astratta degli agenti è la stessa nonostante una procedura dell'engine abbia definizioni differenti.

La definizione di un sistema multi-agente deve quindi prevedere la descrizione delle architetture astratte, delle architetture concrete, delle classi di agenti e delle istanze di queste classi.

Quanto descritto fino ad ora riguarda la rappresentazione e la realizzazione di ciò che sta all'interno di un agente. Abbiamo però osservato che un agente, per essere tale, deve poter compiere delle azioni che lo mettono in relazione con l'ambiente e con gli altri agenti nel sistema. Abbiamo pertanto la necessità di rappresentare l'ambiente in cui gli agenti sono collocati e di consentire loro di percepirlo e modificarlo. Dobbiamo inoltre fornire agli agenti uno strumento per comunicare.

La soluzione adottata prevede l'esistenza di un ambiente (ENV) che contiene la rappresentazione di tutti i fatti dell'ambiente fisico che sono rilevanti per il particolare dominio applicativo. Gli agenti possono percepire l'ambiente direttamente, ma lo possono modificare soltanto interagendo con un agente ambiente . Si supponga ad esempio che un agente robot con architettura BDI possa eseguire una azione primitiva accendi_il_motore. Anche se l'agente robot non lo può sapere, gli effetti di questa azione sull'ambiente sono molteplici: aumenteranno ad esempio il rumore, la temperatura ed il tasso di inquinamento. Non sarebbe quindi realistico consentire all'agente robot di modificare direttamente l'ambiente aggiungendovi solo l'informazione che il suo motore è stato acceso: ci sono conseguenze di questa azione che non sta al robot calcolare, ma che possono essere rilevanti nel dominio applicativo considerato. Per questa ragione si introduce l'agente ambiente, la cui definizione è dipendente dal dominio applicativo ed il cui compito è quello di calcolare tutte le conseguenze delle azioni primitive che gli agenti eseguono e di aggiornare l'ambiente di conseguenza. Le azioni primitive sull'ambiente che un agente può compiere sono tradotte in messaggi all'agente ambiente che sa come gestirle e come modificare ENV coerentemente. Passando infine alla comunicazione, gli agenti possono comunicare con l'agente ambiente e tra loro o direttamente, emulando una sorta di comunicazione "cablata" che richiede la conoscenza di un "indirizzo" dell'agente con cui si vuole comunicare, o tramite l'etere (ETH) che non richiede una conoscenza a priori della struttura del MAS. L'ETH è una struttura dati nella/dalla quale vengono scritti/letti i messaggi. Il linguaggio per la implementazione concreta dell'engine di un agente fornisce primitive per la comunicazione di tipo send, per mettere un messaggio nell'etere, e receive per leggere un messaggio dall'etere, nonché varianti di queste primitive di base.

6.2 Sintassi del linguaggio

In questa sezione viene descritta la sintassi del linguaggio HEMASL, utilizzato per decrivere tutti i livelli di astrazione che caratterizzano gli agenti ed il MAS ossia architetture astratte, concrete, classi di agenti e la loro istanziazione

Per poter definire queste categorie sintattiche bisogna prima definire:

- l'insieme dei tipi τ supportati dal linguaggio;
- il linguaggio delle espressioni;
- l'insieme delle istruzioni per l'accesso e la modifica delle componenti della architettura;
- l'insieme delle istruzioni per la ricezione e l'invio di messaggi;
- l'insieme delle istruzioni per la percezione e la generazione di eventi nell'ambiente.

6.2.1 L'insieme dei tipi e delle espressioni tipate

HEMASL è un linguaggio tipato. Si deve quindi fornire un supporto di tipi di dato minimale con cui eventualmente costruire tipi di dato complessi. I tipi base di hemasl sono:

- gli interi (int)
- i reali (real)
- i booleani (bool)
- i caratteri (char)
- le stringhe (string)

I costruttori di tipi definiti in hemasl sono:

- le liste con capacità di acesso sia in testa che in coda (list)

- container con strategia LIFO o stack
- container con strategia FIFO o code (queue)
- gli insiemi privi di ordinamento (set)
- le tuple tipate (tupla)

HEMASL altresìammette variabili di tipo, importazione di tipi definiti in UML (quindi di classi) e costrutti di definizione di tipi.

Espressioni tipate

Per poter operare sui tipi forniti è necessario poter avere a disposizione un insieme di operatori sufficientemente completo. Per ogni tipo vengono forniti :

- un insieme non vuoto di costanti, o di operatori iniziali
- un insieme di operazioni che operano su argomenti eterogenei e che restituiscono risultati del tipo in questione.

Dato che uno dei contributi originali di questa tesi è stressare la strutturazione di hemasl in termini di oggetti e classi, la sintassi delle operazioni rispecchia la notazione standard dei linguaggi O-O. Nel seguito si fornisce una trattazione informale delle operazioni (o metodi) disponibili per i vari tipi strutturati ed il loro significato. Le operazioni attinenti ad un certo tipo di dato vengono descritte tutte insieme, anche se il tipo delle espressioni contenenti quelle operazioni può non essere omogeneo. Si pensi ad esempio alla operazione di estrazione dell'elemento in testa ad una lista l di interi. Il tipo dell'espressione l.gethlist() è int e non list, tuttavia l'operazione l.gethlist() è caratteristica delle liste e viene pertanto descritta nel relativo paragrafo.

Operazioni sulle liste

- "l.**e_l()**": la lista l viene istanziata come lista vuota.
- "b = l.**is_e_l()**": controlla se la lista l è vuota. restituisce un valore booleano che vale true se la lista è vuota, false altrimenti.
- "l.**inshlist**(e)": inserimento in testa alla lista l dell'elemento e.

- “l.**instlist**(e)”: inserimento in coda alla lista l dell’elemento e.

- “x = l.**gethlist()**”: recupero dell’elemento in testa alla lista l. Il tipo del risultato x è il tipo dell’elemento estratto. La lista l non viene modificata.

- “x = l.**gettlist()**”: recupero dell’elemento in coda alla lista l. Come per l’operazione precedente, il tipo del risultato dipende dall’elemento estratto e la lista l non viene modificata.

- “l.**delhlist()**”: rimozione dell’elemento in testa alla lista l. La lista l viene privata dell’elemento in testa.

- “l.**deltlist()**”: rimozione dell’elemento in coda alla lista l. La lista l viene privata dell’elemento in coda.

- “l3 = l1.**append**(l2)”: concatenazione di liste. l3 è il risultato della concatenazione, cioé una lista contenente gli elementi della prima lista l1 seguiti (in coda) dagli elementi della seconda lista l2.

- “x=l.**get_comp**(modo)”: mappato in l.gethlist() se modo = head ed in l.gettlist() se modo = tail.

- “l.**put_comp**(modo, variabile)”: mappato in l.inshlist(variabile) se modo = head ed in l.instlist(variabile) se modo = tail.

- “l.**del_comp**(modo)”: mappato in l.delhlist() se modo = head ed in l.deltlist() se modo = tail.

Operazioni sugli stack

- “s.**e_s()**”: s viene istanziato come stack vuoto.

- “b = s.**is_e_s()**”: controlla se lo stack s è vuoto. Restituisce un valore booleano che vale true se lo stack è vuoto, false altrimenti.

- “s.**put_stack**(e)”: inserimento dell’elemento e nello stack s. Il risultato è uguale allo stack s sulla cui cima è stato aggiunto l’elemento e.

- “x = s.**top_stack()**”: recupero dell’elemento in cima allo stack. Il tipo del risultato dipende dal tipo dell’elemento estratto. Lo stack s non viene modificato.

- "s.**pop_stack()**": rimozione della testa dello stack s. s viene privato dell'elemento in cima.
- "x=s.**get_comp**(modo)": mappato in s.top_stack(); il parametro modo è ignorato.
- "componente.**put_comp**(modo, variabile)" : mappato in s.put_stack(variabile); il parametro modo è ignorato.
- "componente.**del_comp**(modo)":mappato in s.pop_stack(); il parametro modo è ignorato.

Operazioni sulle code

- "q.**e_q()**": la coda q viene istanziata con la coda vuota.
- "b = q.**is_e_q()**": controlla se la coda q è vuota. Restituisce un valore booleano che vale true se la coda è vuota, false altrimenti.
- "q.**insqueue**(e)": inserimento dell'elemento e in fondo alla coda q.
- "x = q.**getqueue()**": recupero di un elemento dalla coda q. Il tipo del risultato dipende dall'elemento estratto e la coda q non viene modificata da questa operazione. Rispetto alla operazione di estrazione sugli stack, che estrae l'ultimo elemento inserito, l'operazione sulle code estrae il primo elemento inserito.
- "q.**delqueue()**": rimozione di un elemento dalla coda q.
- "x=q.**get_comp**(modo)": mappato in q.getqueue(); il parametro modo è ignorato.
- "q.**put_comp**(modo, variabile)": mappato in q.insqueue(variabile); il parametro modo è ignorato.
- "q.**del_comp**(modo)": mappato in q.delqueue(); il parametro modo è ignorato.

Operazioni sugli insiemi

- "s.**e_set()**": l'insieme s viene istanziato con l'insieme vuoto.
- "b = s.**is_e_set()**": controlla se l'insieme s è vuoto. restituisce un valore booleano che vale true se l'insieme è vuoto, false altrimenti.

- "s.**insert**(e)": inserimento dell'elemento e nell'insieme s. s è un nuovo insieme contenente tutti gli elementi presenti prima dell'operazione ed anche l'elemento e (se non c'era già).

- "x = s.**getset()**": estrazione non deterministica di un elemento dall'insieme s. Il tipo del risultato x dipende dal tipo degli elementi di s, che non viene modificato dall'operazione.

- "s.**removeset**(e)": rimozione dell'elemento e dall'insieme s. l'insieme s risultante dall'operazione è un insieme contenente tutti gli elementi di s escluso e. Se e non era presente in s, il risultato coinciderà con s prima dell'operazione.

- "s.**delset**()": rimozione non deterministica di un elemento dall'insieme s.

- "x=s.**get_comp**(modo)": mappato in s.getset(); il parametro modo è ignorato.

- "s.**put_comp**(modo, variabile)": mappato in s.insert(variabile); il parametro modo è ignorato.

- "s.**del_comp**(modo)": mappato in s.delset(); il parametro modo viene ignorato.

Operazioni sulle tuple

- "t.**e_t()**": la tupla t viene inizializzata con una tupla vuota.
  Il fatto che la tupla sia vuota significa che in tutte le posizioni della tupla ci sono degli argomenti nulli che possono poi essere sostituiti, utilizzando le operazioni che seguono, con elementi aventi il tipo corretto.

- "b = t.**is_e_t()**": controlla se la tupla t è completamente vuota. restituisce un valore booleano che vale true se la tupla è vuota, false altrimenti.

- "t.**puttupla**(p; e)": scrittura dell'elemento e in posizione p nella tupla t. Il risultato dell'operazione è che nelle posizioni diverse da pos il valore viene mantenuto uguale, l'elemento in posizione p vale e.

- "x = t.**gettupla**(p):" lettura dell'elemento in posizione p nella tupla t. Il tipo del risultato è il tipo dell'elemento letto. Se p è maggiore della dimensione della tupla viene restituito NIL.

- "t.**deltupla**(p):" cancellazione dell'elemento in posizione p nella tupla t. L'elemento cancellato viene posto a NIL.

- "x=t.**get_comp**(modo)": mappato in t.gettupla(modo); il parametro modo deve essere di tipo intero.

- "t.**put_comp**(modo, variabile)": mappato in t.puttupla(modo, variabile); il parametro modo deve essere di tipo intero.

- "t.**del_comp**(modo)": mappato in t.deltupla(modo); il parametro modo deve essere di tipo intero.

6.2.2 Le istruzioni del linguaggio

Per la definizione dell'engine di un architettura, ossia del programma di base di un architettura, HEMASL dispone di una serie di operazioni elementari tra loro collegate tramite i seguenti operatori

- ";" : concatenazione di istruzioni;

- "||" : esecuzione in interleaving di istruzioni;

- "|" : scelta non deterministica di istruzioni;

vengono anche forniti costrutti base per il controllo del flusso di un programma:

- "**if then else endif**" ;

- "**while do endwhile**" ;

- "**case otherwise endcase**"

Le seguenti istruzioni primitive fornite, la cui sintassi formale viene data nell'appendice A, sono le seguenti:

- "**decl** variabile : tipo": dichiarazione di variabile. Al momento della dichiarazione, alla variabile viene associato il valore NIL. Il tipo associato alla variabile può essere un tipo parametrico, un'interfaccia o un qualunque altro tipo legale in HEMASL.

- "Proc_name(a1;...; an)": chiamata di procedura. Le variabili di una chiamata di procedura possono essere passate per valore oppure per riferimento, queste ultime sono contrassegnate dal simbolo ! . Come nei linguaggi imperativi tradizionali, le modifiche del valore associato ad una variabile passata per valore, non sono visibili al di fuori del corpo della procedura, mentre per una variabile passata per riferimento lo sono. Inoltre, quando una variabile viene passata per riferimento in una chiamata di procedura, all'interno della chiamata sono visibili le modifiche a tale variabile apportate dall'esterno della procedura da una eventuale seconda procedura in interleaving con la prima.

- "oggetto.metodo(a1,...an)": chiamata di metodo.

- "variabile := espressione": assegnazione di variabile. Alla variabile viene associato il valore ottenuto dalla valutazione dell'espressione, posto che l'espressione sia di tipo corretto.

- "**skip**": comando nullo. L'esecuzione di questo comando non ha alcun effetto.

- Operazioni sulle componenti dell'architettura:
  le componenti di un architettura, se non maggiormente specificate, sono di tipo componente(T) (ove T è una variabile di tipo dell'oggetto contenuto nel componente), ossia di tipo interfaccia con le seguenti operazioni su di essa stabilite:

  - "componente.**e_comp()**": inizializza il componente vuoto.
  - "b = componente.**is_e_comp()**": controlla se il componente è vuoto. Restituisce un valore booleano che vale true se la componente è vuota, false altrimenti.
  - "x=componente.**get_comp**(modo)": estrazione di un elemento dalla componente. Il modo sara NIL nei casi di stack, queue, set; dovrà essere head o tail per le liste; un valore numerico per le tuple. È quindi palese che modo dovrà essere una variabile di tipo mode. Alla varaiabile x, di tipo T, viene associato un elemento della componente selezionato in funzione del tipo della componente e del modo specificato. Il tipo di x sarà quindi parametrico e coerente col contenuto della componente.

- "componente.**put_comp**(modo, variabile)": inserimento di un elemento nella componente. Il valore associato alla variabile viene inserito nella componente tenendo conto del tipo della componente e del modo specificato. Il modo assumerà gli stessi valori visti sopra.
- "componente.**del_comp**(modo)": cancellazione di un elemento contenuto nella componente. L'elemento viene cancellato in funzione del tipo della componente e del modo specificato. In questo caso il modo sarà NIL per stack, queue; un valore del tipo contenuto per il tipo set; head o tail per le liste e un valore numerico per le tuple.

Le variabili *modo* sono tutte di tipo **mode** un tipo definito ad hoc per essere utilizzato con quest'interfaccia predefinita.
È chiaro che il miglior uso di tale interfaccia predefinita è nei casi di indecisione tra una struttura LIFO o FIFO.

- Operazioni per la ricezione e l'invio dei messaggi:
  - "**send**(destinatario, messaggio)": invio di un messaggio. Il messaggio viene inviato al destinatario inserendo nell'etere del sistema la tripla: (mittente, destinatario, messaggio).
  - "**receive**(mittente, messaggio)": ricezione asincrona di un messaggio. La ricezione asincrona consiste nel prelevare il contenuto di un messaggio, in funzione del mittente, dall'etere del sistema, se questo non è vuoto. Nel caso l'etere sia vuoto l'istruzione associa alla variabile messaggio la costante NIL.
  - "**receive_sync**(mittente, messaggio)": ricezione sincrona di un messaggio. La ricezione sincrona è simile alla precedente, differisce solo nel caso in cui l'etere non contenga alcun messaggio: in tal caso l'agente attende l'inserimento di un messaggio.
  - "**exec**(azione)": esecuzione di una azione. Eseguire una azione consiste nell'inviare un messaggio con contenuto azione all'agente che simula l'ambiente del sistema.

- Operazioni per la percezione e la generazione di eventi:
  - "**percept**(variabile)": percezione di un evento. La percezione istanzia la variabile con l'evento prelevato dall'ambiente del sistema.

- "**put_event**(evento)": inserimento di un evento nell'ambiente di sistema. Questa istruzione viene eseguita dall'agente che simula l'ambiente.
- "**remove_event**(evento)": rimozione di un evento dall'ambiente. Come la precedente, viene eseguita dall'agente che simula l'ambiente.

Il corpo delle procedure utilizzate nell'engine è costituito dagli operatori di concatenazione, interleaving, scelta non deterministica e dalle istruzioni di controllo del flusso di programma applicati alle istruzioni menzionate precedentemente.

6.2.3 Definizione delle architetture

Come già osservato in precedenza l'architettura concreta di un agente è ottenuta come istanza di una architettura astratta. Nella architettura astratta vengono definite le componenti, le possibili procedure che costituiscono l'engine astratto e l'engine astratto stesso. Nell'architettura concreta vengono definite le procedure dell'engine astratto che non sono definite nell'architettura astratta. Queste procedure non vengono definite nell'architettura astratta in quanto dipendono da scelte implementative che non hanno alcuna rilevanza nella descrizione ad alto livello dell'architettura. Ad esempio nella descrizione astratta dell'architettura BDI la semantica informale della procedura get_trigger(p; t) è quella di associare alla variabile t il trigger del piano associato a p. Questa procedura viene definita nell'architettura concreta in quanto bisogna conoscere la struttura dati utilizzata per realizzare il piano, mentre nell'architettura astratta questa informazione manca, non essendo rilevante nel descrivere il funzionamento dell'architettura.

Definizione dell'architettura astratta

La sintassi della definizione di una architettura astratta ha la forma mostrata in figura 6.3
abs_arch_name rappresenta il nome dell'architettura astratta.
extends super_arch_name specifica la relazione di ereditarietà (singola) che l'architettura attualmente specificata ha con un altra architettura astratta definita in precedenza.
type_declaration introduce la definizione delle variabili di tipo, le definizioni di

```
abstract_architecture{arch_name} {extends super_arch_name} {
  type_declaration{
    varType tvname;
    typeDef ty_name:type
  }
  components{
    class comp_name1: type
    agent comp_name2: type
  }
  abstract_procedures{
    proc_k(par_1:int);
  }
  procedures{
    proc_1(par_1:int, par_2:tvname){
      instr1
      ..
      instrn
    }
  }
  engine{
    decl var:abstype;
    instr1
    ..
    proc1(a,b);
    ..
    instrn
  }
}
```

Figura 6.3: Un esempio di definizione di un architettura astratta

tipo, le interfacce utilizzate e i tipi importati dall'esterno.

components introduce la definizione delle componenti dell'architettura astratta:
Il *mode* che precede la dichiarazione dei singoli componenti può assumere i valori class e agent, che significano rispettivamente che la componente deve essere istanziata nella definizione di classe oppure al momento della creazione dell'agente. Le componenti dichiarate in modalità class contengono le informazioni che caratterizzano il programma (ossia il comportamento) dell'agente, mentre le componenti dichiarate in modalità agent contengono quelle relative allo stato.
$comp\_name_1 : type ... comp\_name_{n1}$

sono i nomi ed i tipi delle componenti; se una componente ha tipo non dichiarato significa che tale componente avrà come tipo l'interfaccia predefinata component.

