UNIVERSITÀ DEGLI STUDI DI PARMA
Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Introduzione del Supporto per Behaviour Complessi in un Linguaggio per Agenti Software

Relatore:
Chiar.mo Prof. Federico Bergenti

Candidato:
Riccardo Monica

Anno Accademico 2012/2013

Alla mia famiglia,
a Jessica
e a tutti coloro che
mi hanno sempre sostenuto.

A me stesso.

Indice

Introduzione

Il lavoro descritto in questa tesi è la realizzazione di un linguaggio di programmazione con dominio specifico l'ambiente agent-based JADE. Il linguaggio JADEL2 estende il linguaggio JADEL1 scritto da uno studente come lavoro di tesi. Questo DSL (domain-specific language) ha lo scopo di facilitare la creazione di elementi del dominio JADE. Attraverso questo linguaggio è possibile creare con facilità di scrittura e eliminazione di fasi ripetitive di inserimento codice, lasciando questa fase al generatore di codice. Fra le corde del linguaggio JADEL2 vi è quella di generare codice Java corretto per il dominio applicativo di questo DSL.

Il lavoro di tesi viene strutturato:
Nel Capitolo 1 si descrive il dominio specifico del linguaggio JADEL2. Viene studiato cosa s'intende per sistema multi-agente e gli standard che un sistema di questo tipo deve rispettare. Viene studiato anche il framework JADE dall'idea del sistema alle singole classi che compongono il middleware.
Nel Capitolo 2 viene descritto il plugin dell'IDE Eclipse per lo sviluppo dei linguaggi di programazione e DSL. Lo strumento utilizzato è Xtext. Nel capitolo vengono descritte le funzionalità di questo plugin e le caratteristiche principali. Viene anche descritta la sua grammatica che è stata poi molto utile nello sviluppo del linguaggio JADEL2.
Nel Capitolo 3 viene descritto il linguaggio che è stato esteso dal mio lavoro di tesi ovvero JADEL1. Ne vengono descritte le parti di grammatica e implementazione che sono rimaste nella nuova versione di JADEL2.
Nel Capitolo 4 viene descritto il linguaggio sviluppato durante questo lavoro di tesi ovvero JADEL2. Viene mostrata la grammatica e vengono descritte le classi principali usate per la traduzione e generazione del codice.
Alla fine di questo lavoro di tesi vi è anche un'appendice dove sono presenti esempi di programmi scritti nel linguaggio JADEL2 con il relativo codice corretto Java generato.

Capitolo 1

Agenti e JADE

Un agente è un sistema computazionale situato in un ambiente, è capace di azioni autonome nell'ambiente per raggiungere i suoi obiettivi di progetto.

1.1 Un sistema multi-agente

Per poter interagire, gli agenti in un sistema multi-agente devono comunicare fra loro.

Il meccanismo di comunicazione normalmente usato è quello dello scambio di messaggi. L'invio o la ricezione di un messaggio può essere visto come una azione eseguita dall'agente.

Sono stati definiti dei Linguaggi di Comunicazione fra Agenti in cui le azioni comunicative sono modellate basandosi sulla teoria filosofica delle azioni: *inform, request, query-if, ask, tell* etc...

Ci sono ovvie somiglianze fra agenti e oggetti. Entrambi sono entità computationali che incapsulano uno stato, sanno eseguire delle azioni (metodi) e comunicano via scambio di messaggi.

C'è però una differenza significativa: gli oggetti non hanno autonomia. Se un oggetto o ha un metodo pubblico, ogni altro oggetto può invocare questo metodo, e o, quando gli viene richiesto, deve eseguirlo. Viceversa, se un agente i chiede a un agente j di eseguire un'azione, non è garantito che j la eseguirà perché l'agente j è autonomo.

1.1.1 Lo standard FIPA

The *Foundation for Intelligence Physical Agent (FIPA)* è un'organizzazione che compone la IEEE Computer Society, promuovendo le tecnologie basate su agenti e l'interoperabilità con altre tecnologie. Gli standard definiti da questa organizzazione sono stati ufficialmente accettati dalla IEEE a partire dal 2005.

Un sistema multi-agente, per aderire allo standard, deve garantire alcuni aspetti:

- Una *piattaforma*, o *Agent Platform*, che identifica l'hardware contenente il sistema. Il sistema la vedrà come un unico spazio anche se potrà essere distribuita su più macchine contemporaneamente;
- La *possibilità di creare e gestire Agenti*. Ogni agente deve essere univoco e deve essere fornito di un nome identificativo e un indirizzo, utile per comunicare messaggi;
- Un *Directory Facilitator*: fornisce un elenco di servizi che vari agenti possono utilizzare sulla piattaforma fra i quali quello di "yellow pages" per gli agenti che desiderano ottenere servizi;
- Un *sistema di gestione* per gli agenti, gestisce lo spazio fisico e tiene l'elenco degli agenti presenti nella piattaforma;
- Un *sistema di comunicazione* standard per i messaggi tra le piattaforme di agenti, detto Message Transport System o MTS;
- Un *linguaggio* comune agli agenti per scambiarsi informazioni, o Agent Communication Language (*ACL*). In particolare viene anche implementato il concetto di ontologia, ovvero una rappresentazione che garantisce all'agente di carpire correttamente il messaggio ricevuto.

1.1.2 L'agente secondo lo standard FIPA

Come definito in precedenza un agente è un processo software fisico e come tale possiede un suo ciclo di vita, che è possibile dettagliare in una serie di stati e transizioni, secondo lo standard FIPA.

Gli stati sono:

- ACTIVE: l'agente è attivo e può ricevere messaggi;
- INITIATED: l'agente esiste ma non è ancora in grado di comunicare;
- WAITING: l'agente è in attesa di un evento che lo risvegli;
- SUSPENDED: l'esecuzione è temporaneamente sospesa;
- TRANSIT: l'agente si sta muovendo verso una nuova piattaforma, l'MTS raccoglie i messaggi con quell'agente come destinatario e li consegnerà non appena esso si sarà insediato a destinazione;
- UNKNOWN: l'agente è in uno stato non definito. In questo caso l'MTS mantiene i messaggi destinati a lui, in attesa di poterli consegnare o doverli distruggere.

Vi sono una serie di transizioni specifiche che consentono agli agenti di passare tra i vari stati. Possono interagire tra loro attraverso dei messaggi che vengono definiti dallo standard ACL.

La struttura di un messaggio generico è la seguente:

- PERFORMATIVE: definisce il tipo di messaggio tra gli agenti come: INFORM, REQUEST, CONFIRM, CANCEL etc...
- SENDER: chi invia il messaggio;
- RECEIVER: chi deve ricevere il messaggio;
- REPLY-TO: l'ente cui deve essere inoltrata una eventuale risposta;
- CONTENT: il contenuto della comunicazione tra i due agenti;
- LANGUAGE: il tipo di linguaggio che definisce il contenuto;
- ENCODING: la particolare codifica del messaggio;
- ONTOLOGY: indica l'ontologia specifica per decifrare i simboli espressi nel contesto;
- PROTOCOL: determina il protocollo di interazione richiesto dal mittente;
- CONVERSATION-ID: viene utilizzato per determinare un codice utile per riunire i messaggi relativi ad una comunicazione;
- REPLY-WITH: specifica una espressione che il destinatario utilizzerà per riferirsi in maniera univoca al messaggio;
- IN-REPLY-TO: specifica una eventuale azione cui il messaggio è in risposta;
- REPLY-BY: determina un limite di tempo per la risposta al messaggio.

1.2 JADE

JADE (Java Agent DEvelopment framework) è un framework completamente implementato in linguaggio Java. È un software free sviluppato e distribuito da Telecom Italia in collaborazione con l'Università degli Studi di Parma. L'ambiente JADE semplifica l'implementazione di sistemi multi-agente attraverso un middleware, che permetta di conformarsi alle specifiche FIPA e attraverso un insieme di tools supporti le fasi di debug e sviluppo.

Il framework JADE dispone di quattro principi guida:

- *Interoperabilità*: JADE soddisfa le specifiche FIPA. Come conseguenza gli agenti JADE possono interoperare con altri agenti a patto che soddisfino gli stessi standard.

- *Uniformità e portabilità*: JADE dispone di omogenei set di API che sono indipendenti dal network sottostante o dalla versione di Java utilizzata. Questo principio è fondamentale permettendone l'utilizzo in qualsiasi ambiente si voglia implementare. Più nel dettaglio, il runtime di JADE dispone all'utilizzo le stesse API per gli ambienti J2EE, J2SE e J2ME. Teoricamente, gli sviluppatori possono decidere l'ambiente runtime Java a tempo di sviluppo.

- *Facile da utilizzare*: la complessità del middleware è mascherata da un semplice e intuitivo set di API.

- La filosofia *Pay-as-you-go*: il programmatore non necessita di utilizzare tutte le funzionalità di cui dispone il middleware. Le funzionalità non richieste dal programmatore possono essere non conosciute.