Nel blocco *abstract_procedures* vengono dichiarate le segnature delle procedure utilizzate nelle funzioni sottostanti che ancora non possono essere (o non si vuole che siano) maggiormente specificate.

procedures introduce le definizioni ad alto livello delle procedure che vengono chiamate nell'engine. Per ogni procedura invocata dall'engine o dalle procedure da esso chiamate deve esistere quantomeno un prototipo. La dichiarazione di un prototipo equivale alla dichiarazione di un metodo virtuale che quindi andrà istanziaziato prima dell'utilizzo della classe. Per ogni procedura è possibile fornire indicazioni alternative (eventualmente una o nessuna) su come andrà realizzata:
$proc_1\{...\}...proc_r\{...\}$
i cui corpi sono composti dalle istruzioni descritte nella sezione 5.2.

La definizione dell'*engine* consiste nella definizione delle variabili globali dell'architettura:
$declvar : type$
E del programma che determina il comportamento dell'architettura:
$instr\_1op...proc\_1(a, b)op...instr\_n$
ove op sono gli operatori che contanenano le varie instruzioni.

Esempio

Si consideri una architettura BDI (la cui definizione completa si trova in appendice H.1.1) costituita da tre componenti:

- plans_components la cui istanza rappresenta il programma dell'agente. Nel caso dell'architettura BDI contiene i piani corrispondenti alla conoscenza procedurale dell'agente;
- beliefs_components la cui istanza rappresenta lo stato dell'agente, in questo caso le credenze;
- intentions_components: anche l'istanza di questa componente insieme alla precedente fa parte dello stato dell'agente, in particolare contiene le sue intenzioni.

L'engine dell'architettura ha il compito di:

- percepire un evento;
- selezionare un piano;
- inserire il piano selezionato nelle intenzioni;
- selezionare una intenzione;
- eseguire una istruzione dell'intenzione selezionata.

La componente plans_components viene dichiarata di tipo Components(Plan) in modalità class, ciò significa che verrà istanziata durante la fase di definizione della classe di agente, mentre le componenti beliefs_components, di tipo Components(Belief), e intentions_components, di tipo Components(Intention), sono dichiarate in modalità agent, quindi verranno istanziate nella fase di creazione dell'agente.

Nell'engine dell'architettura astratta bdi vengono dichiarate le quattro variabili globali:

- event;
- selected_event;
- triggered_plans;
- selected_plans.

Le prime due variabili vengono utilizzate per mantenere gli eventi percepiti dall'ambiente del MAS, le altre mantengono informazioni riguardanti rispettivamente i piani utilizzabili "triggered" dall'evento e tra questi ultimi quelli le cui condizioni sono verificate. La dichiarazione di queste quattro variabili nell'engine dell'architettura astratta le rende globali cioè visibili in tutte le procedure dell'engine a qualsiasi livello di annidamento. Le quattro variabili globali non vengono inizializzate in quanto non è ancora conosciuta l'implementazione degli eventi percepiti, della struttura dati che mantiene i piani utilizzabili e di quella che mantiene i piani la cui condizione è verificata. La loro inizializzazione avverrà nella fase di definizione dell'architettura concreta.

La procedura percept_event ha lo scopo di percepire un evento utilizzando la procedura percept_event_from_extern, quindi di inserirlo nella variabile globale event. Al livello di descrizione astratto dell'architettura non si possono fare ipotesi sull'implementazione di event, quindi percept_event_from_extern verrà definita nell'architettura concreta.

La procedura plan_triggering ha lo scopo di istanziare la variabile globale triggered_plans con i piani utilizzabili dall'evento percepito dalla procedura percept_event. In questa procedura vengono dichiarate cinque variabili locali: event_trigger di tipo Trigger, plan_trigger di tipo Trigger, plan di tipo Plan, aux_plans, di tipo stack(Plans), ed empty di tipo booleano, utilizzate per contenere rispettivamente il trigger di un evento, il trigger di un piano, un piano, uno stack di piani ed infine un valore booleano utilizzato come guardia. Le prime tre variabili contengono informazioni non specificate nell'architettura astratta, quindi non vengono inizializzate, aux_plans, invece, è una variabile di servizio il cui utilizzo non caratterizza il comportamento dell'architettura BDI, quindi può essere inizializzata, in questo caso, con uno stack vuoto. La variabile booleana empty viene inizializzata nelle procedure in cui è utilizzata. Per accedere alle variabili ed alle componenti vengono utilizzate procedure quali get_event, get_event_trigger, is_empty_plans_component, get_plans_component ed altre, che verranno definite nella architettura concreta.

La procedura plan_selection ha il compito di valutare le condizioni dei piani utilizzabili, e quindi istanziare la variabile selected_plans con i piani la cui condizione è verificata. Anche in questa procedura vengono dichiarate variabili locali e vengono utilizzate procedure che verranno definite nell'architettura concreta al fine di accedere alle informazioni contenute nelle componenti e manipolare piani, oppure condizioni di piani.

Una volta ottenuto l'insieme dei piani che sono stati utilizzabili e le cui condizioni sono verificate, la procedura upgrade_intentions_component ne sceglie uno e lo inserisce nella componente intentions_component. La scelta del piano da inserire nella componente delle intenzioni viene fatta tramite la procedura select_plan che viene definita nell'architettura concreta. La definizione di select_plan corrisponde alla definizione della strategia con cui viene scelto un piano affinché diventi una intenzione; diverse architetture possono avere diverse strategie per trasformare un piano in una intenzione pur essendo tutte architetture BDI.

La procedura exec_intention() seleziona una intenzione dalla componente intentions_component, quindi seleziona una azione di tale intenzione e la esegue. La procedura get_intentions_component estrae l'intenzione, get_action estrae una azione da tale intenzione e quindi exec_action la esegue.

Definiamo altresìun architettura di tipo reattivo che segue il seguente schema

- si percepisce un evento;
- si controlla quale regole sono attivate;
- si eseguono tutte le regole attivate.

con due componenti:

- un insieme di conoscenze procedurali che utilizza regole della forma evento-azione;
- un database dei fatti conosciuti dall'agente;

tale semplice architettura si rivela particormante adatta per modellare un agente dal comportamento essenzialmente passivo, che fornisce quindi i suoi servizi solo dietro richiesta esplicita da parte di un altro componente del MAS. Un agente di tale tipo si rivela particolarmente adatto per modellare i compiti forniti da un agente facilitatore come quello che fornisce un servizio di pagine gialle.

Definizione dell'architettura concreta

La sintassi della definizione dell'architettura concreta ha la forma mostrata nella figura 6.4.

- Il termine *abs_arch_name* è il nome dell'architettura astratta che viene istanziata ed arch_name è il nome dell'istanza;
- nella sezione *type_definition* viene attribuito un tipo alle variabili di tipo ed alle interfacce; i tipi attribuiti alle interfacce devono ovviamente rispettare l'interfaccia;
- nella sezione components viene attribuito un tipo alle componenti dichiarate nell'architettura *abs_arch_name*;
- nella sezione *init_global_vars* vengono istanziate le variabili globali
  $v1...vn$
  di tipo $type1..typen$ rispettivamente con il valore assegnato dalle espressioni
  $exp1...expn$

```
architecture{arch_name} is a {abs_arch_name} {
  type_definition{
    tvname: type;
    ty_name: type
  }
  components{
    class comp_name1: type
    agent comp_name2: type
  }
  init_global_vars{
    v1:type1 = exp1;
    ..
    vn:typen = expn;
  }
  procedures{
    proc1(par1:int, par2:tvname){
      instr1
      ..
      instrn
    }
  }
}
```

Figura 6.4: Un template per la definizione di un architettura concreta

- nel blocco introdotto da *procedures* vengono implementate le procedure dell'engine dell'architettura astratta che dipendono dall'implementazione, in particolar modo le funzione astratte nell'architettura astratta.

Esempio

Nella definizione dell'architettura concreta vengono fissati i dettagli che dipendono dall'implementazione, quindi vengono associati i tipi alle componenti, vengono fissate le strutture dati che rappresentano l'informazione all'interno delle componenti e le strategie legate alla loro manipolazione.
È ovvio che tali decisioni sono profondamente legate al dominio in cui gli agenti andranno ad operare. Vogliamo usare come esempio un mercato distribuito in cui sono presenti operatori di quattro tipi:

- Venditore : immette sul mercato dei beni con lo scopo di ricavarne denaro. La sua architettura è di tipo BDI, la sua strategia di vendita si basa su di una funzione di prezzo decrescente ed elabora un singolo contratto alla volta.

- Acquirente: ricerca sul mercato dei beni con lo scopo di ricavarne denaro. La sua architettura è di tipo BDI, la sua strategia di acquisto si basa su di una funzione crescente ed elabora un singolo contratto alla volta.

- Commerciante: assume sia il ruolo di acquirente che di venditore.

- Broker: registra la presenza dei venditori e fornisce informazioni agli acquirenti su dove trovare i prodotti cercati; l'architettura è di tipo reactive.

Una specifica del sistema in cui questi agenti operano, scritta in MAS-adl si trova nell'appendice F.

Viene definita l'architettura concreta bdi_1 a partire dall'architettura astratta descritta nell'esempio 6.2.3.

Nella realizzazione dell'architettura bdi alle componenti plans_component e beliefs_component viene associato il tipo list, mentre alla componente intentions_component il tipo set. Quest'ultima scelta permette una selezione non deterministica delle intenzioni da eseguire.

- Gli eventi, le credenze e i messaggi sono modellati tramite lo stesso tipo di dato. Questo perchè gli unici eventi dell'ambiente sono messaggi, e molrte credenze rappresentano messaggi ricevuti o da inviare.Tale tipo di dato è formato da una coppia trigger (etichetta) e da un valore:
  Il valore trigger, definito di tipo stringa, fornisce una descrizione dell'evento manifestatosi nell'ambiente, fornisce un etichetta univoca per le credenze ed esplicita la tipologia del messaggio ricevuto. Il valore è a sua volta una tupla di quattro elementi:
  (string, list of string, integer, real)
  rappresenta l'informazione associata ad un messaggi/evento o ad una credenza. L'utilizzo specifico dei campi varia a seconda dell'etichetta associata all'evento/credenza. Forniamo di seguito alcuni esempi.

  - Rappresentazione dei messaggi/eventi diamo un esempio della rappresentazione di due tipi di messaggio:

  * (find) (sender_Name, [product,list of seller], NIL, NIL) come si puo' vedere il primo elemento della tupla fornisce la classe del messaggio e le informazioni sono contenute nella seconda parte della tupla;
  * (propose)(sender_name, [product], qty, price) in questa tipologia di messaggio (rispondente alla forma specificata nell'ontologia) si vede come tutti i campi siano stati utilizzati per contenere le informazioni rilevanti del messaggio.

- Rappresentazione delle credenze Le credenze sono contraddistinte da un identificatore (unico all'interno dell'insieme delle credenze dell'agente) e dai dati collegati a tale identificatore. Alcuni esempi possono essere:
  * (stock)(NIL, product_list,NIL, NIL): in tale credenza vengono memorizzate le tipologie di merce che sono a disposizione dell'agente per la vendita
  * (sell_step)(NIL, NIL, NIL, percentage): è la percentuale di modifica del prezzo che viene applicata durante le fasi di contrattazione.
  * (sell_start)(NIL, NIL, NIL, percentage): la percentuale del prezzo base che viene utilizzata per stabilire il prezzo di partenza di una contrattazione.
  * (PRODUCT)((sell/buy),[list_of_sellers],qty, price): nella credenza vengono memorizzati i dati relativi ad una singola tipologia di merce venduta/acquistata. Nel primo elemento della tupla viene memorizzato il nome della merce, nel secondo elemento vengono immessi, nell'ordine, se la merce viene venduta o acquistata, l'elenco dei venditori (solo per le merci acquistate), la quantitá disponibile o richiesta ed il prezzo massimo di acquisto (o minimo di vendita).
  * (contract)(contractor,merch,qty,price): in tale credenza vengono memorizzati i dati della contrattazione in corso.
  * (agent_type)(AgentType, NIL, NIL, NIL): questa credenza viene utilizzata per memorizzare la tipologia a cui appartiene l'agente.
  * (mode)((sell/buy)): in questa credenza é contenuta l'informazione

- I piani contenuti nella componente plans_component sono rappresentati da triple il cui primo elemento è il trigger, il secondo la lista delle condizioni da verificare, legate tra loro in and, ed il terzo una sequenza di azioni:
  piano: (trigger, condizioni, sequenza di azioni)
  Il trigger del piano è una stringa di caratteri. Le singola condizione è una coppia di stringhe che rappresenta un'espressione che può essere valutata in funzione del valore associato all'evento percepito ed alle credenze mantenute. In alcuni casi, per ottenere un comportamento maggiormente reattivo, alcuni azioni sono intrprese immediatamente al verificarsi di alcune condizioni. In questo esempio i piani possono avere le seguenti possibili condizioni:

  - (check_broker)(broker): confronta se l'evento proviene dal broker; se la condizione viene verificata vengono immediatamente memorizzati i dati contenuti nel messaggio.
  - (check_channel)(channel): risulta vera se il messaggio proviene dall'agente con cui si e' attualmente in contrattazione o se la credenza "channel" non esiste. Se si verifica questo secondo caso (ed il messaggio e' una proposta) la proposta viene memorizzata nella credenza "contract".
  - (wrong_channel)(channel): controlla se l'evento (messaggio) proviene da un agente diverso da quello con cui si e' in contatto. Se verificata spedisce un messaggio di tipo "busy".
  - (seller_mode)(mode): controlla se il belief indicato ha valore *sell*.
  - (buyer_mode)(mode):controlla se il belief indicato ha valore *buy*.
  - (less_eq_ev_int)(): controlla se il valore intero dell'evento attuale sia minore o uguale di quello della credenza indicata dal nome del materiale dell'evento contrattato nel messaggio che ha generato l'evento stesso.
  - (great_ev_int)(): controlla se il valore intero dell'evento attuale sia maggiore di quello della credenza indicata dal nome del materiale dell'evento contrattato nel messaggio che ha generato l'evento stesso.
  - (less_eq_ev_real)(): controlla se il valore reale dell'evento attuale sia minore o uguale di quello della credenza indicata dal nome del mate-

riale dell'evento contrattato nel messaggio che ha generato l'evento stesso.

- (great_ev_real)(): controlla se il valore reale dell'evento attuale sia maggiore di quello della credenza indicata dal nome del materiale dell'evento contrattato nel messaggio che ha generato l'evento stesso.
- (is_zero_merch)(): vera se il valore intero della credenza indicata dal nome del materiale dell'evento contrattato nel messaggio che ha generato l'evento stesso è uguale a zero.

- La sequenza di azioni viene implementata come una lista di coppie della forma
  action: (actionType,value)
  il cui primo elemento rappresenta il tipo di azione, rappresentato tramite una stringa, e il secondo il valore utilizzato in tale azione, rappresentato come una credenza, la quale assume il significato richiesto dall'azione da intraprendere. Le possibili azioni sono
  - (skip): non effettua azioni.
  - (send)(Belide, NIL, NIL, NIL): spedisce come messaggio la credenza identificata da Belide.
  - (rem_belief)(Belide): rimuove la credenza identificata da Belide.
  - (put_belief)(Belief): inserisce una nuova credenza.
  - (action)(Belide): invia all'ambiente la credenza identificata da Belide.
  - (accept)(): costruisce un messaggio di accettazione del contratto.
  - (refuse)(): costruisce un messaggio per rifiutare il contratto.
  - (ack)(): costruisce un messaggio di conferma (*acknowledge*) del contratto
  - (propose)(): se la proposta contenuta nella credenza "contract" e' accettabile, costruisce un messaggio di proposta.
  - (rem_stock)(): Rimuove la quantitá di materiale venduta dallo stock, e, se il materiale e' finito, spedisce un messaggio di "finished" al broker.

  - (fill_stock)(): Rimuove la quantitá di materiale acquistata dallo stock, e, se non vi è piú richiesta, aggiorna lo stato eliminando il prodotto dalla lista dei prodotti ricercati.
  - (mod_offer_price)(): se la credenza "contract" non è presente nello stato ne costruisce una nuova, altrimenti il contratto preesistente viene modificato secondo il valore del prezzo memorizzato nella credenza buy_step o sell_step.
  - (mod_offer_qty): se la credenza "contract" non è presente nello stato ne costruisce una nuova, altrimenti il contratto preesistente viene modificato modificando la quantitá secondo la quantitá di materiale presente/richiesto nella credenza indicata dal nome del materiale.
  - (next_seller)(): seleziona un nuovo venditore da contattare, costruisce una prima proposta di contratto.
  - (change_mode)(): cambia il comportamento di un agente di tipo Trader da acquirente a fornitore e viceversa.

- Le intenzioni contenute nella componente intentions_component corrispondono alla sequenza di azioni dei piani il cui trigger era in accordo con l'evento percepito e la cui condizione era soddisfatta.

Nell'appendice H.2.1 viene illustrata l'architettura concreta dell'esempio.

La procedura percept_event_from_extern è utilizzata nell'architettura astratta con il fine di associare alla variabile globale event l'evento percepito, in questo caso viene associato ad un messaggio ricevuto dagli altri partecipanti al MAS.

I piani il cui trigger è in accordo con l'evento percepito e le cui condizioni sono soddisfatte, vengono inseriti nella variabile globale selected_plans, e da questa variabile viene selezionato tramite la procedura select_plan, il piano da inserire nella componente intentions_component.

Nell'architettura bdi_1 la variabile selected_plans viene istanziata con una lista di piani, in questo modo viene selezionato il primo elemento della lista:

```
select_plan(plan) {
  plan := gethlist(selected_plans)
}
```

6.2.4 Definizione del M.A.S.

Il MAS consiste in un insieme di agenti che condividono sia l'ambiente ENV in cui sono collocati, che un etere ETH utilizzato nella comunicazione tra agenti.

```
MAS {
  class_agent {arch_name1}{classagent_name1}
    role_playing{Role1; ... ; Rolen} {
    init_comp{

        comp_name1 [elem11 ; : : : ; elem1i ];
         . . .
        comp_namec [elemc1 ; : : : ; elemcj ];
      };
    }
  }
   . . .
  class_agent {arch_name1; classagent_namez}
     {extends classagent_name1} role_playing{Rolez1; ... ; Rolezn} {
    init_comp
     . . .
  };
  init_ENV [event1; : : : ; evente];
  init_ETH [msg1; : : : ; msgm];
  create_agent(typeagent_namet1 ,agent_name1); {
    init_comp comp_name1 [elem11 ; : : : ; elem1i ]; . . .
    init_comp comp_namec [elemc1 ; : : : ; elemcj ];
  }
   . . .
  create_agent(typeagent_nametag , agent_nameag); { . . . }
}
```

Figura 6.5: Un esempio di definizione del MAS

La definizione del MAS (figura 6.5) consiste nel:

- definire la classe degli agenti;
- definire l'inizializzazione dell'ambiente;
- definire l'inizializzazione dell'etere;
- definire la creazione delle istanze delle classi di agenti, cioè gli agenti veri e propri.

La definizione della classe dell'agente consiste nell'istanziazione delle componenti con le informazioni che caratterizzano il programma dell'agente. La classe dell'agente, denotata dalla parola chiave class_agent, è definita da arch_name1 e classagent_name1 che sono, rispettivamente il nome dell'architettura e della classe dell'agente. Se la classe che si stà specificando è sottoclasse di un'altra classe di agente ciò viene descritto tramite
extends class_agent_name
Il significato che assume in tale contesto è che la sottoclasse eredita tutte le definizioni delle componenti della superclasse (e quindi offre gli stessi servizi della superclasse) ma tali definizioni possono essere ampliate o (anche solo parzialmente) riscritte nella sottoclasse.

Il blocco introdotto dalla parola chiave role_playing elenca i ruoli (specificati antecedentemente in MAS-adl) che la classe agente inpersonificherà durante la sua vita tramite la lista
*role*1 ... *rolen*.

Le componenti vengono istanziate, dopo la parola chiave init_comp come
*comp_name*1 ... *comp_namec*,

comp_name1 ... comp_namec sono le componenti, che nella definizione dell'architettura astratta sono dichiarate come class che vengono istanziate rispettivamente con i valori
*elem*11; ... ; *elem*1*i* e *elemc*1; ... ; *elemcj*

Ambiente e etere vengono inizializzati in init_ENV e init_ETH rispettivamente con gli eventi iniziali
*event*1; ... ; *evente*
e con i messaggi iniziali *msg*1; ... ; *msgm*.

Infine le istanze degli agenti vengono generate da create_agent i cui parametri sono il nome della classe di agente istanziata ed il nome dell'istanza. Nella creazione degli agenti vengono anche istanziate le componenti dichiarate come agent.

Esempio

Le definizioni del programma e dello stato dell'agente corrispondono alla definizione del suo comportamento. In questo esempio vengono esaminate due classi di agente relative alle istanze di architettura astratta dell'esempio 6.2.3. Una definizione completa del MAS si trova nell'appendice H.3. La prima classe definisce un agente Seller che si occupa di vendere la mercanzia in suo possesso e il secondo definisce l'agente Broker che si occupa di fornire un servizio di pagine gialle al sistema. Nella definizione delle classi di agenti vengono definite le regole che governano il comportamento della classe stessa. La classe Seller definisce una classe di agenti che agisce nel seguente modo: aspetta un messaggio da parte di un compratore con una prima proposta di contratto; se la proposta è soddisfacente essa viene accettata, notificando tale fatto con un messaggio di "accept"; se la merce richiesta e' esaurita viene inviato un messaggio di refuse; se vi sono ancora margini contrattuali viene inviata all'acquirente una nuova proposta.
La classe Broker possiede un sistema di regole semplicissimo: Se riceve un messaggio di tipo "register" memorizza le informazioni ricevute tra i fatti a sua disposizione; se riceve un messaggio di tipo "finished" aggiorna i fatti del suo stato eliminando dalla lista dei fornitori di una determinata merce il nome dell'agente che ha inviato il messaggio; se infine riceve un messaggio di tipo "search" invia all'acquirente che ha inviato il messaggio una serie di risposte in cui vengono elencati i venditori disponibili per ogni merce richiesta.

Nella fase di creazione degli agenti stessi (ossia all'istanziazione della classe di agente in agente) per i Seller vengono fissati le seguenti credenze: nella credenza identificata da "stock" niene inserita la lista delle merci poste in vendita; Vengono fissate le percentuali di partenza e di contrattazione nelle credenze "sell_start" e "sell_step"; viene fissato il nome, la classe ed il modo di partenza dell'agente ed il nome del broker da contattare; Viene inizializzata ogni credenza riguardante le merci a disposizione ed infine viene aperto un canale con il broker e costruito un messaggio di registrazione.
Il componente che contiene le intenzioni viene inizializzato con l'operazione di invio al broker del messaggio di registrazione contente le merci messe in vendita dall'agente.
Anche in questo caso assai piu' banale l'inizializzazione dell'agente broker durante la quale non viene svolta alcuna operazione.

Parte III

Integrazione di UML in CaseLP

Capitolo 7

Un formato testuale per UML

Per poter trattare facilmente con UML si è reso necessario utilizzare un formato testuale. L'OMG ha proposto un formato definito tramite XMI che viene utilizzato per definire il DTD (*Data Type Definition*) XML dei file utilizzati come mezzo per poter trasferire i dati di una specifica UML da un tool ad un altro. Il difetto principale (per gli scopi di questa tesi) è che si tratta di un linguaggio complesso e di difficile lettura per un utente umano. Per tali ragioni è stato deciso di fornire un semplice linguaggio testuale (chiamato Text-UML) per il ristretto sottoinsieme di UML utilizzato nel proseguo di questo lavoro.
Essenzialmente Text-UML permette di tradurre in formato testuale *user-friendly* Class Diagram e Statechart Diagram semplificati. La sintassi utilizzata, specialmente per la parte statica, ricorda quella di Java o del C++.
la sintassi formale completa del linguaggio si trova nell'appendice E

7.1 Class Diagram

Un class diagram è formato da classifier (in questo caso classi, interfacce, oggetti o template binding) e dalle associazioni tra loro intercorrenti.

7.1.1 Class

Una classe viene mappata in una struttura del tipo illustrato in fig 7.1.

Esaminando in dettaglio

- **dichiarazione del nome della classe**: la definizione di una classe viene introdotta dalla parola chiave *class* seguita dal nome della classe

```
class Class_name
parametric < Parameters >
active(Method) abstract {
  attribute{
    private class Attr_name1 : Type1 = Value1;
    protected instance Attr_name2 : Type2 = Value2;
    public instance Attr_name3 : Type3 = Value3;
  }
  operations{
    private abstract instance ops1() : ret_type1;
    private instance ops2(inout par1:type1) : ret_type2;
  }
}
```

Figura 7.1: Un esempio di definizione di una Classe.

Class_name. Tale nome deve essere unico all'interno della specifica. Questa dichiarazione mappa la dichiarazione del nome della classe del name compartment di una classe UML.

- **dichiarazione di parametri**: se la classe che si sta definendo è una classe parametrica (rappresentata in UML come un box sovrapposto alla classe nell'angolo alto a sinistra), avrà una lista dei suoi parametri racchiusi tra <> introdotta dalla parola chiave *parametric*.
  I parametri possono essere di due tipi
  - parametri di tipo : ossia sono la dichiarazione di una variabile di tipo che deve essere instanziata in seguito per poter rendendere effettiva la classe, un esempio classico è List<T> che verrà istanziata come List<int>, List<char> ...
  - parametri di valore: rappresentano invece dei valori che sono essenziali alla definizione di classe, e che si vuole che siano comuni a tutte le istanze della classe, ma che risulta utile che non siano fissati al momento della definizione della classe. Ad esempio se si realizza la classe Limited_Stack<Limit:int = 10> si potrà utilizzare direttamente la classe Limited_Stack e il limite sarà quello di default. Se si dichiara Limited_Stack< 20 > il limite sarà posto a 20.
- **tipo della classe**

  - la parola chiave *active(Method)* indica che la classe è una classe attiva e che il metodo che viene chiamato alla sua creazione (l'equivalente del metodo main di una classe attiva java) è quello indicato come parametro alla keyword.
  - la parola chiave *abstract* sta ad indicare che la classe che si sta definendo è una classe astratta e che quindi uno o più dei suoi metodi non saranno per il momento implementati.

- **dichiarazione di ereditarietà** *extend : SuperClass_name*

- **dichiarazione di realizzazione** *realize : Interface_Name*

- **lista degli attributi** il blocco preceduto dalla parola chiave *attribute* contiene la lista degli attributi della classe ossia delle variabili proprie della classe in questione. Gli attributi formano lo stato dell'oggetto. Ogni singolo attributo sarà definito tramite le seguenti caratteristiche

  - **visibilità**: la visibilità dell'attributo deve assumere uno dei seguenti valori: public, private o protected.
    - *public*: l'attributo è accessibile dall'esterno della classe
    - *private*: l'attributo è accessibile solo dalla classe e dalle sue eventuali sottoclassi.
    - *protected*: l'attributo è accessibile solo all'interno della classe.
  - **livello di scope**: può assumere i due valori di class o instance
    - *class*: l'attributo è unico per tutti gli oggetti appartenenti alla classe. Ogni modifica si ripercuoterà quindi su tutti gli oggetti appartenenti alla classe in questione.
    - *instance*: l'attributo è di proprietà delle istanze della classe cioé degli oggetti. La modifica di tale attributo rimarrà circoscritta all'oggetto che l'ha fatta.
  - **nome dell'attributo**: nome assegnato all'attributo. Esso viene usato per accedere ai dati in esso contenuti. Se acceduto dall'esterno della classe deve essere utilizzata la *dot notation* ClassName.AttributeName
  - **tipo**: il tipo dell'attributo. Esso può essere un tipo base (integer, string, char, real, boolean), una classe o un tipo parametrico.

  - **valore di default**: valore che viene assunto dall'attributo all'istanziazione di un oggetto della classe.

- **lista delle operazioni** il blocco preceduto dalla parola chiave *operations* contiene la lista delle operazioni della classe ossia dei metodi che permettono la manipolazione dello stato dell'oggetto e implementano i servizi offerti dalla classe in questione. Le operazioni rappresentano quindi il comportamento dell'oggetto. Ogni singolo attributo sarà definito tramite le seguenti caratteristiche

  - **visibilità**: il significato è uguale a quello degli attributi.
  - **livello di scope**: il significato è uguale a quello degli attributi.
  - **nome**: il significato è uguale a quello degli attributi.
  - **lista dei parametri**: le operazioni sono essenzialmente delle funzioni definite sulla classe. Possono necessitare quindi di parametri per poter essere definite. Tali parametri sono definiti tramite
    - **kind** definisce come il parametro viene passato all'interno dell'operazione. Esistono tre modalità possibili
      - *in*: il valore del parametro influenza il risultato dell'operazione. Se ad esso viene assegnato un nuovo valore la modifica ha effetto solo all'interno dell'operazione (passaggio per valore).
      - *out*: il valore del parametro non influenza il risultato dell'operazione. Se ad esso viene assegnato un valore tale modifica ha effetto anche all'esterno dell'operazione (passaggio per riferimento).
      - *inout*: il valore del parametro influenza il risultato dell'operazione. Se ad esso viene assegnato un valore tale modifica ha effetto anche all'esterno dell'operazione (passaggio per riferimento).
    - **nome** : il nome con cui ci si riferisce al parametro all'interno dell'operazione.
    - **tipo** : il tipo del parametro. Essenzialmente uguale ai tipi degli attributi.
    - **valore di default**: valore che assume il parametro se ad esso non viene esplicitamente assegnato nessun altro valore.

  - **tipo di ritorno**: definisce il tipo di ritorno dell'operazione. Se l'operazione non restituisce un valore ed è quindi un procedura, il tipo è void.

Utilizzando una sintassi molto simile è possibile definire delle interfacce le quali sono, ovviamente, prive dei reparti riguardanti l'eventuale realizzazione di interfaccia e della lista degli attributi. Gli altri due tipi di classificatori definibili sono gli oggetti (ossia l'istanziazione di una classe) e l'istanziazione dei parametri di una classe parametrica.

7.1.2 Definizione delle associazioni

Tra le associazioni descritte nel capitolo 2 due sono state trattate in modo differente rispetto alle altre perché sono quelle più usate e per consentire una più rapida comprensibilità della specifica (il colpo d'occhio si perde passando da un disegno ad un testo); queste associazioni sono

- generalization: esprime la nozione del rapporto di sottoclasse. espressa in fase di definizione della classe tramite il blocco *refine SuperClassName*.

- realize: esprime che la classe in questione realizza i servizi dichiarati in un interfaccia. Espresso in fase di definizione della classe dal blocco *realize:Interface_List*.

che sono entrambe esplicitate all'interno della definizione della classe.
Ogni altra associazione viene definita tramite lo schema illustrato in figura 7.2.

```
{Class_A multeplicity(1..1) navigability(out) role(RoleNameA)}
  -(association_name: AssociationClass: kind)->
{Class_B multeplicity(1..*) navigability(in) role(RoleNameB)}.
```

Figura 7.2: Un esempio di definizione di una associazione.

Le associazioni sono quindi composte da un *association end* ad ogni estremità ed un blocco di *association data*.Esaminando in dettaglio

- association end

  - **Classe**: nome della classe a cui l'associazione è associata.

  - **multeplicity**: numero di istanze della classe che possono partecipare all'associazione. Ad esempio un associazione uno a molti avrà dall'estremo di partenza *multeplicity(1..1)* e all'estremo di arrivo *multeplicity(1..*)*.
  - **navigability**: indica se l'estremo dell'associazione è traversabile. Può assumere tre valori
    * **in**: si può traversare solo verso la classe.
    * **out**: si può traversare solo dalla classe.
    * **inout**: si può traversare in entrambi i sensi.

    Ad esempio se la classe A può accedere alle operazioni o agli attributi di B ma non vale il viceversa si troverà *navigability(out)* su A ed *navigability(in)* su B.
  - *Role*: Il ruolo ricoperto dalla classe nell'associazione. Ad esempio in una associazione che modella la relazione "lavora per" un'estremità avrà il ruolo *datore* e l'altra *dipendente*.

- association data
  - **association name**: il nome dato all'associazione.
  - **association Class**: se l'associazione viene modellata tramite una classe questo è il nome della classe che la modella.
  - **kind**: il tipo di associazione utilizzata.
    Le associazioni predefinite sono
    * **use**: indica che la classe A utilizza la classe B. Implica che deve esistere un link tra le due.
    * **instance of**: indica che il classificatore A è un'istanza del classificatore B. Il suo uso più classico è quello di legare un oggetto con la sua classe.
    * **aggregation**: indica che la classe A contiene la B. La relazione di contains non prevede durata di vita uguale tra contenitore e contenuto.
    * **composition**: indica che un'istanza della classe A è composta da elementi della classe B. In questo caso la durata di vita del contenuto e del contenitore sono uguali. Un caso classico è quello degli attributi di una classe.

In caso di necessità altre associazioni potranno essere definite.

È anche possibile raggruppare le classi così definite in Package seguendo lo schema proposto in figura 7.3

```
package: package_name{
public Classname_1;
....
private Classname_n;
}
```

Figura 7.3: Un esempio di definizione di un package.

7.2 Statechart Diagram

Uno statechart diagram è composto da un insieme di macchine a stati le quali, a loro volta, sono composte dagli stati e dalle transizioni intercorrenti tra di essi. Un esempio di definizione di una macccchina a stati viene fornito in figura 7.4.

```
Class_name operation_name(par_1: type1,..,par_n:type_n): ret_type
{
  variables{
    var1:type1=val1
  }
  state start;
  state state1{
    entry/command1;
    do/command2; command3;
  };
  state end;
start-(completion)-> state1;
state1 -(signal{sig1(par1:int)} [par1>0] do/var1:=val2)-> end;
}
```

Figura 7.4: Un esempio di definizione di una state machine.

Una singola macchina a stati definisce un'operazione di una classe.

La definizione di una macchina a stati inizia con lo specificare la classe e l'operazione che viene definita, con la lista dei parametri che vengono passati all'operazione stessa ed il tipo di ritorno dell'operazione. La specifica della macchina vera e propria viene fatta tramite tre blocchi:

- definizione delle variabili;
- definizione degli stati;
- definizione delle transizioni.

7.2.1 Definizione delle variabili

Inizialmente vengono definite le variabili che verranno utilizzate all'interno della macchina. Tali variabili vengono definite nel blocco preceduto dalla parola chiave *variables*.
Esse sono definite tramite

- **nome**: il nome della variabile.
- **tipo**: il tipo assegnato alla variabile. Sono a disposizione gli stessi tipi utilizzabili per la definizione degli attributi di una classe.
- **valore**: il valore assegnato all'inizializzazione della variabile.

7.2.2 Definizione degli stati

A parte i due stati predefiniti per ogni state machine, cioé gli stati start ed end, ogni altro stato viene definito tramite i seguenti elementi:

- **etichetta dello stato**: ogni stato viene etichettato con un'etichetta unica all'interno della macchina. Essa serve per stabilire le transizioni.
- **lista delle transizioni interne**: il comportamento dello stato viene stabilito mediante una o più liste di azioni precedute da un'etichetta.
  - **label**: l'etichetta indica la tipologia delle azioni che seguono ad essa. l'etichetta può essere di tre tipi
    - **entry**: indica una sequenza di azioni non interrompibile da eseguire non appena la macchina entra nello stato.

    * **exit**: indica una sequenza di azioni non interrompibile da eseguire subito prima che la macchina esca dallo stato.
    * **do** : indica che la sequenza di azioni che segué dovra essere svolta finché la macchina rimane nello stato o fino al termine delle operazioni specificate (in tal caso viene generato un evento di tipo *completion*).
  - **comandi**: sono disponibili tutti i comandi di HEMASL cioé send, receive, receive_sync, exec, put_event, remove_event, percept, assegnazione, chiamata di metodo e di procedura

7.2.3 Definizione delle transizioni

Le transizioni stabiliscono il passaggio della state machine da uno stato ad un altro. Esse sono definite da

- **Tipo della transizione**: le transizioni che avvengono all'interno di una macchina a stati possono essere di diversi tipi. In textUML ne vengono forniti tre
  - **semplice**: transizione che va da uno stato ad un altro stato. Indicata con $-()->$.
  - **fork**: transizione che va da uno stato a più stati concorrenti. Indicata tramite $-() =>$.
  - **join**: transizione che proviene da più stati concorrenti e termina in un unico stato. Rappresentata tramite $= ()->$.
- **Stato di partenza**: stato, o elenco di stati nel caso di una transizione *join*, da cui la transizione parte.
- **Stato d'arrivo**: stato, o elenco di stati nel caso di una transizione *fork*, in cui la transizione arriva.
- **Etichettatura della transizione**: una transizione viene etichettata esplicitandone
  - Tipo dell'evento: le transizioni possono essere di tipi diversi a seconda dell'evento che le attiva. I tipi definiti sono
    * **completion**: la transizione viene attivata al completamento delle azioni etichettate con *do/*. Tipica delle computazioni sequenziali.

* **change**: transizione attivata dal fatto che una determinata condizione diventi vera. La condizione è subito dopo indicata nell'etichetta tra parentesi graffe.
* **signal**: ricezione di un segnale specifico dall'esterno. Il segnale viene specificato tramite la sua segnatura, indicata tra parentesi graffe subito dopo l'indicazione del tipo. Il segnale rappresenta una chiamata di metodo alla classe.
* **time**: transizione attivata dopo un periodo di tempo fissato, indicato tra parentesi graffe subito dopo il tipo.

– **Condizione di guardia**: espressione a valori booleani che deve essere verificata prima di attivare la transizione. Se la guardia risulta falsa la transizione non avviene.

– **action**: un eventuale lista di azioni può essere specificata avvenire nel momento in cui la transizione "spara". Le azioni specificate devono essere terminate prima che avvenga qualunque azione dello stato di arrivo.

La scelta di limitarsi ad una traduzione dei due diagrammi sopra esposti è stata motivata dalla necessita di trovarte un sottoinsieme minimale di notazioni che permettesse comunque di poter descrivere gran parte dei sistemi presi in esame. Un vantaggio delle macchine a stati è anche la facilità con cui possono essere simulate tramite un linguaggio logico come Prolog o tramite un linguaggio logico lineare come $\mathcal{E}_{hhf}$. Questa scelta è stata chiaramente fatta pensando a CaseLP come ambiente dove andare ad eseguire le specifiche sopra descritte.

Capitolo 8

Da UML a CaseLP e da CaseLP a UML

Il principale scopo di questa tesi è stato duplice:

- rivedere ed estendere il metodo di specifica di CaseLP;
- proporre un metodo di specifica per agenti basato su UML che seguisse la medologia di lavoro proposta in CaseLP.

Sviluppando i due metodi di specifica in parallelo seguendo una stessa metodologia si è potuta ottenere un integrazione tra i due, permettendo ad uno sviluppatore di scegliere ed utilizzare il linguaggio ritenuto più adatto tra quelli supportati tra CaseLp e UML, o che gli è maggiormente congeniale, in ogni fase della specifica. Perché questa possibilità di scelta sia possibile diviene quindi necessario poter passare da una metodologia ad un altra in modo semplice, possibilmente automatico o semi automatico.