1.2.1 Il sistema peer-to-peer e l'azione di comunicazione fra agenti

I sistemi agent-based fanno intrinsecamente riferimento a sistemi architetturali peer-to-peer: ogni agente è un peer che potenzialmente deve iniziare una comunicazione con un qualsiasi altro agente.

In un sistema peer-to-peer non vi è, infatti, alcuna distinzione di ruoli e ogni peer possiede diverse capacità. Ogni nodo può iniziare una comunicazione, essere soggetto o oggetto all'interno di una richiesta, essere proattivo. La logica delle applicazione non è più concentrata sul server, ma distribuita fra tutti i peer della rete dove ogni nodo ha la possibilità di scoprire ogni altro nodo; può anche entrare, unirsi o disconnettersi dalla rete in qualsiasi momento, qualsiasi sia la sua posizione.

Il ruolo della comunicazione è importante in un sistema agent-based, questo modello deve infatti rispettare tre principali caratteristiche:

- *"Gli agenti sono entità attive che possono rifiutare e sono debolmente accoppiati."* Questa proprietà sta alla base della scelta della messaggistica asincrona fra agenti. Se un agente vuol comunicare deve solo inviare un messaggio a una determinata destinazione. Questa modalità di comunicazione permette al ricevitore di scegliere quali messaggi elaborare e quali scartare, oppure quali messaggi servire per primo. Questa modalità permette anche al mittente di controllare il suo thread di esecuzione e non entrare in stato di block finchè il ricevente non ha letto e elaborato il messaggio. In conclusione, questa modalità di comunicazione rimuove qualsiasi dipendenza fra il mittente e il ricevente.

- *"Gli agenti eseguono azioni e la comunicazione è solo un tipo di azione."* Costruire una comunicazione allo stesso livello delle azioni permette all'agente di sfruttare un piano di azioni che includa al suo interno le azioni di comunicazione. Al fine di rendere la comunicazione pianificabile, gli effetti e le precondizioni quando possibile devono essere ben definite.

- *"Il verso della comunicazione ha significato semantico."* Quando un agente è oggetto di un'azione comunicativa, ad esempio quando riceve un messaggio, deve capirne il significato e perchè quell'azione è stata svolta. Questa proprietà fa notare il bisogno di uno standard universale semantico ed è da ricercare all'interno dello stato PERFORMATIVE del messaggio.

1.2.2 Il middleware e la tecnologia Java

Il termine *middleware* descrive tutte le librerie ad alto livello che permettono di sviluppare facilmente e efficientemente applicazioni software mettendo a disposizione utili servizi generici che non servono quindi a una sola applicazione, ma a una varietà più ampia possibile. L'idea che sta al di sotto del middleware è anche quella di aggirare il sistema operativo e rendere i servizi offerti liberi dai vincoli dei sistemi operativi. Questa capacità di riutilizzo attraverso più domini di applicazioni suggerisce il termine "orizzontale", opposto al termine "verticale", dove viene generata una soluzione ad-hoc. Proprio per facilitare ciò che è appena stato elencato si è pensato di utilizzare la tecnologia Java che fra i suoi paradigmi più famosi vi è il riutilizzo.

1.2.3 Il modello architetturale

Come è stato affermato in precedenza JADE è un middleware che facilita lo sviluppo di sistemi multi-agente. Questo include:

- Un ambiente runtime dove gli agenti possono "vivere" e devono essere attivi su l'host dato prima che altri
- Una classe di librerie che i programmatori possono utilizzare, direttamente o specializzandole, per sviluppare i loro agenti.
- Una suite di tool grafici che permette all'amministratore di monitorare le attività degli agenti in fase di esecuzione.

Ogni istanza dell'ambiente runtime di JADE è chiamato *Container* e può contenere diversi agenti. L'insieme di Containers attivi si chiama *Platform*. In una piattaforma deve sempre essere attivo un *Main Container* dove tutti gli altri contenitori si registrano ad esso come vengono avviati. Questo significa che il primo contenitore ad essere avviato sarà il Main Container mentre tutti gli altri, i "non principali", devono comunicare dove trovare, host e porta, del loro Main Container.

Se un altro contenitore principale è stato avviato da qualche altra parte nella rete, può costituire una piattaforma diversa dove è possibile registrarsi.

Gli sviluppatori non hanno la necessità di conoscere come l'ambiente di runtime di JADE funzioni, questa parte descritta in precedenza è nascosta allo sviluppatore che deve ricordarsi di avviare l'ambiente prima di eseguire i suoi agenti.