Per poter ottenere il risultato voluto si rivela dunque necessario esaminare innanzitutto come poter tradurre un linguaggio di specifica nell'altro o, ove questo non sia facilmente ottenibile, si deve almeno poter inserire ed integrare parti di una specifica scritte in un linguaggio in una scritta nell'altro.

8.1 Da CaseLP A UML

In questa prima sezione sezione studiamo come una specifica data tramite CaseLP può essere "tradotta" in UML (dotato delle estensioni introdotte in questa tesi). Tale traduzione viene ritenuta possibile per ogni parte della specifica scritta in MAS-adl, nel linguaggio di definizione dei protocolli, ODL ed HEMASL. Passare da una specifica scritta in EHHF o ACLP, che sfruttano le caratteristiche di dichiaritività messe a disposizione da un linguaggio logico, non è purtroppo facilmente ottenibile.

8.1.1 MAS_adl

Le specifiche MAS-adl possono essere tradotte nei Role Diagram proposti in 3.4.
Queste traduzioni sono molto semplici tanto che riteniamo che non sarebbe difficile sviluppare un sistema automatico che consenta di passare da una notazione ad un'altra.

I singoli ruoli presentati in MAS-adl si traducono così come sono nei ruoli del Role Diagram, così come le Agent Class ed i links. Un esempio di questa traduzione si può avere confontando la figura 3.1 con la definizione testuale riportata in F.

8.1.2 Protocolli

Il linguaggio di definizione dei protocolli proposto deriva concettualmente dal linguaggio di definizione dei protocolli proposto da Odell per la FIPA. Anche in questo caso la traduzione da una forma testuale ad una grafica non dovrebbe presentare grossi problemi.

Per presentare un esempio di questa traduzione, mostriamo in forma diagrammatica (figura 8.1) il protocollo di contrattazione proposto nell'appendice F. Come si vede il contenuto semantico è equivalente ed il passaggio da un formato ad un altro segue regole molto semplici, facilmente automatizzabili.

8.1.3 Ontology

Il linguaggio di definizione delle ontologie proposto per il metodo di sviluppo di CaseLP sussume la metodologia ad oggetti, anche se in modo molto ristretto, basti pensare che non sono state forniti gli strumenti per la definizione dei metodi. Questo comunque permette di trasferire una definizione data tramite ODL in un insieme di classi UML in modo semplice, trasformando i concetti in classi e racchiudendo l'ontologia in un package che le contenga.

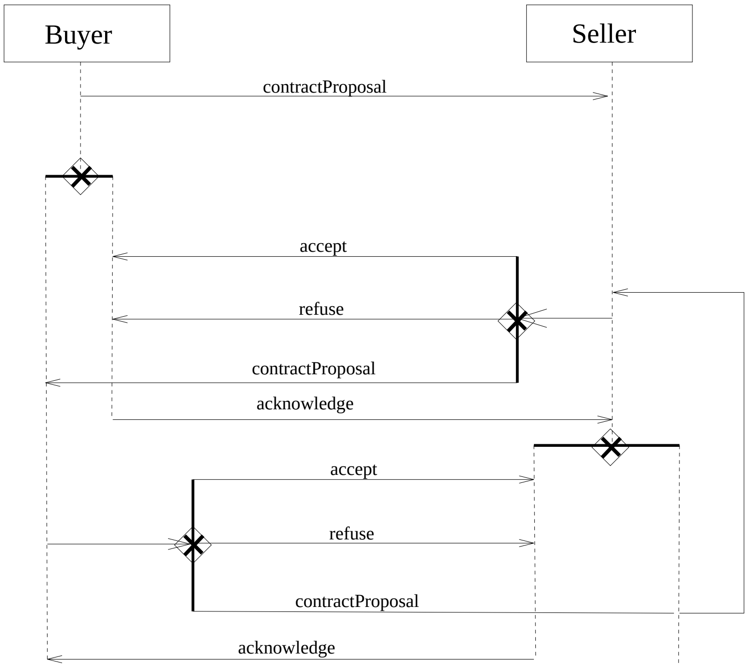

Figura 8.1: Un esempio di Protocol diagram: la fase di contrattazione.

Anche in questo caso mostriamo un esempio di traduzione dall'ontologia presentata nell'esempio in appendice G con il class diagram presentato nella figura 8.2

8.1.4 Da HEMASL a UML

Uno dei lavori principali di questa tesi è stato studiare la possibilità di tradurre delle specifiche HEMASL in UML. La traduzione si è principalmente occupata di passare da una specifica HEMASL ad un Class Diagram ed uno Statechart Diagram.

La scelta del class diagram risulta immediatamente evidente poiché è l'unico strumento messo a disposizione in UML per descrivere la parte statica di una specifica. Per quanto riguarda la scelta dello statechart diagram è necessario fornire le motivazioni della scelta. Un primo fattore che è stato tenuto in considerazione è la capacità espressiva del diagramma: i diagrammi delle interazioni esprimono solo i collegamenti tra le varie classi ed il flusso di informazioni che avviene tra di esse, ma non danno informazioni sul trattamento delle informazioni acquisite; Il flowchart diagram risulta essere, dal punto di

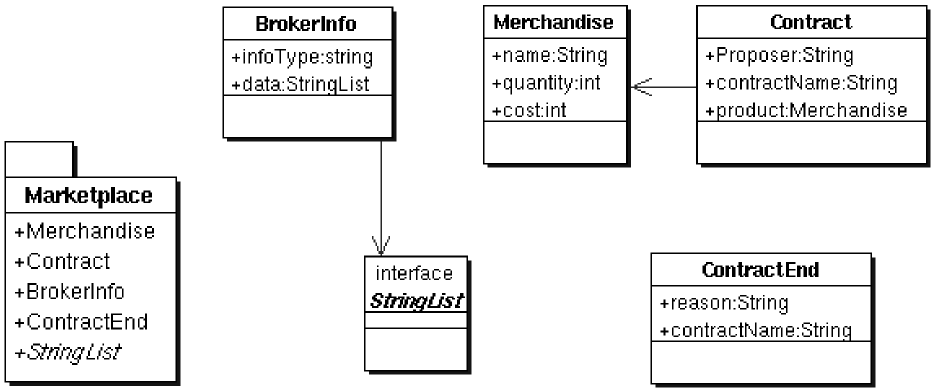

Figura 8.2: L'ontologia come class diagram.

vista della capacità espressiva, un sottoinsieme degli statechart orientato alla modellazione di un attività sequenziale, e poò quindi essere ritenuto sufficiente per una traduzione di HEMASL, ma rappresenterebbe comunque una scelta restrittiva. Per questa ragione quindi la scelta è ricaduta sugli statechart diagram.

Prima di poter procedere a questa traduzione è stato necessario tradurre i costrutti del linguaggio in modo che possano essere il più vicino possibile a costrutti "tipici" dei linguaggi ad oggetti. Si sono quindi definiti:

- un'interfaccia Mailbox con le operazioni necessarie per poter spedire, e ricevere messaggi. L'implementazione sarà poi *language dependent* e verrà fatta solo in fase di realizzazione della specifica;

- le classi parametriche lista, stack, queue, set;

- un'interfaccia component;

- un'interfaccia verso gli attuatori;

- un'interfaccia verso i sensori.

Le tuple devono venire trattate in modo particolare: per ogni definizione di tupla viene definita una classe priva di operazioni con gli attributi pubblici equivalenti ai tipi dichiarati nella tupla, una sorta di "record".

La traduzione avviene in due fasi distinte secondo i diagrammi generati

1. **Class Diagram**
   All'interno del class Diagram si trovano le classi che rappresentano le architetture utilizzate dal MAS, gli agenti e le eventuali classi UML importate dalle architture. È ovvio che le classi importate non verranno tradotte. Ogni singola Meta-classe di specifica (ossia architetture astratte, concrete, agent class ed agenti) seguirà regole particolari per la traduzione.

   - *Abstract architecture.*
     Queste verranno tradotte in classi astratte, attive, parametriche
     I componenti e le variabili globali sono mappati in attributi privati della classe che viene specificata. I componenti di classe avranno scope di classe, quelli di agente scope di istanza. Tutte le variabili globali avranno scope di istanza.
     Le procedure (sia quelle astratte che quelle realizzate) sono mappate in operazioni private della classe (ovviamente le operazioni non ancora definite sono operazioni astratte). Se si pensa ad una specifica orientata al linguaggio Java si rinomina la procedura engine come *main*.
   - *Concrete architecture*
     Sono classi attive ottenute dall'istanziazione dei parametri delle architetture astratte e delle funzioni/metodi precedentemente non definite. La traduzione avverrà quindi in due fasi: quella di istanziazione dei parametri e quella di derivazione in cui saranno implementate le funzioni precedentemente definite come astratte. È necessario che tutte le funzioni astratte dell'architettura astratta siano definite, dato che l'architettura completa deve essere una classe istanziabile. In questa fase diviene necessario adottare una modifica strutturale della specifica per poter effettuare una traduzione: le tuple devono essere mappate in "Records" ossia in classi con solo attributi pubblici, poiché il concetto di tupla non esiste in un linguaggio ad oggetti.
   - *Agent class*
     Le Agent class sono classi attive ottenute istanziando i valori degli attributi di classe delle architetture concrete precedentemente definite.

     Tali attributi solitamente saranno il "programma" della classe di agente ossia quelle componenti che stabiliscono il comportamento comune a tutti gli agenti che ne saranno istanze. Nell'esempio dell'architettura BDI in questa fase verranno istanziate i piani della classe di agente.

- *Agenti*
  Gli agenti saranno infine ottenuti istanziando anche i valori di istanza delle agent class predefinite.

  Tali valori rappresenteranno le conoscenze specifiche dell'agente rigurdo sia il proprio stato che le percezioni che l'agente ha dell'ambiente che lo circonda. Tornando all'esempio dell'architettura BDI in questa fase verranno istanziate le credenze e le intenzioni proprie dell'agente.

Per fornire un esempio in figura 8.3 diamo il diagramma della classe dell'architettura astratta BDI come statechart diagram.

2. **Statechart diagram** Per procedere alla specifica del comportamento delle classi di cui antecedentemente è stata data la visione statica viene costruita una macchina a stati per ogni procedura definita in HEMASL.

   Dato che UML non fissa un linguaggio per la descrizione delle transizioni interne degli stati, la scelta effettuata è stata quella di utilizzare il linguaggio dei comandi di HEMASL, sufficientemente espressivo per gli scopi richiesti.

   La realizzazione della macchina a stati segue il seguente schema:

   - Si generano i due stati di default *start* ed *end* che indicano rispettivamente lo stato di partenza e di fine della macchina a stati definita.
   - Si suddividono le istruzioni di una procedura in blocchi a seconda del flusso del programma. Ad ogni blocco corrisponderà uno stato che avrà al suo interno le istruzioni così isolate.
   - A seconda dell'istruzione di controllo del flusso che ha generato i blocchi si stabiliranno le transizioni che li uniranno. Ad esempio un istruzione del tipo `if (Bool) then command1 else command2 endif command3` genererà tre blocchi contenenti `command1`, `command2` e `command3` e cinque transizioni:
     (a) da start a command1 con guardia Bool;
     (b) da start a command2 con guardia not(Bool);

   (c) da command1 a command3;
   (d) da command2 a command3;
   (e) da command3 a end.

- dato che il programma esaminato è di tipo sequenziale tutte le transizioni che vengono definite sono del tipo completion, ossia verranno attivate dal completamento delle istruzioni contenute nel blocco.

Anche in questo caso si è visto necessario fare ricorso ad alcuni accorgimenti: se ci sono coppie del tipo `endif if`; `endif while`; `endwhile while`; `endwhile if` si inserisce uno stato *fake* tra i due comandi che serve solo per poter definire le transizioni dotate di guardia da tale stato agli stati d'arrivo determinati dalle condizioni derivanti dalla condizione delle istruzioni.

Per chiarificare questa spiegazione viene mostrato in figura 8.4 un estratto dello statechart diagram della architettura astratta BDI.

8.2 Da UML a CaseLP

La traduzione di diagrammi UML in CaseLP non risulta purtoppo effettuabile ad ogni livello come invece avviene nell'altro senso. Esistono alcuni diagrammi che non hanno un corrispettivo in CaseLP sia perché si occupano di problematiche che non rientrano nella definizione di un prototipo (tutti gli implementation diagrams) o perché non fanno parte del suo iter di sviluppo (come ad esempio gli use case diagram, in gran parte sostituiti dai role diagram o gli interaction diagram, utili solo all'interno della specifica di una classe di agente e quindi di minor rilevanza in un sistema ad agenti rispetto a quella che assumono in un sistema ad oggetti). Tutti i nuovi diagrammi introdotti possono comunque essere facilmente tradotti in uno dei linguaggi propri di CaseLP, dato che sono stati ideati o adattati pensando alla metodologia proposta in tale sistema.

Per quanto riguarda quindi role diagram. agent architecture diagram e protocol diagram la loro traduzione segue, in maniera speculare, le indicazioni fornite nella parte precedente sulla traduzione da CaseLP ad UML.

8.2.1 Da UML a HEMASL

HEMASL è un linguaggio pensato per poter definire dei prototipi di MAS in cui molti particolari di dettaglio implementativo possono venir ignorati, e quindi non è un linguaggio di programmazione completo. Esso infatti non fornisce molte istruzioni per la definizione di nuovi tipi di dato e non supporta in alcun modo la definizioni di nuovi oggetti da utilizzarsi all'interno degli agenti che vengono definiti. Si è quindi esteso il linguaggio HEMASL per importare tipi di dato specificati in UML. ottenendo un duplice vantaggio:

1. Da un lato, non è stato necessario estendere HEMASL con nuovi costrutti e tipi di dato, mantendendo così il linguaggio più semplice possibile. L'import di strutture dati definite in UML consente di arricchire HEMASL senza estenderne la sintassi.

2. Dall'altro lato, un utente abituato a usare UML può imparare il minimo indispensabile su HEMASL, ed utilizzarlo senza grandi difficoltà definendo le strutture di cui ha bisogno in UML, e descrivendo in HEMASL solo la parte procedurale.

Per poter effettuare l'inclusione di parti definite in UML all'interno di HEMASL si sono messe a disposizione dello sviluppatore le seguenti utilità:

- durante la dichiarazione di un architettura HEMASL si può effettuare l'importazione di un *package* UML;

- tra i comandi base di HEMASL vi è la chiamata di un metodo di una classe o interfaccia importata;

- come gia fatto notare nel capitolo 6 la sintassi dei comandi di HEMASL ed in special modo la sintassi per l'acesso ai tipi compositi ricalca quella dei linguaggi orientati ad oggetti.

È importante far notare che nessuna assunzione sulla semantica dei metodi invocati di una classe UML importata all'interno di un programma HEMASL viene fatta, ne d'altronde sarebbe fattibile. È quindi completamente responsabilità dello sviluppatore fare si che il programma definito abbia un significato.

8.3 Conclusioni

Sfuttando la traduzione di HEMASL in UML presentata nella sezione precedente si ottiene alla fine una specifica completa in un solo linguaggio di specifica.

La scelta di UML come linguaggio di specifica finale permette di ottenere anche alcuni vantaggi ausiliari:

- La specifica UML è leggibile da un pubblico sicuramente più vasto di una specifica HEMASL, ed essendo in formato grafico è dotata di una maggior immediatezza.

- La specifica in UML può essere inserita in uno strumento CASE ed essere utilizzata per generare automaticamente il codice corrispondente.

- Una specifica UML si presta sicuramente di più di una specifica in HEMASL ad essere ulteriormente raffinata per passare dalla fase prototipale all'implementazione definitiva in un linguaggio ad oggetti. Considerazione importante dato che il linguaggio attualmente più utilizzato per i sistemi ad agenti risulta essere Java.

- Un'ultima ragione e quella che si è dimostrato possibile, almeno su alcuni semplici esempi, costruire un programma Prolag a partire dalla specifica in TextUML di un agente. La potenza espressiva di un linguaggio logico permette di tradurre con facilità le macchine a stati finiti della specifica in clausole logiche, aiutandosi con la segnatura data dai class diagram.

Se questo sistema risulterà funzionare su casi maggiormente complessi sarà possibile ottenere un sistema di prototipazione rapida di un sistema ad agenti (e volendo di un qualunque sistema ad oggetti), a partire da una specifica UML.

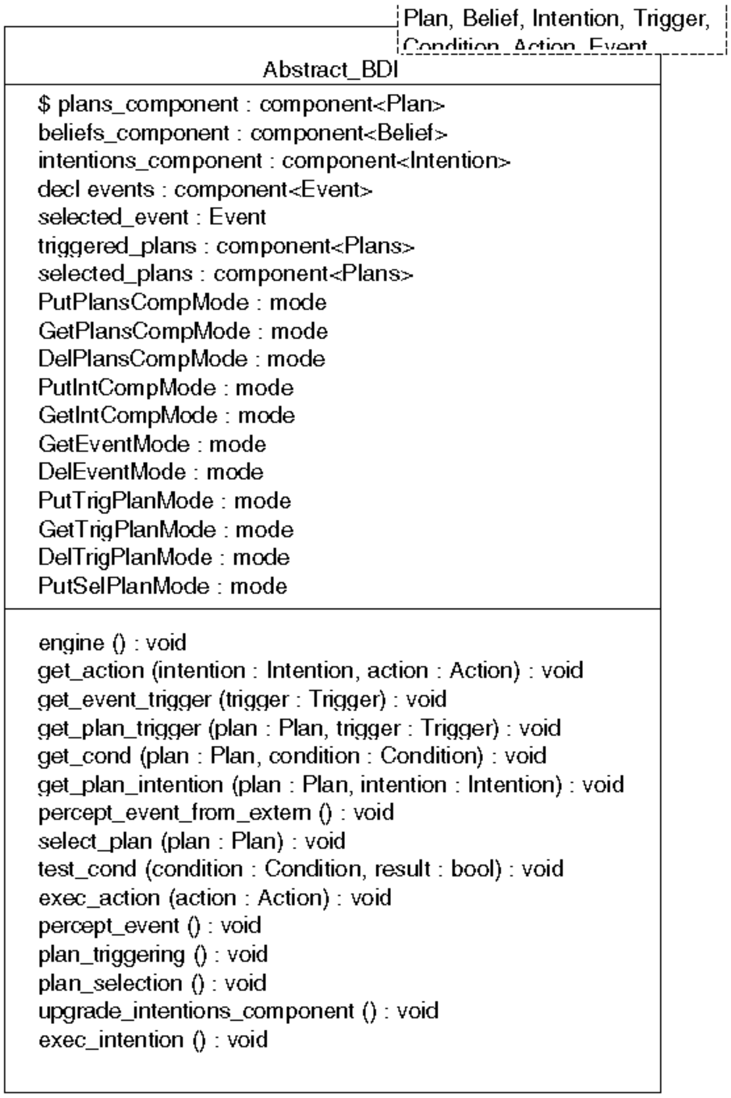

Figura 8.3: La classe architettura astratta BDI.