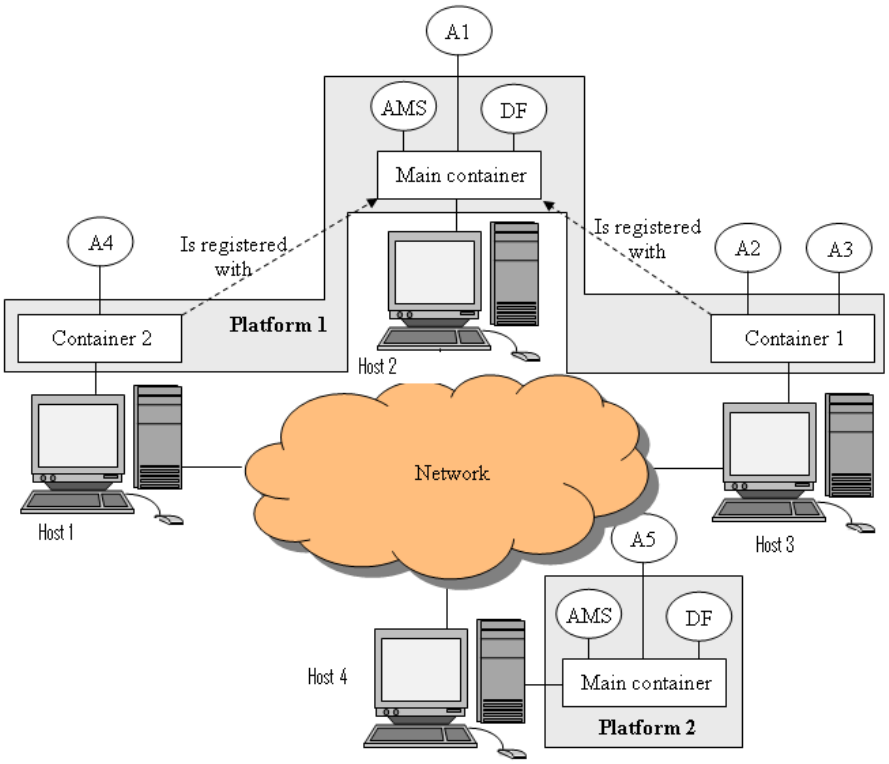
1.2.4 Gli agenti AMS e DF

All'interno del Main Container, in conformità a quanto previsto da FIPA, sono presenti degli agenti con ruoli speciali deputati alla gestione della piattaforma stessa. Questi due agenti speciali hanno come obiettivo quello di accettare le registrazioni da altri contenitori, vengono automaticamente avviati quando viene lanciato il contenitore principale.

Il primo agente è *AMS* (Agent Management System) che provvede al servizio di denominazione, ad esempio si assicura che ogni agente all'interno della stessa piattaforma abbia lo stesso nome. Quando l'AMS si è assicurato che l'agente ha un nome univoco assegna al nuovo agente un AID (Agent Identifier) univoco all'interno della piattaforma stessa. Rappresenta l'autorità dedita al management del sistema all'interno della piattaforma, ad esempio è possibile creare o uccidere agenti attraverso un contenitore remoto solo richiedendo questo all'AMS. Per permettere di utilizzare queste funzionalità l'AMS dispone di un servizio di pagine bianche della piattaforma, mantenendo un elenco di tutti gli agenti che in

un certo istante risiedono nella piattaforma stessa e memorizzano per ciascuno di essi il relativo AID.

Il secondo agente è *DF* (Directory Facilitator) che fornisce un servizio di pagine gialle ovvero un agente può trovare un altro agente che dispone di un determinato servizio attraverso una richiesta al sistema.



1.2.5 La struttura del middleware: i package

La libreria di JADE è sviluppata in package ovvero gruppi di classi Java raggruppate per svolgere azioni simili. I package che ho utilizzato per la maggiore sono:

- *jade.core*: contiene le classi che costituiscono il nucleo di JADE. Al suo interno vi sono le classi Agent, Container, Platform e Behaviours, all'interno del suo sub-package jade.core.behaviours.

- *jade.lang.acl*: contiene le classi di gestione della comunicazione fra agenti rispettando le specifiche FIPA come la classe ALCMessage o la classe MessageTemplate.