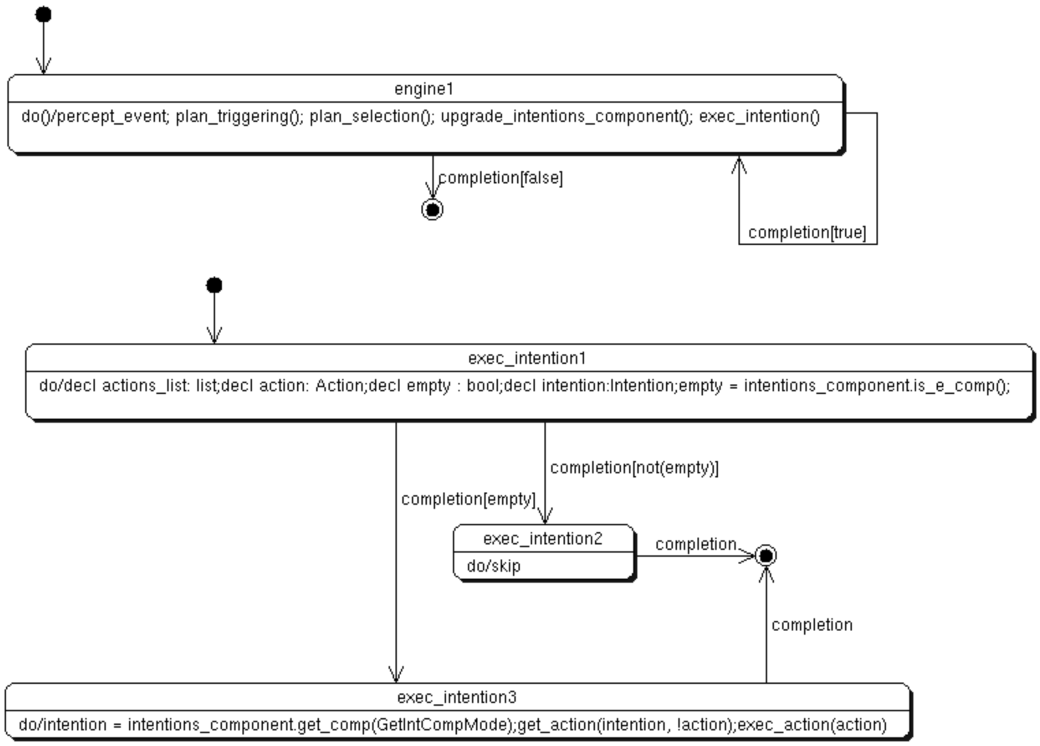

Figura 8.4: Lo statechart diagram dell'architettura astratta BDI.

Conclusioni e sviluppi futuri

Conclusioni

Il lavoro di questa tesi è stato suddiviso in tre parti. Nella prima parte, dopo aver introdotto le nozioni di agente e di sistema ad agenti, si sono isolate le caratteristiche fondamentali che un linguaggio di specifica deve avere per poter essere utilizzato per definire un sistema ad agenti. Sono stati quindi esaminati alcuni linguaggi di specifica esistenti sulla base dei criteri individuati nella fase precedente ed è stato selezionato UML come miglior candidato per supportare la specifica di un sistema ad agenti. Infine è stata proposta un'estensione di UML atta a modellare un sistema ad agenti. Tale estensione, in parte basata su proposte preesistenti, è stata sviluppata tenendo in mente la metodologia di CaseLP ed è composta da tre nuovi diagrammi:

- un diagramma atto a modellare i ruoli che verranno assunti dagli agenti all'interno del sistema (*Role Diagram*);
- un diagramma che si occupa della specifica dei protocolli di comunicazione che verranno adottati (*protocol diagram*);
- un diagramma che si occupa della modellazione architetturale degli agenti che comporranno il MAS Agent Diagram.

Nella seconda parte ci si è occupati di sistemi di supporto per la specifica e la prototipazione di sistemi ad agenti. Si è iniziato offrendo al lettore una breve panoramica di alcuni strumenti attualmente disponibili sul mercato, in particolare una descrizione più estesa è stata fatta di Zeus da cui ci si è ispirati per la modellazione dei ruoli degli agenti. Si è poi proseguito con un esame della metodologia di CaseLP (un metodo di specifica e prototipazione sviluppato presso il DISI) e degli strumenti messi a disposizione da tale metodo. Il metodo stesso è stato rivisto ed ampliato con l'introduzione del concetto di ruolo e, per poter fare ciò sono stati arricchiti i linguaggi forniti dalla metodologia, in particolare MAS-adl (un linguaggio per la specifica architetturale del sistema), il linguaggio per stabilire il modello di conversazione, ODL (un semplice linguaggio di definizione di dati utilizzato per stabilire l'ontologia del dominio in cui il MAS andrà ad operare). Infine è stato esaminato e rivisto HEMASL, uno dei linguaggi forniti dal sistema CaseLP. HEMASL è un linguaggio di specifica per agenti basato sul paradigma imperativo. Esso viene utilizzato per specificare sia le architetture, astratte e concrete, che la loro istanziazione in classi di agenti prima ed in agenti in seguito. HEMASL è stato modificato in un ottica orientata agli oggetti modificandone la sintassi, imponendo un tipaggio delle variabili e delle espressioni di tipo forte ed infine inserendo dei costrutti che permettono di importare classi ed interfacce precedentemente definite in UML.

Nella terza parte, infine, l'attenzione è stata rivolta all'integrazione dei due sistemi di specifica presentati in questa tesi. Si è inizialmente definito un semplice linguaggio (TextUML) per tradurre i diagrammi delle classi e gli statechart diagram di UML in un formato testuale. Lo scopo di questo linguaggio è quello di essere utilizzato all'interno di una specifica UML o mista UML-CaseLP per poter ottenere un prototipo e per fornire la base sintattica per le importazioni di classi in HEMASL. Nell'ultimo capitolo si sono analizzate le relazioni tra i due sistemi di specifica e le possibilità di traduzione ed integrazione di un sistema nell'altro. Il risultato finale è la possibilità di tradurre interamente una specifica fornita con CaseLP in una in UML esteso con i diagrammi presentati in questa tesi. Questa possibilità offre interessanti prospettive sull'utilizzo di strumenti CASE per sviluppare sistemi ad agenti.

Sviluppi futuri

L'ultimo passo per integrare veramente UML in CaseLP è quello di compilare HEMASL in ProlAg, in modo che le specifiche scritte in HEMASL possano essere eseguite. Il lavoro svolto finora si è occupato di testare un semplice traduttore scritto in Prolog da HEMASL in TextUML. I risultati su programmi HEMASL molto semplici sono stati incoraggianti.
Il passo successivo, ossia la traduzione automatica o semi-automatica di un programma in Text-UML in ProlAg si è dimostrata teoricamente possibile, del resto la specifica del comportamento di una classe espressa tramite macchine a stati ben si presta ad essere simulata tramite un linguaggio logico, ed il si-

stema dei mondi multipli è adatto a modellare l'incapsulazione degli oggetti. Purtoppo l'esperienza fatta sull'argomento è sicuramente troppo limitata. Un interessante sviluppo sarebbe arrivare a fornire un traduttore automatico per entrambi questi passi.

Un ulteriore sviluppo interessante sarebbe l'adozione di XML come linguaggio intermedio, abbandonando così TextUML ed ottenendo una specifica UML direttamente fornibile ad uno dei sistemi commerciali utilizzanti UML ed ottenere così un sistema completo di prototipazione e sviluppo per MAS.

Un passo intermedio molto interessante sarebbe una traduzione completa della libreria di architettura fornita in $\mathcal{E}_{hhf}$ in HEMASL ed UML, che svincolerebbe così lo sviluppatore dal dover definire l'architettura dei suoi agenti ma gli consentirebbe di scegliere semplicemente quella più adatta.

Dato il maggior rigore teorico della modellazione tramite logica lineare di concetti come stato e risorse limitate una traduzione della specifica HEMASL-UML in $\mathcal{E}_{hhf}$ invece che in Prolag sarebbe auspicabile e consentirebbe una ancora maggiore integrazione del sistema CaseLP con UML.

L'ultimo passo di sviluppo potrebbe essere la realizzazione di un sistema grafico per l'utilizzo di UML in CaseLP senza passare da una traduzione testuale ma fornendo per ogni passo della specifica la possibilità di utilizzare indifferentemente lo strumento grafico o testuale preferito.

Infine sarebbe utile effettuare un testing del sistema completo specificando un sistema ad agenti completo, realizzandone il prototipo in Prolag, ed arrivando, con l'utilizzo di un tool CASE ad un programma JAVA.

Elenco delle figure

Bibliografia

[age] AgentBuilder. Reticular Systems, Inc. http://www.agentbuilder.com/.

[AR91] E. Astesiano and G. Reggio. Algebraic specification of concurrency. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science, pages 1–39. Springer-Verlag, 1991.

[ATN] K. Futatsugi Ataru T. NAkagawa, T. Sawada. *CafeOBJ User's Manual 1.4*.

[AUM] http://www.auml.org/auml.

[BG95] P. Borba and J. Goguen. Refinement of concurrent object oriented programs. Technical Report PRG-TR-18-95, Oxford University, Computing Laboratory, Programming Research Group, 1995.

[Boo83] G. Booch. Object-oriented design. In P. Freeman and A. I. Wasserman, editors, *Tutorial: Software Design Techniques*, pages 420–436. IEEE Comp. Soc. Press, 1983.

[Del96] G. Delzanno. *Logic & Object-Orinted Programming in Linear Logic*. PhD thesis, Universita of Pisa, Dipartimento di Informatica, December 1996.

[des] The DESIRE Research Programme. Vrije Universiteit. Amsterdam. http://www.cs.vu.nl/vakgroepen/ai/projects/desire/desire.html.

[DKRS91] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. The object-Z specification language - version 1. Technical Report

91-1, Software Verification Research Centre; The University of Queensland, Queensland 4072, May 1991.

[FFMM94] Tim Finin, Rich Fritzon, Don McKay, and Robin McEntire. KQML – A language and protocol for knowledge and information exchange. In *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, pages 126–136, Seatle, WA, July 1994.

[fip] FIPA Home Page. Foundation for Intelligent Physical Agents. http://www.fipa.org/.

[Har87] D. Harel. Statecharts: A visual formalism for complex system. *Science of Computer Programming*, 8(3):231–274, 1987.

[HG96] David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press / ACM Press, 1996.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985. & 0-13-153289-8.

[JKS91] H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: Marriage of actions and objects. In *11th International Conference on Distributed Computing Systems*, pages 142–151, Washington, D.C., USA, May 1991. IEEE Computer Society Press.

[JO99] Bernhard Bauer James Odell, H. Van Dyke Parunak. Response to the omg analysis & design task force uml 2.0 request for information —- extending uml for the specification of agent interaction protocols, 1999.

[JSW98a] 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.

[JSW98b] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[Mar99] S. Marini. Specifica di Sistemi Multi-Agente Eterogenei. Master's thesis, DISI - Università di Genova, Genova, Italy, 1999. In Italian.

[MAS99] 1999. Department of Electrical & Computer Engineering. Air Force Institute of Technology. http://en.afit.af.mil/ai/mase.htm.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[Mil96] D. Miller. Forum: A multiple-conclusion specification logic, 1996.

[MMZ97] M. Martelli, V. Mascardi, and F. Zini. CaseLP: a Complex Application Specification Environment Based on Logic Programming. In *Proc. of ICLP'97 Post Conference Workshop on Logic Programming and Multi–Agents*, pages 35–50, Leuven, Belgium, July 1997.

[NNLC99] Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis. ZEUS: A toolkit and approach for building distributed multi-agent systems. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99)*, pages 360–361, New York, May 1–5 1999. ACM Press.

[RG95] A. S. Rao and M. Georgeff. BDI Agents: from Theory to Practice. In *Proc. of International Conference on Multi Agent Systems (ICMAS'95)*, San Francisco, CA, USA, 1995.

[RP94] G. Reggio and F. Parodi. Metal: a metalanguage for smolcs. Technical Report DISI-TR-94-13, DISI - Università di Genova, 1994.

[Sho91] Y. Shoham. Agent-0: A simple agent language and it's interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence, vol.II*, pages 704–709, Anaheim,CA, 1991.

[sod] Sodabot. MIT Artificial Intelligence Lab. http://www.ai.mit.edu/people/sodabot/sodabot.html.

[Spi87] Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1987.

[Tur] Kennet J. Turner. *The Formal Specification Language LOTOS A Course for Users.*

[UML00] Object Modeling Group. *Unified Modelling Language Specification, version 1.3*, March 2000. OMG document formal/00-03-01. URL: `http://cgi.omg.org/cgi-bin/doc?formal/00-03-01.ps.gz`.

[zeu] The Zeus Agent Building Toolkit. British Telecommunications. http://www.labs.bt.com/projects/agents/zeus/.

Appendice A

Sintassi formale di MAS-adl

La sintassi formale di MAS-ADL espressa come BNF [1] è la seguente:

MasDescription ::= AgentRoleDefList AgentClassDefList AgentInstancesDef
    LinksDefs
AgentRoleDefList ::= AgentRoleDef | AgentRoleDef AgentRoleDefList
AgentRoleDef ::= **role** RoleName "{" RoleDef "}"
RoleName ::= alphanumeric-string
RoleDef ::= [RoleRelations] RequiredAndProvidedServices
RoleRelations ::= [**contains** RoleNameList] | [**refine** RoleName]
RoleNameList ::= RoleName | RoleName RoleNameList
AgentClassDefList ::= AgentClassDef | AgentClassDef AgentClassDefList
AgentClassDef ::= **agentclass** AgentClassName "{" ClassDef "}"
ClassDef ::= **kind** ":" Kind ";" **roles** ":" RoleNameList ";"
    architecture ":" Architecture ";" **interpreters** ":" InterpretersList
Kind ::= **logical** | **interface** | **facilitator** | **manager** | . . .
Architecture ::= **reactive** | **proactive** | **bdi** | . . .
InterpretersList ::= Interpreters | Interpreters InterpretersList
Interpreters ::= alphanumeric-string

[1]note sulla sintassi delle BNF:
::= significa "è definito da"
| significa or
i simboli terminali sono in grassetto o tra doppi apici es."}"
un simbolo tra parentesi quadre ([sim]) è opzionale
un simbolo tra parentesi graffe ({sim}) deve occorrere 1 o piú volte
il simbolo * rappresenta da 0 a N ripetizioni
il simbolo $^+$ rappresenta 1 o piú ripetizioni

RequiredAndProvidedServices ::= [**requires** ServicesList“;”]
[**provides** ServicesList“;”] [**imports** ServicesList“;”]
[**exports** ServicesList“;”]
ServicesList ::= Service | Service ServicesList
Service ::= alphanumeric-string
AgentInstancesDef ::= **agentInstances** AgentInstDefList
AgentInstDefList ::= AgentInstDef | AgentInstDef AgentInstDefList
AgentInstDef ::= NumOfInstances AgentName AgentClassName
NumOfInstances ::= natural-number
LinksDef ::= **link** LinkList
LinkList ::= Link | Link LinkList
Link ::= OneToOneLink | OneToManyLink | ManyToOneLink
OneToOneLink ::= LinkEnd “− >” LinkEnd
OneToManyLink ::= LinkEndList “− >” LinkEnd
ManyToOneLink ::= LinkEnd “− >” LinkEndList
LinkList ::= LinkEnd | LinkEnd LinkEndList
LinkEnd ::= [AgentClass[“.”ID]“.”]RoleName“.”Service

Appendice B

Sintassi formale del linguaggio di definizione dei protocolli

CONV_DEF::= CONVERSATION_LIST RELATIONS
CONVERSATION_LIST ::= CONVERSATION |
    CONVERSATION CONVERSATION_LIST
CONVERSATION ::= **Conversation** CONV_NAME
    “(” ROLE.SERVICE “−− >” ROLE.SERVICE “)” [PARAMETERS]
    MESSAGE_LIST
CONV_NAME ::= alphanumeric_string
ROLE ::= alphanumeric_string
PARAMETERS ::= **Parameters** “(” PARAM_LIST “)”
PARAM_LIST ::= PARAM_NAME | PARAM_NAME “,” PARAM_LIST
PARAM_NAME ::= alphanumeric_string
MESSAGE_LIST ::= MESSAGE | MESSAGE MESSAGE_LIST
MESSAGE ::= ORD“.” [LIFEPATH] “)”
    Message“:” ROLE “−− >” ROLE
    MESSAGE_FORM [CONNECTED_MESSAGE_LIST] |
    CAPACITY
CONNECTED_MESSAGE_LIST ::= CONNECTED_MESSAGE |
    CONNECTED_MESSAGE CONNECTED_MESSAGE_LIST
CONNECTED_MESSAGE ::= CONNECTIVE MESSAGE_FORM
ORD ::= NAT // in ascending order for each lifepath
LIFEPATH ::= alphanumeric_string // created by a switch
MESSAGE_FORM ::= [“[” GUARD_CONDITION “]”]
    Performative “:” PERF; **Content** “:” CONT“;”

[**Goto** (LABEL_NAME)]
GUARD_CONDITION ::= BOOLEAN_EXPRESSION.
CONNECTIVE ::= **AND** | **OR** | **XOR**
CAPACITY ::= **Label** LABEL_NAME |
Switch CONNECTIVE (CON_LIFE_LIST)
CON_LIFE_LIST ::= CON_LIFE | CON_LIFE “,” CON_LIFE_LIST
CON_LIFE ::= CONTENT “– >” LIFEPATH
RELATIONS ::= **Relation** RELATION_LIST
RELATION_LIST ::= RELATION | RELATION RELATION_LIST
RELATION ::= REL_NAME“:” CONV_NAME REL_TYPE
CONV_NAME“;”
REL_NAME ::= alphanumeric_string
REL_TYPE ::= **sc**_ORD“.” [LIFEPATH] | **sc?**_ORD“.” [LIFEPATH]

Appendice C

Sintassi formale di ODL

ONTOLOGY ::= **ontology** ONT_NAME “{” CONCEPT_LIST“}”
CONCEPT_LIST ::= CONCEPT | CONCEPT CONCEPT_LIST
CONCEPT ::= CON_NAME CON_TYPE RELATIONS
    “{” ATTRIBUTE_LIST “}”
RELATIONS ::= [**contains** “{” CON_NAME_LIST “}”] |
    [**refine**: CON_NAME]
CON_NAME_LIST ::= CON_NAME | CON_NAME CON_NAME_LIST
CON_NAME ::= alphanumeric-string.
CON_TYPE ::= **abstract** | **effective**
ATTRIBUTE_LIST ::= ATTRIBUTE | ATTRIBUTE ATTRIBUTE_LIST
ATTRIBUTE ::= NAME : TYPE [“[”CONSTRAINTS“]”];
NAME ::= alphanumeric-string
TYPE ::= **string** | **integer** | **real** | **boolean** | CON_NAME
CONSTRAINTS ::= BEXP
BEXP ::= BEXP CONNECTIVE BEXP | not BEXP | VALUE |
    VALUE ORDER VALUE | **true** | **false**
ORDER ::= “<” | “>” | “=”
CONNECTIVE ::= **and** | **or** | **xor**
VALUE ::= PRIMITIVE_VALUE | CON_NAME“.”NAME
PRIMITIVE_VALUE ::= STRING | INTEGER | REAL

Appendice D

Sintassi formale di Hemasl

NOTE:
Il simbolo “e-s” rappresenta la stringa vuota.
Nella definizione della categoria sintattica PROG il simbolo “skip” rappresenta il programma costituito da nessuna istruzione.
Le stringhe precedute da // rappresentano commenti o vincoli di semantica statica

PROGRAM ::= [IMPORT]“;” ABSTRACT_ARCHITECTURE*;
    CONCRETE_ARCHITECTURE*“;” MAS“.”
IMPORT ::= import “{” UML_PACKAGE* “}”
ABSTRACT_ARCHITECTURE ::= **abstract_architecture**“{”ARCH_NAME“}”
    “{”**extends** ARCH_NAME“}” “{” TYPE_DECL“;” ABS_COMPONENTS“;”
    ABS_PROCS“;” PROCS“;” ENGINE“}” “;”
CONCRETE_ARCHITECTURE ::= **architecture** “{” ARCH_NAME “}”
    is a “{” ARCH_NAME “}” “{” TYPE_DEF“;”
    COMPONENTS“;” INIT_GLOBAL_VARS“;” PROCS “}”
MAS ::= **MAS** “{” AGENT_CLASS*“;” INIT_ENV_ETH“;” AGENTS* “}”
AGENT_CLASS ::= **class_agent** “{” ARCH_NAME “}” “,”
    “{” CLASS_AGENT_NAME “}” “,”
    [{ **extends** CLASS_AGENT_NAME }] {ROLE_PLAYING}
    {INIT_COMPONENTS};
TYPE_DECL ::= **type_declaration** {DECL_TYPE_PAR*};
DECL_TYPE_PAR ::= **varType** T_V_NAME; |
    typeDef TY_NAME : TYPE; |
    INT_NAME: INTERFACE; |
    use IMP_NAME : IMP_TYPE; |

TY_NAME ::= ALPHANUMERIC_STRING
IMP_TYPE ::= ALPHANUMERIC_STRING
T_V_NAME ::= ALPHANUMERIC_STRING
INT_NAME ::= ALPHANUMERIC_STRING
ABS_COMPONENTS ::= **components** {DEF_ABS_COMPS*}
DEF_ABS_COMPS ::= MODED COMP_NAME : ABS_TYPE;
ABS_TYPE ::= **e-s** | PARAM_TYPE
PARAM_TYPE ::= T_V_NAME | TY_NAME | TYPE | INTERFACE | IMP_TYPE
TYPE ::= **int** | **real** | **bool** | **char** | **string** | **mode** | TYPE_CONS | IMP_TYPE
TYPE_CONS ::= **list** (TYPE) | **stack** (TYPE) | **queue** (TYPE) |
    set (TYPE)| **tupla** (NAT, TYPE[,TYPE])
INTERFACE ::= ALPHANUMERIC_STRING //previously defined
MODED ::= **agent**|**class**
ABS_PROCS ::= **abstract_procedures** { ABS_PROC* };
ABS_PROC ::= PROCEDURE_NAME(FORMAL_PARAMS);
PROCS ::= **procedures** {DEF_PROC*}
DEF_PROC ::= PROCEDURE_NAME(FORMAL_PARAMS) {PROG};
FORMAL_PARAMS ::= FORMAL_PARAM [,FORMAL_PARAMS]* |
    EMPTY
FORMAL_PARAM ::= VARS_DECL | ! VARS_DECL
VARS_DECL ::= VARS : PARAM_TYPE
PROG ::= **skip** |
    INSTRUCTION |
    INSTRUCTION ";" PROG |
    INSTRUCTION "||" PROG |
    INSTRUCTION "|" PROG
INSTRUCTION ::= **if** (BOOL) **then** {PROG} **else** {PROG} **endif** |
    case (EXP) CASE_ELEM+ **otherwise** {PROG} **endcase** |
    while (BOOL) **do** {PROG} **endwhile** |
    send(RECEIVER, CONTENT) |
    receive(SENDER, CONTENT) |
    receive_sync(SENDER, CONTENT) |
    exec(CONTENT) |
    put_event(EVENT) |
    remove_event(EVENT) |
    percept(EVENT) |
    decl VARS_DECL |
    VARS **:=** EXP |

PROC_CALL |
COMP_INT |
METHOD_CALLING
CASE_ELEM ::= VALUE : {PROG}
PROC_CALL ::= PROCEDURE_NAME(ACTUAL_PARAMS)
ACTUAL_PARAMS ::= EXP[,EXP]* | **EMPTY**
TYPE_DEF ::= **type_definition** { TYPE_DEFS* }
TYPE_DEFS ::= TY_NAME : TYPE ; |
T_V_NAME : TYPE ; |
INT_NAME : TYPE ; | INTERFACE : TYPE;
ROLE_PLAYING ::= **role_playing**{ROLES}
ROLES ::= ROLENAME [; ROLENAME]
ROLENAME ::= ALPHANUMERIC_STRING
COMPONENTS ::= **components**{DEF_COMP* }
DEF_COMP ::= COMP_NAME : TYPE; // only for undefined components
ENGINE ::= **engine**{DEF_GLOBAL_VAR* PROG};
DEF_GLOBAL_VAR ::= **decl** VARS : ABS_TYPE;
INIT_COMPONENTS ::= {INIT_COMP*};
INIT_COMP ::= **init_comp** {DEF_INIT_COMP*};
DEF_INIT_COMP ::= COMP_NAME : TYPE [VALUE];
INIT_ENV_ETH ::= **init_ENV**[EVENT*]; **init_ETH**[MESSAGGES*]
AGENTS ::= DEF_AGENTS*
DEF_AGENTS ::= **create_agent**(CLASS_AGENT_NAME, AGENT_NAME)
{INIT_COMP*}
INIT_GLOBAL_VARS ::= **init_global_vars**{INIT_VARS*}
INIT_VARS ::= VARS : TYPE **:=** EXP;
EVENT ::= (ALPHANUMERIC_STRING, VALUE)
MESSAGGES ::= (SENDER, RECEIVER, CONTENT)
ARCH_NAME ::= ALPHANUMERIC_STRING
CLASS_AGENT_NAME ::= ALPHANUMERIC_STRING
COMP_NAME ::= ALPHANUMERIC_STRING
SENDER ::= **any** | ALPHANUMERIC_STRING
RECEIVER ::= **all** | ALPHANUMERIC_STRING
CONTENT ::= ALPHANUMERIC_STRING
AGENT_NAME ::= ALPHANUMERIC_STRING
PROCEDURE_NAME ::= ALPHANUMERIC_STRING
VARS ::= ALPHANUMERIC_STRING
VALUE ::= EXP

ALPHANUMERIC_STRING ::= LETTER|[LETTER|DIGIT|SPC_SYM]
LETTER ::= **'a'..'z'** | **'A'..'Z'**
DIGIT ::= **'1'..'9'**
SPC_SYM::= '_'|'-'

********************ESPRESSIONI********************

EXP ::= **NIL** |
        EXP_LIST |
        EXP_STACK |
        EXP_QUEUE |
        EXP_SET |
        EXP_TUPLA |
        BOOL |
        NAT |
        INT |
        REAL |
        STRING |
        VARS |
        EXP_LIST.**gethlist()** |
        EXP_LIST.**gettlist()** |
        EXP_STACK.**top_stack()** |
        EXP_QUEUE.**getqueue()** |
        EXP_SET.**getset()** |
        EXP_TUPLA.**gettupla**(POS, EXP)
EXP_LIST ::= //double linked
        VARS
EXP_STACK ::= //Lifo
        VARS
EXP_QUEUE ::= //fifo
        VARS
EXP_SET ::= //unordered
        VARS
EXP_TUPLA ::= VARS
ORD_EXP ::= NAT | INT | REAL
BOOL ::= B_TERM **and** BOOL |
        B_TERM **or** BOOL |
        not BOOL |
        B_TERM

B_TERM ::= **false** |
 true |
 VARS |
 EXP = EXP |
 ORD_EXP < ORD_EXP |
 ORD_EXP > ORD_EXP
NAT ::= NAT_TERM + NAT | NAT_TERM - NAT | NAT_TERM
NAT_TERM ::= NAT_NUM * NAT_TERM |
 NAT_NUM / NAT_TERM |
 NAT_NUM
NAT_NUM ::= NUMBER | VARS
INT ::= INT_TERM + INT | INT_TERM - INT | INT_TERM
INT_TERM ::= INT_NUM * INT_TERM | INT_NUM / INT_TERM | INT_NUM
INT_NUM ::= -NUMBER | +NUMBER | number | VARS | STRING.**stringToInt**()
REAL ::= REAL_TERM + REAL | REAL_TERM - REAL | REAL_TERM
REAL_TERM ::= REAL_NUM * REAL_TERM |
 REAL_NUM / REAL_TERM |
 REAL_NUM
REAL_NUM ::= -RNUMBER | +RNUMBER | RNUMBER | VARS
RNUMBER ::= NUMBER.NUMBER | NUMBER
STRING ::= VARS|
 STRING.**append**(STRING) |
 STRING.**substring**(NAT,NAT) |
 STRING.**intToString**(INT) |
CHAR ::= charset standard Ascii |
 STRING.**charAt**(NAT)
MODE ::= NAT | **head** | **tail**
MODE_EXP ::= MODE | **NIL** | EXP

********************method Calling********************
METHOD_CALLING ::= CALL_LIST |
 CALL_STACK |
 CALL_QUEUE |
 CALL_SET |
 CALL_TUPLA |
 CALL_STRING |
 IMPORTED_METHOD_CALLS
CALL_STRING ::= STRING.**new**(CHAR+) | STRING.**e_string**()

CALL_LIST ::= EXP_LIST.**e_l**() |
EXP_LIST.**is_e_l**() |
EXP_LIST.**inshlist**(EXP) |
EXP_LIST.**instlist**(EXP) |
EXP_LIST.**delhlist**() |
EXP_LIST.**deltlist**() |
EXP_LIST..**append**(EXP_LIST,EXP_LIST)
CALL_STACK ::= EXP_STACK.**e_s**() |
EXP_STACK.**is_e_s**() |
EXP_STACK.**put_stack**(EXP) |
EXP_STACK.**pop_stack**()
CALL_QUEUE ::= EXP_QUEUE.**e_q**() |
EXP_QUEUE.**is_e_q**() |
EXP_QUEUE.**insqueue**(EXP) |
EXP_QUEUE.**delqueue**()
CALL_SET ::= EXP_SET.**e_set**() |
EXP_SET.**is_e_set**() |
EXP_SET.**insert**(EXP) |
EXP_SET.**removeset**(EXP)
CALL_TUPLA ::= EXP_TUPLA.**e_t**(NAT) |
EXP_TUPLA.**is_e_t**() |
EXP_TUPLA.**puttupla**(POS,EXP) |
EXP_TUPLA.**deltupla**(POS)
COMP_INT::= COMP_NAME.**e_comp**() |
COMP_NAME.**is_e_comp**() |
COMP_NAME.**get_comp**(MODE_EXP) |
COMP_NAME.**put_comp**(MODE_EXP,EXPRESSION) |
COMP_NAME.**del_comp**(MODE_EXP)

Appendice E

Sintassi formale di Text-UML

E.1 Class Diagram

CLASS_DIAGRAM ::= CLASSIFIER* ASSOCIATION*
CLASSIFIER ::= CLASS | INTERFACE | OBJECT | BOUND_ELEMENT
CLASS ::= **class** NAME [**parametric** < PARAM >] [CL_TYPE]
    [**extend**: SUPERCLASS][**realize**:INTERFACE_NAME_LIST]
    "{"ATT_LIST_COMP; OPS_LIST_COMP"}"
INTERFACE ::= [**parametric** < PARAM >] **interface** NAME [CL_TYPE]
    [**extend** SUPERINTERFACE] "{"OPS_LIST_COMP"}"
OBJECT ::= **object** NAME : CLASS "{"[ATTRIBUTE=VALUE*]"}"
PARAM ::= NAME : TYPE = VALUE | NAME
DEFAULT ::= EXPRESSION
NAME ::= A_STRING
CL_TYPE ::= [**Active** (ACTIVE_METHOD)] [GEN_ATTR]
ACTIVE_METHOD::= A_STRING
GEN_ATTR ::= [**abstract**]
ATT_LIST_COMP ::= **attributes** "{" ATTRIBUTE* "}"
VISIBILITY ::= **public** | **private** | **protected**
TYPE ::= PREDEF_TYPE | CLASS_NAME | TYPE_COSTR |
    ENUMERATION
PREDEF_TYPE ::= **int** | **real** | **boolean** | **char** | **string** | **void**
TYPE_COSTR ::= **array** "["NUM[,NUM]"]" ":" TYPE
OPS_LIST_COMP ::= **operations** "{" OPERATION* "}"
ATTRIBUTE ::= VISIBILITY SCOPE NAME ":" TYPE
    ["=" VALUE] PROPERTY

PROPERTY ::= [OCL_FORMULAS]
OPERATION ::= VISIBILITY OP_ATTR NAME “(”FOR_PARA_LIST “)”
    “:” RETURN_TYPE PROPERTY
RETURN_TYPE ::= **void** | TYPE
FOR_PARA_LIST ::= FORMAL_PARAM*
FORMAL_PARAM ::= KIND NAME “:” TYPE “=” DEFAULT_VALUE
KIND ::= **in** | **out** | **inout**
OP_ATTR ::= SCOPE CONCURRENCY GEN_ATTR
SCOPE ::= **class** | **instance**
BOUND_ELEMENT ::= NAME **is** TEMPLATE “<” VALUE_LIST “>”
ENUMERATION ::= **enumeration** NAME “[”VALUES“]”
ASSOCIATION ::= {ASSOCIATION_ENDS} -(ASSOCIATION_DATA)– >
    {ASSOCIATION_ENDS}+ “.”
ASSOCIATION_DATA ::= NAME [:CLASS] “:” KIND
KIND ::= **use** | **instance of** | **aggregation** | **composition**
XOR ::= **xor** “[” ASSOCIATIONS* “]”
ASSOCIATION_RELATIONS ::= [ASSOCIATION] RELATIONS+
RELATIONS ::= “==”RELATION“==>” [ASSOCIATION]
RELATION ::= **AND** | **OR** | **XOR**
ASSOCIATION_END ::= classifier “[” MULTIPLICITY “]” NAVIGABILITY
    ROLENAME
NAVIGABILITY ::= **in** | **out** | **inout**
MULTIPLICITY ::= Num [.. Num|symb]
OCL_FORMULAS ::=
PACKAGE ::= **package**: NAME “{” PACKAGE_ELEMENT_LIST “}”
PACKAGE_ELEMENT_LIST ::= PACKAGE_ELEMENT | PACKAGE_ELEMENT;
    PACKAGE_ELEMENT_LIST
PACKAGE_ELEMENT ::= VISIBILITY CLASS_NAME “:”

E.2 Statechart Diagram

STATEMACHINE ::= CLASS_NAME OPERATION_NAME { VARIABLE_DEF
    STATE* TRANSITION* }
VARIABILE_DEF ::= **variables** { VARIABLE+}
VARIABLE ::= NAME : TYPE = VALUE;
STATE ::= **state** STATE_NAME {INTERNAL_TRANS} | **start** | **end**

INTERNAL_TRANS ::= ACTION_LABEL “/” ACTION_EXP
ACTION_LABEL ::= entry | exit | do | include
ACTION_EXP ::= send | receive | receive_sync | exec | put_event | remove_event
| percept | “:=” | decl | method_calling
EVENT ::= completion | change{BOOLEAN_FORMULA} |
signal{EVENT_NAME(PARLIST)} | time{TIME_EXPRESSION}
TRANSITION ::= **transition** SIMPLE_TRANSITION | TRAN_TO_CONC_STATE
| TRAN_FROM_CONC_STATE
SIMPLE_TRANSITION ::= STATE_NAME −(LABEL)− > STATE_NAME
LABEL ::= EVENT [GUARD] “/” ACTION_EXP
GUARD ::= BOOLEAN_EXPRESSION
TRAN_TO_CONC_STATE ::= STATE −(LABEL) => STATE* //fork
TRAN_FROM_CONC_STATE ::= STATE* = (LABEL)− > STATE //join
OCL_FORMULAS ::=

Appendice F

esempio di specifica in MAS-adl

```
role Seller {
  requires buy_Merchandise; register_Sellers
  provides sell_Merchandise;
  import get_Stock;
  export report_Status;
}
role Buyer {
  requires locate_Sellers, sell_Merchandise;
  provides buy_Merchandise;
  import get_acquire_order;
  export report_Status;
}
role Broker {
  provides register_Sellers, locate_Sellers;
}
agentclass Seller {
  kind: logical;
  roles: Seller;
  architecture: BDI;
}
agentclass Buyer {
  kind: logical;
  roles: Buyer;
  architecture: BDI;
}
```

```
agentclass Trader {
  kind: logical;
  roles: Buyer, Seller;
  architecture: BDI;
}

agentclass Broker {
  kind: facilitator;
  roles: Broker;
  architecture: reactive;
}

agentInstance{
1 seller Seller
1 buyer  Buyer
1 broker Broker
}

link{
Seller.buy_Merchandise  <- Buyer.buy_Merchandise;
Seller.register_Sellers <- Broker.register_Sellers;

Buyer.locate_Sellers   <- Broker.locate_Sellers;
Buyer.sell_Merchandise <- Seller.sell_Merchandise;
}

MESSAGES DEFINITION

Conversation registrate (Seller.register_Sellers <-- Broker.register_Sellers)
1) message: Seller --> Broker
   performative: tell
   content: register(Name,Product_1..Product_n)

Conversation update (Seller.register_Sellers <-- Broker.register_Sellers)
1) message: Seller --> Broker
   performative: tell
   content: finished(Name,Product)
```

```
Conversation locate (Buyer.locate_Sellers <-- Broker.locate_Sellers)
1) message: Buyer --> Broker
   performative: ask
   content: search(Name,Product_1...Product_n)

2) message: Broker --> Buyer
   performative: reply
   content: find(product,Name_1..Name_n)

Conversation acquire (Buyer.buy_Merchandise <-- Seller.buy_Merchandise)

label(start)

1) message: Buyer --> Seller:
   performative: ask;
   content: propose(BuyerName, product, qty, price);

label(seller_reply)

2) message: Seller --> Buyer
   performative: reply
   content: busy(SellerName);
   goto(start)
   XOR
2) message: Seller --> Buyer
   performative: reply
   content: accept(SellerName);
   XOR
2) message: Seller --> Buyer
   performative: reply
   content: refuse(SellerName);
   XOR
2) message: Seller --> Buyer
   performative: reply
   content: propose(SellerName, product, qty, price);
Switch(propose -> a) (accept, refuse -> b)

3.a) message: Buyer --> Seller
```

```
     performative: reply
     content: accept(BuyerName);
     goto(end)
     XOR
3.a) message: Buyer --> Seller
     performative: reply
     content: refuse(BuyerName);
     goto(end)
     XOR
3.a) message: Buyer --> Seller
     performative: ask
     content: propose(BuyerName, product, qty, price);
     goto(seller_reply)


3.b) message: Buyer --> Seller
     performative: tell
     content: acknowledge(BuyerName);

label(end)
4) message: Seller --> Buyer
     performative: tell
     content: acknowledge(SellerName);
```

Appendice G

esempio di definizione di ontologie

```
ontology marketplace{
    merchandise{
      name:string;
      cost:int [cost>0];
      quantity:int [quantity>0 or quantity=0];
    }
    Contract{
       proposer : string [Contract.proposer = AgentName];
       product : Merchandise;
    }
    ContractEnd{
      reason: string [reason = "accept" or reason = "refuse" or
        reason = "acknowledge" or reason = "busy"]
      contractName : string;
    }
    BrokerInfo{
      infoType:string [infoType = "subscribe" or infoType = "query" or
        infoType = "answer" or infoType = "unsubscribe"];
      data: list(string);
    }
}
```

Appendice H

esempio di MAS in HEMASL

H.1 abstract architecture

H.1.1 BDI

```
abstract_architecture {bdi} {
  type_declaration {
    varType Plan;
    varType Belief;
    varType Intention;
    varType Trigger;
    varType Conditions;
    varType Action;
    varType Event;
  }
  components {
    class plans_component: component(Plan);
    agent beliefs_component: component(Belief);
    agent intentions_component: component(Intention);
  };
  abstract_procedures{
    get_action(intention : Intention, !action : Action);
    get_event_trigger(!trigger : Trigger);
    get_plan_trigger(plan : Plan, !trigger : Trigger);
    get_cond(plan : Plan, !conditions : Conditions);
    get_plan_intention(plan : Plan, !intention : Intention);
```

```
    percept_event_from_extern();
    select_plan(!plan : Plan);
    test_cond(conditions : Conditions, !res:bool);
    exec_action(action : Action);
  }
  procedures {
    percept_event() {
      percept_event_from_extern()
    };
    plan_triggering() {
      decl event_trigger: Trigger;
      decl plan_trigger: Trigger;
      decl plan: Plan;
      decl aux_plans: stack(Plans);
      decl empty: bool;

      aux_plans.e_s();
      empty = events.is_e_comp();
      if (empty) then {skip}
      else{
        selected_event = events.get_comp(GetEventMode);
        events.del_comp(DelEventMode);
        get_event_trigger(!event_trigger);
        empty = plans_component.is_e_comp();
        while (not(empty)) do {
          plan = plans_component.get_comp(GetPlansCompMode);
          get_plan_trigger(plan, !plan_trigger);
          if (plan_trigger = event_trigger)
            then {
              triggered_plans.put_comp(PutTrigPlanMode,plan)
             } else {skip}
          endif;
          aux_plans.put_stack(plan);
          plans_component.del_comp(DelPlanCompMode);
          empty = plans_component.is_e_comp();
        } endwhile
        while not(aux_plans.is_e_s()) do {
          plans_component.put_comp(PutPlansCompMode,aux_plans.top_stack());
```

```
        aux_plans.pop_stack();
      } endwhile
    } endif
  };
  plan_selection() {
    decl plan: Plan;
    decl conditions: Conditions;
    decl res: bool;
    decl empty : bool;
    empty = triggered_plans.is_e_comp();
    while (not(empty)) do {
      plan = triggered_plans.get_comp(GetTrigPlansMode);
      get_cond(plan, !conditions);
      test_cond(conditions, !res);
      if (res) then {selected_plans.put_comp(PutSelPlanMode,plan)}
      else {triggered_plans.del_comp(DelTrigPlanMode)}
      endif
    } endwhile
  };
  upgrade_intentions_component() {
    decl plan: Plan;
    decl empty: bool;
    decl intention: Intention;
    empty = selected_plans.is_e_comp();
    if not(empty) then {
      select_plan(!plan);
      get_plan_intention(plan, !intention);
      intentions_component.put_comp(PutIntCompMode, intention)
    } else {skip}
  };
  exec_intention() {
    decl actions_list: list(Action);
    decl action: Action;
    decl empty : bool;
    decl intention:Intention;

    empty = intentions_component.is_e_comp();
    if (empty) then {skip}
```

```
      else {
        intention = intentions_component.get_comp(GetIntCompMode);
        get_action(intention, !action);
        exec_action(action)
      } endif
    };
  engine {
    decl events: component(Event);
    decl selected_event: Event;
    decl triggered_plans: component(Plans);
    decl selected_plans: component(Plans);

    decl PutPlansCompMode: mode;
    decl GetPlansCompMode: mode;
    decl DelPlansCompMode: mode;

    decl PutIntCompMode: mode;
    decl GetIntCompMode: mode;

    decl GetEventMode: mode;
    decl DelEventMode: mode;

    decl PutTrigPlanMode: mode;
    decl GetTrigPlanMode: mode;
    decl DelTrigPlanMode: mode;

    decl PutSelPlanMode: mode;

    while (true) do {
      percept_event();
      plan_triggering();
      plan_selection();
      upgrade_intentions_component();
      exec_intention()
    } endwhile
  }
};
```

H.1.2 reactive

```
abstract_architecture {reactive} {
  type_declaration{
    varType Rule;
    varType Fact;
    varType Event;
    varType Trigger;
    varType Action;
  }
  components{
    facts: component(Fact);
    rules: component(Rule);
  }
  abstract_procedures{
    get_action(rule: Rule, !action : Action);
    get_event_trigger(event : Event, !trigger: Trigger);
    get_rule_trigger(rule: Rule, !trigger: Trigger);
    percept_event_from_extern();
    exec_action(action: Action, event: Event);
  }
  procedures{
    percept_event(){
       percept_event_from_extern();
    };
    rules_triggering(){
      decl event_trigger: Trigger;
      decl rule_trigger: Trigger;
      decl rule: Rule;
      decl aux_rules: stack(Rule);
      decl empty: bool;
      decl selected_event:Event

      aux_rules.e_s();
      empty = events.is_e_comp();
      if (empty) then {skip}
      else {
        selected_event = events.get_comp(GetEventMode);
```

```
      events.del_comp(DelEventMode);
      get_event_trigger(selected_event, !event_trigger);
      empty = rules.is_e_comp();
      while (not(empty)) do {
        rule = rules.get_comp(GetRulesMode);
        get_rule_trigger(rule, !rule_trigger);
        if (rule_trigger = event_trigger) then {
          active_rules.put_comp(PutTrigRuleMode,rule);
          active_event.put_comp(PutActEventMode,selected_event);
        }else {skip}
        endif;
        aux_rules.put_stack(rule);
        rules.del_comp(DelRulesMode);
        empty = rules_component.is_e_comp();
      } endwhile
      while not(aux_rules.is_e_s()) do{
        rules.put_comp(PutRulesMode,aux_rules.top_stack());
        aux_rules.pop_stack();
      } endwhile
    } endif
  };
  exec_action(){
    decl rule: Rule;
    decl action: Action;
    decl empty : bool;

    empty = active_rules.is_e_comp();
    while (not empty) do {
      rule := active_rules.get_comp(GetActRuleMode);
      active_rules.del_comp(GetActRuleMode);
      get_action(rule, !action);
      event := active_event.get_comp(GetActEventMode);
      exec_action(action, event);
      empty = active_rules.is_e_comp();
    }endwhile
  };
}
engine {
```

```
    decl active_rules: component(Rules);
    decl events: component(Event);
    decl active_events:component(Event);
    while (true) do{
      percept_event();
      rules_triggering();
      exec_action();
    } endwhile
  }
}
```

H.2 concrete architecture

H.2.1 BDI

```
architecture {bdi_1} is a {bdi} {
  type_definition{
    varType Plan: tupla(3,Trigger, Conditions, Intention);
      varType Trigger: string;
      varType Conditions: list(Condition);
      varType Intention: list(Action);
    varType Belief: tupla(2,BelIde,BelValue);
    varType Condition: tupla(2,CondType, BelIde);
    varType Action: tupla(2,ActType, ActValue);
    varType Event:Belief;

    Message: Event;
    BelIde: string;
    BelValue: tupla(4,string,list(string), int, real);
    CondType : string;
    ActType : string;
    ActValue: Belief;
  }

  components {
    plans_component: list(Plan);
    beliefs_component: list(Belief);
    intentions_component: set(Intention);
  };
  init_global_vars {
    events: queue(Event) = e_q;
    selected_event = e_t(2);
    triggered_plans: list(Plan) = e_l;
    selected_plans: list(Plan) = e_l;

    PutPlansCompMode = head;
    GetPlansCompMode = head;
    DelPlansCompMode = head;
```

```
    PutIntCompMode = NIL;
    GetIntCompMode = NIL;

    GetEventMode = NIL;
    DelEventMode = NIL;

    PutTrigPlanMode = head;
    GetTrigPlanMode = head;
    DelTrigPlanMode = head;

    PutSelPlanMode = head;
  };

  procedures {

    /* procedure sulla variabile globale events */

    percept_event_from_extern() {
      decl mex: Message;
      receive(All, mex);
      events.insqueue(mex);
    }
    get_event_trigger(!trigger : Trigger) {
      trigger := selected_event.gettupla(1)
    };
    get_event_value(!value:BelValue) {
      value := selected_event.gettupla(2);
    };

    /* procedure sulla variabile globale selected_plans */

    select_plan(!plan : Plan) {
      plan := selected_plans.gethlist()
    };

    /* procedure sui piani */

    get_plan_trigger(plan : Plan, !trigger : Trigger) {
```

```
  plan_trigger := plan.gettupla(1)
};
get_cond(plan : Plan, !conditions : Conditions) {
  conditions := plan.gettupla(2)
};
get_plan_intention(plan : Plan, !intention : Intention) {
  intention := plan.gettupla(3)
};
test_cond(conditions : Conditions, !res:bool) {
  decl condition:Condition;
  decl partRes : bool;

  res:true;
  partRes:true;

  if (conditions.is_e_l()) then {skip}
  else{
    while (not (conditions.is_e_l() and partRes){
      condition := conditions.gethlist();
      test_single_cond(condition, partRes)
      conditions.delhlist();
    } endwhile;
    res := partRes;
  } endif;
};
test_single_cond (condition : Condition, !res:bool) {
  decl auxbel : Belief;
  decl data : BelValue;
  decl aux_data : BelValue;
  decl event_value :ActValue;
  decl auxstr:string;
  if (condition.gettupla(2) = NIL) then
    {skip}
  else {get_belief(condition.gettupla(2), !data);} endif;
  case condition.gettupla(1)
    check_broker: get_event_value(event_value);
                  if (event_value.gettupla(1)=data.gettupla(1))
                  then
```

```
                 {res:=true;
                  strlist:= event_value.gettupla(2);
                  str:=strlist.gethlist();
                  strlist.delhlist();
                  get_belief(str,aux_data);
                  aux_data.puttupla(2,strlist);
                  auxbel.puttupla(1,str);
                  auxbel.puttupla(2,aux_data);
                  put_belief(auxbel);
                 }
               else {res:=false} endif;
wrong_channel: get_event_value(event_value);
                if (data.gettupla(1) = event_value.gettupla(1))
                then
                  {res:=false}
                 else {
                   res:=true;
                   send(event_value.gettupla(1), "busy")
                 } endif;
check_channel: get_event_value(event_value);
                get_event_trigger(auxstr);
                if (auxstr="propose") then {
                  if (data = NIL or
                      data.gettupla(1) = event_value.gettupla(1))
                  then {
                        res:= true,
                        auxbel.puttupla(1,"contract");
                         auxbel.puttupla(2,event_value);
                         put_belief_component(auxbel);
                   } else {res:= false} endif;
                 } else {
                     if (data.gettupla(1)=event_value.gettupla(1))
                     then {res:=true}
                     else {res:=false} endif;
                 }endif;
seller_mode: if (data.gettupla(1) = "sell") then
                  {res:=true}
                 else {
```

```
                     res:=false;
                   } endif;
    buyer_mode:if (data.gettupla(1) = "buy") then
                    {res:=true}
                   else {
                     res:=false;
                   } endif;
    is_zero_merch: get_event_value(event_value);
                   get_belief(eventValue.gettupla(2).gethlist(), aux_data)
                   if (aux_data.gettupla(3) = 0) then
                    {res:=true;}
                   else {res:=false} endif;
    less_eq_ev_int: get_event_value(event_value);
                    get_belief(eventValue.gettupla(2).gethlist(), aux_data)
                    if (event_value.gettupla(3) <= aux_data.gettupla(3))
                    then {res:=true}
                    else {res:=false} endif;
    great_ev_int: get_event_value(event_value);
                  get_belief(eventValue.gettupla(2).gethlist(), aux_data)
                  if (event_value.gettupla(3) > aux_data.gettupla(3)) then
                    {res:=true}
                  else {res:=false} endif;
    less_eq_ev_real: get_event_value(event_value);
                     get_belief(eventValue.gettupla(2).gethlist(), aux_data)
                     if (event_value.gettupla(4) <= aux_data.gettupla(4))
                       then {res:=true}
                       else {res:=false}
                     endif;
    great_ev_real: get_event_value(event_value);
                   get_belief(eventValue.gettupla(2).gethlist(), aux_data)
                   if (event_value.gettupla(4) > aux_data.gettupla(4))
                     then {res:=true}
                     else {res:=false}
                   endif;
  endcase


};
```

```
/* procedure sulla componente belief _component */
get_belief(id: BelIde, !data:Belief) {
  decl bel: Belief;
  decl aux_bel: stack(Belief);
  decl empty: bool;
  aux_bel.e_s();
  empty:= belief_component.is_e_l();
  bel:= belief_component.gethlist();
  while (not(id = bel.gettupla(1)) and not(empty)) do {
    bel:= belief_component.gethlist()
    if (id = bel.gettupla(1)) then {
      data := bel;
      }
    else {
      belief_component.delhlist();
      aux_bel.put_stack(bel);
      empty:= belief_component.is_e_l();
    } endif
  } endwhile;
  if (empty) then {data := NIL } else {skip} endif;
  while (not(aux_bel.is_e_s())) do {
    belief_component.inshlist(top_stack(aux_bel));
    aux_bel.pop_stack();
  }endwhile
};
get_belief_value(id: BelIde, !data:BelValue) {
  decl bel: Belief;
  decl aux_bel: stack(Belief);
  decl empty: bool;
  aux_bel.e_s();
  empty:= belief_component.is_e_l();
  bel:= belief_component.gethlist();
  while (not(id = bel.gettupla(1)) and not(empty)) do {
    bel:= belief_component.gethlist()
    if (id = bel.gettupla(1)) then {
      data := bel.gettupla(2);
      }
    else {
```

```
      belief_component.delhlist();
      aux_bel.put_stack(bel);
      empty:= belief_component.is_e_l();
    } endif
  } endwhile;
  if (empty) then {data := NIL } else {skip} endif;
  while (not(aux_bel.is_e_s())) do {
    belief_component.inshlist(top_stack(aux_bel));
    aux_bel.pop_stack();
  }endwhile
};
remove_belief(id: BelIde) {
  decl bel: Belief;
  decl aux_bel: stack(Belief);
  decl empty:bool;
  aux_bel.e_s();

  empty:= belief_component.is_e_l();
  if (not(empty)) then {
    bel := belief_component.gethlist();
    while (not(id = bel.gettupla(1)) and not(empty)) do {
      aux_bel.put_stack(bel);
      belief_component.delhlist();
      empty:= belief_component.is_e_l();
      if (empty) then {skip}
      else {
        bel := belief_component.gethlist()
      } endif
    } endwhile;
    if empty then {skip}
    else {
      belief_component.delhlist();
    } endif

  } else {skip}
  endif;
  while (not(aux_bel.is_e_s())) do {
    belief_component.inshlist(top_stack(aux_bel));
```

```
    aux_bel.pop_stack();
  } endwhile;
};
put_belief_component(belief: Belief) {
  decl id: BelIde;
  decl bel: Belief;
  decl aux_bel: stack(Belief);
  decl empty:bool;
  aux_bel.e_s();
  id := belief.gettupla(1);
  empty:= belief_component.is_e_l();
  if not(empty) then {
    bel := belief_component.gethlist();
    while (not(id = bel.gettupla(1)) and not(empty)) do {
      aux_bel.put_stack(bel);
      belief_component.delhlist();
      empty:= belief_component.is_e_l();
      if (empty) then {skip}
      else {bel := belief_component.gethlist()}
      endif
    } endwhile;
    if (empty) then {
      belief_component.inshlist(belief)
    } else {
      belief_component.delhlist();
      belief_component.inshlist(belief)
    } endif
  } else {
    belief_component.inshlist(belief)
  } endif;
  while not(aux_bel.is_e_s()) do {
    belief_component.inshlist(top_stack(aux_bel));
    aux_bel.pop_stack();
  } endwhile;
};
get_action(intention : Intention, !action : Action) {
  if not(intention = e_l) then{
    action := intention.gethlist();
```

```
      intentions_component.removeset(intention);
      intention.delhlist();
      intentions_component.insert(intention);
    }
    else {action:= NIL}
    endif
  }
  exec_action(action : Action) {
    decl data: BelValue;
    decl auxData: BelValue;
    decl auxbel: Belief;
    decl receiver: string;
    decl strlist: list(string);
    decl name: string;
    decl content: string;
    decl result: boolean;

    if action = NIL then skip
    else
      case action.gettupla(1)
        send : get_belief(action.gettupla(2).gettupla(1), auxbel);
               if (auxbel = NIL) then {skip;}
               else{
                 get_belief("channel", auxdata);
                 str:= auxdata.gettupla(1);
                 send(str, auxbel);
               } endif;
        skip: skip;
        put_bel: data:= action.gettupla(2);
                 put_belief(data);
        rem_bel: name:= action.gettupla(2).gettupla(1);
                 remove_belief(name);
        action: get_belief(action.gettupla(2).gettupla(1), auxbel);
                if (auxbel = NIL) then {skip;}
                else{exec(auxbel);} endif;
        accept: get_belief("agent_name", data);
                auxbel.e_t();
                auxbel.puttupla(1,"accept");
```

```
        auxdata.e_t();
        auxdata.puttupla(1,data.gettupla(1));
        auxbel.puttupla(2,auxdata);
        put_belief(auxbel);
refuse: get_belief("agent_name", data);
        auxbel.e_t();
        auxbel.puttupla(1,"refuse");
        auxdata.e_t();
        auxdata.puttupla(1,data.gettupla(1));
        auxbel.puttupla(2,auxdata);
        put_belief(auxbel);
ack: get_belief("agent_name", data);
     auxbel.e_t();
     auxbel.puttupla(1,"ack");
     auxdata.e_t();
     auxdata.puttupla(1,data.gettupla(1));
     auxbel.puttupla(2,auxdata);
     put_belief(auxbel);
propose: get_belief("mode", data);
         str := auxdata.gettupla(1);
         get_belief("contract",auxdata);
         get_belief(auxdata.gettupla(2).gethlist(),data);
         if (str="buy") then {
           if (auxdata.gettupla(4) < data.gettupla(4)) then
             {result:=true;}
           else {
             result:=false;
             put_intention_component(next_seller)
           } endif;
         } else {
           if (auxdata.gettupla(4) > data.gettupla(4)) then
             {result:=true;}
           else {result:=false;} endif;
         }
         if (result) then {
           auxbel.puttupla(1,"propose");
           auxbel.puttupla(2,auxdata);
           put_belief(auxbel);
```

```
          } else {
            get_belief("agent_name", data);
            str := data.gettupla(1);
            auxbel.e_t();
            auxbel.puttupla(1,"refuse");
            data.e_t();
            data.puttupla(1,str);
            auxbel.puttupla(2,auxdata);
            send(auxdata.gettupla(1), auxbel)
            rem_belief("propose");
            rem_belief("contract");
          } endif
rem_stock: get_belief("contract", auxdata);
             str:=auxdata.gettupla(2).gethlist();
             get_belief(str, data);
             data.puttupla(3,data.gettupla(3)-auxdata.gettupla(3));
             auxbel.puttupla(1,str);
             auxbel.puttupla(1,data);
             put_belief(auxbel);
             if (data.gettupla(3) = 0) then{
               auxbel.e_t();
               auxbel.puttupla(1,"finished");
               auxdata.e_t();
               strlist.e_l();
               strlist.puthlist(str);
               get_belief("agent",data);
               strlist.puthlist(data.gettupla(1));
               auxdata.puttupla(2,strlist);
               auxbel.puttupla(2,auxdata);
               get_belief("broker",data);
               send(data.gettupla(1),auxbel);
             } else {skip};
fill_stock: get_belief("contract", auxdata);
             str:=auxdata.gettupla(2).gethlist();
             get_belief(str, data);
             data.puttupla(3,data.gettupla(3)-auxdata.gettupla(3));
             auxbel.puttupla(1,str);
             auxbel.puttupla(1,data);
```

```
                  put_belief(auxbel);
                  if (data.gettupla(3) = 0) then{
                    get_belief("need", data);
                    strlist:= data.gettupla(2);
                    remove_string(strlist,str);
                    data.puttupla(2,strlist);
                    auxbel.puttupla(1,"need");
                    auxbel.puttupla(2,data);
                    put_belief(auxbel);
                  } else {skip};
      mod_offer_price: get_belief("mode", data);
                       if (data.gettupla(1) = "sell") then {
                         get_belief("contract", data);
                         if (data = NIL) then{
                           get_belief("propose", data);
                           get_belief("sell_start", data);
                           start:=auxdata.gettupla(4);
                           get_belief(data.gettupla(2).gethstring(),
                                      auxdata);
                           data.puttupla(4,auxdata.gettupla(4)*start);
                         } else{
                           get_belief("sell_step",auxdata)
                           data.puttupla(4,
                             data.gettupla(4)*auxdata.gettupla(4));
                         } endif;
                       } else {
                         get_belief("contract", data);
                         if (data = NIL) then{
                           get_belief("propose", data);
                           get_belief("buy_start", data);
                           start:=auxdata.gettupla(4);
                           get_belief(data.gettupla(2).gethstring(),
                                      auxdata);
                           data.puttupla(4,auxdata.gettupla(4)*start);
                         } else{
                           get_belief("buy_step",auxdata)
                           data.puttupla(4,
                             data.gettupla(4)*auxdata.gettupla(4));
```

```
                  } endif;
                }endif;
                auxbel.puttupla(1,"contract");
                auxbel.puttupla(1,data);
                put_belief(auxbel);
mod_offer_qty: get_belief("contract", data);
               if (data = NIL) then{
                 get_belief("propose", data);
               } else{skip}
               get_belief(data.gettupla(2).gethstring(),auxdata);
               data.puttupla(3, auxdata.gettupla(3));
               auxbel.puttupla(1,"contract");
               auxbel.puttupla(1,data);
               put_belief(auxbel);
next_seller: get_belief("need",data);
             if (data.gettupla(2).is_e_l()) then {
             message to broker;next_seller}
             else{
               str:=data.gettupla(2).gethlist();
               data.gettupla(2).delhlist();
               data.gettupla(2).puttlist(str);
               get_belief(str,data);
               str:=data.gettupla(2).gethlist();
               data.gettupla(2).delhlist();
               build propose
               send(str,propose);
             }
change_mode: get_belief("type", auxdata);
             if (auxdata.gettupla(1) = "trader" then{
               get_belief("mode", auxdata);
               if (auxdata.gettupla(1) = "buy") then
                 {get_belief("stock", data);
                  if data.gettupla(2) = NIL then {skip}
                  else {
                    auxbel.e_t();
                    auxdata.puttupla(1,"sell");
                    auxbel.puttupla(1,"mode");
                    auxbel.puttupla(2,auxdata);
```

```
                                      put_belief(auxbel);
                                    } endif;
                                    }
                                 }else {
                                   get_belief("need", data);
                                    if data.gettupla(2) = NIL then {skip}
                                    else {
                                      auxbel.e_t();
                                      auxdata.puttupla(1,"buy");
                                      auxbel.puttupla(1,"mode");
                                      auxbel.puttupla(2,auxdata);
                                      put_belief(auxbel);
                                      put_intention_component(next_seller);
                                    } endif;
                                 } endif;
                               } else {skip}
        endcase
      endif
    }
  }
};
```

H.2.2 Reactive

```
architecture {react_1} is a {reactive} {
  type_definition{
    varType Fact: tupla(2,FactIde,Value);
    varType Rule: tupla(2,Trigger,Action);
    varType Trigger: string;
    varType Event: tupla(2,Trigger,evValue);
    varType Action: list(string);
    Value: list(string);
    FactIde: string;
    EvValue: tupla(4,string,list(string), int, real);
    Message: Event;
  }
  components {
    facts: list(Fact);
    rules: list(Rule);
  };
  init_global_vars {
    decl active_rules: queue(Rules);
    decl events: queue(Event);
    decl active_events: queue(Event);
  };
  procedures {
    percept_event_from_extern() {
      decl mex: Message;
      receive(All, mex);
      events.insqueue(mex);
    };
    get_action(rule : Rule, !action : Action){
      action := rule.gettupla(2);
    };
    get_rule_trigger(rule : Rule, !trigger : Trigger){
      trigger := rule.gettupla(1);
    };
    get_event_trigger(event: Event, !trigger : Trigger){
      trigger = event.gettupla(1);
    };
```

```
get_event_value()(event: Event, !value : EvValue){
  value:= event.gettupla(2);
}
insert_fact(fact:Fact){
  decl aux_fact : Fact;
  decl aux_facts : stack(Fact);
  aux_facts.e_s();
  if (facts.is_e_l()) then { skip}
  else {
    while(not(facts.is_e_l())) do {
      aux_fact:= facts.gethlist();
      if (aux_fact.gettupla(1) = fact.gettupla(1)) then {skip}
      else{aux_facts.put_stack(aux_fact)} endif;
      facts.delhlist();
    }endwhile;
    while(not(aux_facts.is_e_s())) do {
      aux_fact:= facts.top_stack();
      facts.puthlist(aux_fact);
      facts.pop_stack();
    }endwhile;
  } endif;
  facts.puthlist(fact);
};
remove_fact(factide:FactIde){
  decl aux_fact : Fact;
  decl aux_facts : stack(Fact);
  aux_facts.e_s();
  if (facts.is_e_l()) then {skip}
  else {
    while(not(facts.is_e_l())) do {
      aux_fact:= facts.gethlist();
      if (aux_fact.gettupla(1) = factide) then {skip}
      else{aux_facts.put_stack(aux_fact)} endif;
      facts.delhlist();
    }endwhile;
    while(not(aux_facts.is_e_s())) do {
      aux_fact:= facts.top_stack();
      facts.puthlist(aux_fact);
```

```
      facts.pop_stack();
    }endwhile;
  } endif;
};
get_fact(factide:FactIde; !fact:Fact){
  decl aux_fact : Fact;
  decl aux_facts : stack(Fact);
  aux_facts.e_s();
  fact:=NIL;
  if (facts.is_e_l()) then {skip}
  else {
    while(not(facts.is_e_l())) do {
      aux_fact:= facts.gethlist();
      if (aux_fact.gettupla(1) = factide) then {fact:= aux_fact}
      else{skip} endif;
      aux_facts.put_stack(aux_fact);
      facts.delhlist();
    }endwhile;
    while(not(aux_facts.is_e_s())) do {
      aux_fact:= facts.top_stack();
      facts.puthlist(aux_fact);
      facts.pop_stack();
    }endwhile;
  } endif;
};
exec_action(action:Action, event:Event){
  while (not(action.is_e_l()) do {
    exec_single_action(action.gethlist(), event);
    action.delhlist();
  } endwhile;
}
exec_single_action(action:string, event:Event){
  case
    put_fact: aux_fact.e_t();
      aux_fact.puttupla(1,event.gettupla(2).gettupla(1));
      aux_fact.puttupla(2,event.gettupla(2).gettupla(2));
      insert_fact(aux_fact);
    remove_fact:
```

```
  remove_fact(event.gettupla(2).gettupla(1));
send: aux_mex.e_t();
  get_fact(event.gettupla(2).gettupla(1));
  aux_mex.puttupla(1,);
  aux_mex.puttupla(2,);
  send(event.gettupla(2).gettupla(2).gethlist(),aux_mex);
register: strlist:= event.gettupla(2).gettupla(2);
  aux_str:=event.gettupla(2).gettupla(1);
  while (not(strlist.is_e_s())) do {
    str:= strlist.gethlist();
    strlist.delhlist();
    get_fact(str,auxfact);
    auxfact.puttupla(2, auxfact.gettupla(2).instlist(aux_str));
    insert_fact(auxfact);
  } endwhile;
finished:
  aux_str:=event.gettupla(2).gettupla(2).gethlist();
  get_fact(aux_str,aux_fact);
  strlist:= aux_fact.gettupla(2);
  aux_str:=event.gettupla(2).gettupla(1);
  while (not(strlist.is_e_s())) do {
    if (strlist.gethlist()=auxstr) then {skip}
    else{aux_strlist.puthlist(strlist.gethlist())} endif;
    strlist.delhlist();
  }
  auxfact.puttupla(2,aux_strlist);
  put_fact(aux_fact);
query:
  aux_str:=event.gettupla(2).gettupla(2);
  while (not aux_str.is_e_l()) do {
    get_fact(aux_str.gethlist(), aux_fact);
    if (aux_fact.gettupla(2).is_e_l()){skip}
    else{
      aux_mex.puttupla(1,"find");
      aux_mex.puttupla(2, aux_str.gethlist());
      aux_mex.puttupla(2,aux_fact.gettupla(2));
      send(event.gettupla(2).gettupla(1),aux_mex);
    }endif;
```

```
          aux_str.delhlist();
        }endwhile;
      }
    };
  }
};
```

H.3 MAS

```
MAS {
  class_agent {bdi_1}{Buyer}
   role_playing{Buyer} {
   init_comp{
     plans_component:[
       (find)[(check_broker)(broker)][(next_seller)(NIL)],
       (propose)[(wrong_channel)(channel)][(skip)(NIL)],
       (propose)[(buyer_mode)(mode), (check_channel)(channel),
                 (less_eq_ev_int)(NIL), (less_eq_ev_real)(NIL)]
                 [(accept)(NIL), (send)(accept,NIL,NIL,NIL)],
       (propose)[(buyer_mode)(mode), (check_channel)(channel),
                 (great_ev_int)(NIL), (less_eq_ev_real)(NIL)]
                 [(mod_offer_qty)(NIL), (propose)(NIL),
                  (send)(propose,NIL,NIL,NIL)],
       (propose)[(buyer_mode)(mode), (check_channel)(channel),
                 (less_eq_ev_int)(NIL), (great_ev_real)(NIL)]
                 [(mod_offer_price)(NIL), (propose)(NIL),
                  (send)(propose,NIL,NIL,NIL)],
       (propose)[(buyer_mode)(mode), (check_channel)(channel),
                 (great_ev_int)(NIL), (great_ev_real)(NIL)]
                 [(mod_offer_qty)(NIL), (mod_offer_price)(NIL),
                  (propose)(NIL), (send)(propose,NIL,NIL,NIL)],
       (propose)[(buyer_mode)(mode), (check_channel)(channel),
                 (is_zero_merch)(NIL)]
                 [(refuse)(NIL), (send)(refuse),
                  (remove_belief)(contract,NIL,NIL,NIL),
                  (remove_belief)(channel,NIL,NIL,NIL)],
       (accept)[(wrong_channel)(channel)][(skip)(NIL)],
       (accept)[(buyer_mode)(mode), (check_channel)(channel)]
               [(fill_stock)(NIL), (ack)(NIL), (send)(ack,NIL,NIL,NIL)
                (remove_belief)(contract,NIL,NIL,NIL),
                (remove_belief)(channel,NIL,NIL,NIL)
                (next_seller)(NIL)],
       (ack)[(wrong_channel)(channel)][(skip)(NIL)],
       (ack)[(buyer_mode)(mode), (check_channel)(channel)]
            [(fill_stock)(NIL), (remove_belief)(contract,NIL,NIL,NIL),
```

```
                (remove_belief)(channel,NIL,NIL,NIL)
                (next_seller)(NIL)],
          (refuse)[(wrong_channel)(channel)][(skip)(NIL)],
          (refuse)[(buyer_mode)(mode), (check_channel)(channel)]
                  [(remove_belief)(contract,NIL,NIL,NIL),
                   (remove_belief)(channel,NIL,NIL,NIL)
                   (next_seller)(NIL)],
          (busy)[(wrong_channel)(channel)][(skip)(NIL)],
          (busy)[(buyer_mode)(mode), (check_channel)(channel)]
                [(remove_belief)(contract,NIL,NIL,NIL),
                 (remove_belief)(channel,NIL,NIL,NIL)
                 (next_seller)(NIL)],
    };
    }
  class_agent {bdi_1}{Seller}
    role_playing{Seller} {
    init_comp{
      plans_component:[
        (propose)[(wrong_channel)(channel)][(skip)(NIL)],
        (propose)[(seller_mode)(mode), (check_channel)(channel),
                  (less_eq_ev_int)(NIL), (great_ev_real)(NIL)]
                  [(accept)(NIL), (send)(accept,NIL,NIL,NIL)],
        (propose)[(seller_mode)(mode), (check_channel)(channel),
                  (great_ev_int)(NIL), (great_ev_real)(NIL)]
                  [(mod_offer_qty)(NIL), (propose)(NIL),
                   (send)(propose,NIL,NIL,NIL)],
        (propose)[(seller_mode)(mode), (check_channel)(channel),
                  (less_eq_ev_int)(NIL), (less_eq_ev_real)(NIL)]
                  [(mod_offer_price)(NIL), (propose)(NIL),
                   (send)(propose,NIL,NIL,NIL)],
        (propose)[(seller_mode)(mode), (check_channel)(channel),
                  (great_ev_int)(NIL), (less_eq_ev_real)(NIL)]
                  [(mod_offer_qty)(NIL), (mod_offer_price)(NIL),
                   (propose)(NIL), (send)(propose,NIL,NIL,NIL)],
        (propose)[(seller_mode)(mode), (check_channel)(channel),
                  (is_zero_merch)(NIL)]
                  [(refuse)(NIL), (send)(refuse),
                   (remove_belief)(contract,NIL,NIL,NIL),
```

```
                  (remove_belief)(channel,NIL,NIL,NIL)],
      (accept)[(wrong_channel)(channel)][(skip)(NIL)],
      (accept)[(seller_mode)(mode), (check_channel)(channel)]
              [(less_stock)(NIL), (ack)(NIL), (send)(ack,NIL,NIL,NIL)
               (remove_belief)(contract,NIL,NIL,NIL),
               (remove_belief)(channel,NIL,NIL,NIL)],
      (ack)[(wrong_channel)(channel)][(skip)(NIL)],
      (ack)[(seller_mode)(mode), (check_channel)(channel)]
           [(less_stock)(NIL), (remove_belief)(contract,NIL,NIL,NIL),
            (remove_belief)(channel,NIL,NIL,NIL)],
      (refuse)[(wrong_channel)(channel)][(skip)(NIL)],
      (refuse)[(seller_mode)(mode), (check_channel)(channel)]
              [(remove_belief)(contract,NIL,NIL,NIL),
               (remove_belief)(channel,NIL,NIL,NIL)],
    ];
  };
  }
class_agent {bdi_1}{Trader}{extends Buyer}
  role_playing{Seller; Buyer} {
  init_comp{
    plans_component:[
      (propose)[(wrong_channel)(channel)][(skip)(NIL)],
      (propose)[(seller_mode)(mode), (check_channel)(channel),
                (less_eq_ev_int)(NIL), (great_ev_real)(NIL)]
                [(accept)(NIL), (send)(accept,NIL,NIL,NIL)],
      (propose)[(seller_mode)(mode), (check_channel)(channel),
                (great_ev_int)(NIL), (great_ev_real)(NIL)]
                [(mod_offer_qty)(NIL), (propose)(NIL),
                 (send)(propose,NIL,NIL,NIL)],
      (propose)[(seller_mode)(mode), (check_channel)(channel),
                (less_eq_ev_int)(NIL), (less_eq_ev_real)(NIL)]
                [(mod_offer_price)(NIL), (propose)(NIL),
                 (send)(propose,NIL,NIL,NIL)],
      (propose)[(seller_mode)(mode), (check_channel)(channel),
                (great_ev_int)(NIL), (less_eq_ev_real)(NIL)]
                [(mod_offer_qty)(NIL), (mod_offer_price)(NIL),
                 (propose)(NIL), (send)(propose,NIL,NIL,NIL)],
      (propose)[(seller_mode)(mode), (check_channel)(channel),
```

```
                 (is_zero_merch)(NIL)]
                 [(refuse)(NIL), (send)(refuse),
                  (remove_belief)(contract,NIL,NIL,NIL),
                  (remove_belief)(channel,NIL,NIL,NIL)],
      (accept)[(wrong_channel)(channel)][(skip)(NIL)],
      (accept)[(seller_mode)(mode), (check_channel)(channel)]
               [(less_stock)(NIL), (ack)(NIL), (send)(ack,NIL,NIL,NIL)
                (remove_belief)(contract,NIL,NIL,NIL),
                (remove_belief)(channel,NIL,NIL,NIL)],
      (ack)[(wrong_channel)(channel)][(skip)(NIL)],
      (ack)[(seller_mode)(mode), (check_channel)(channel)]
            [(less_stock)(NIL), (remove_belief)(contract,NIL,NIL,NIL),
             (remove_belief)(channel,NIL,NIL,NIL)],
      (refuse)[(wrong_channel)(channel)][(skip)(NIL)],
      (refuse)[(seller_mode)(mode), (check_channel)(channel)]
               [(remove_belief)(contract,NIL,NIL,NIL),
                (remove_belief)(channel,NIL,NIL,NIL)],
    ];
  };
  }
class_agent {Reactive}{Broker}
  role_playing{Broker} {
  init_comp{
    rules:[
    (register)(register),
    (finished)(finished),
    (search)(query)];
  };
  }
init_ENV [];
init_ETH [];
create_agent(Buyer,buyer1); {
  beliefs_component:[
    (need)(NIL,[apple],NIL,NIL),
    (buy_step)(NIL,NIL,NIL,0.1),
    (buy_start)(NIL,NIL,NIL,0.5),
    (agent_name)(buyer1,NIL,NIL,NIL),
    (agent_type)(buyer,NIL,NIL,NIL),
```

```
      (mode)(buy,NIL,NIL,NIL),
      (broker)(broker1,NIL,NIL,NIL),
      (apple)(buy,[],10,5.2),
      (pears)(buy,[],10,4.1),
      (channel)(broker1),
      (search)(buyer1,[apple,pears], NIL,NIL)];
    intentions_component:[
      (send)(search)
    ];
  }
  create_agent(Seller,seller1); {
    beliefs_component:[
      (stock)(NIL,[apple],NIL,NIL),
      (sell_step)(NIL,NIL,NIL,0.9),
      (sell_start)(NIL,NIL,NIL,1.5),
      (agent_name)(seller1,NIL,NIL,NIL),
      (agent_type)(seller,NIL,NIL,NIL),
      (mode)(sell,NIL,NIL,NIL),
      (broker)(broker1,NIL,NIL,NIL),
      (apple)(sell,[],20,5.0),
      (channel)(broker1),
      (register)(seller1,[apple],NIL,NIL)];
    intentions_component:[
      (send)(register,NIL,NIL,NIL)
    ];
  }
  create_agent(Trader ,trader1); {
    beliefs_component:[
      (stock)(NIL,[pears],NIL,NIL),
      (need)(NIL,[apple],NIL,NIL),
      (sell_step)(NIL,NIL,NIL,0.9),
      (sell_start)(NIL,NIL,NIL,1.3),
      (buy_step)(NIL,NIL,NIL,0.1),
      (buy_start)(NIL,NIL,NIL,0.6),
      (agent_name)(trader1,NIL,NIL,NIL),
      (agent_type)(trader,NIL,NIL,NIL),
      (mode)(buy,NIL,NIL,NIL),
      (broker)(broker1,NIL,NIL,NIL),
```

```
      (apple)(buy,[],10,5.1),
      (pears)(sell,[],10,4.0),
      (channel)(broker1),
      (search)(buyer1,[apple], NIL,NIL),
      (register)(seller1,[pears],NIL,NIL)];
    intentions_component:[
     (send)register;
     (send)search ];
  }
  create_agent(Broker ,broker1); {
    facts[]
  }
}
```