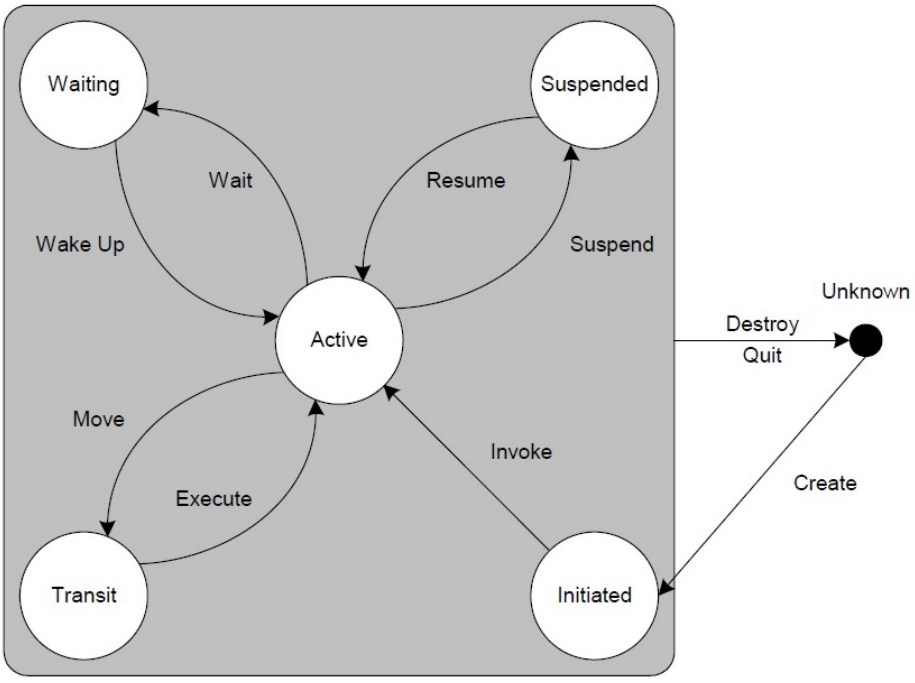
- *jade.content*: contiene le classe per la creazione e la gestione dell'ontologia definita dall'utente.

1.2.6 La classe Agent

La classe `Agent` rappresenta la classe base per definire gli agenti secondo le specifiche FIPA. Per creare un proprio agente JADE ha necessità di estendere la classe base `Agent`. Questo implica l'eredità di funzionalità utilizzabili per realizzare interazioni di base con la piattaforma di agenti come: registrazione, configurazione, rimozione. Viene ereditato dalla classe base anche tutti i metodi che possono essere chiamati per implementare il comportamento specializzato dell'agente e dei suoi behaviours.

Il modello computazionale di un agente è multitask, dove ogni task, o *behaviour* (dall'inglese "comportamento"), sono eseguiti concorrentemente. Ogni funzionalità o servizio sviluppato in un agente dovrebbe essere implementato in uno o più behaviours. Uno scheduler, interno alla classe base `Agent` e nascosto al programmatore, gestisce automaticamente lo scheduling dei task o behaviours.

Essendo conforme alle normative FIPA, un agente JADE dispone degli stati prima elencati. La classe `Agent` dispone di metodi pubblici per sviluppare transizioni fra i vari stati; questi metodi prendono il nome dalle transizioni dell'automa a stati finiti mostrato in figura. Ad esempio, il metodo doWait() mette l'agente in stato WAITING dallo stato ACTIVE.



L'avvio dell'esecuzione

Il framework di JADE controlla la nascita di un nuovo agente attraverso i seguenti passi:

I il costruttore dell'agente è eseguito;

II all'agente viene assegnato un identificazione (jade.core.AID);

III viene registrato con l'AMS, l'agente Agent Management System;

IV l'agente viene posto in stato ACTIVE;

V viene eseguito il metodo `setup()`; in accordo con le specifiche FIPA;

VI un nome unico globale. Per standard interno JADE compone questo nome unico come la concatenazione fra il nome locale, ovvero il nome che l'utente da a linea di comando, concatenato al carattere '@' a sua volta concatenato con l'identificatore della piattaforma alla quale l'agente è assegnato;

VII un insieme di indirizzi di agenti. Ogni agente eredita l'indirizzo di trasporto degli agenti appartenenti alla sua piattaforma;

VIII un insieme di servizi. Ad esempio quello di pagine bianche alla quale l'agente è registrato.

Il metodo `setup()` è comunque il punto dove qualsiasi applicazione basata su agenti inizia. Il programmatore deve implementare il metodo `setup()` per poter inizializzare il suo agente. All'interno di questo metodo il programmatore può, se necessario, modificare i dati registrati in AMS oppure registrarsi a più domini contemporaneamente. Le istruzioni obbligatorie da inserire all'interno del metodo `setup()` sono una lista di metodi `addBehaviours()` dove ogni singolo behaviour verrà schedulato appena il metodo `setup()` sarà terminato.

Terminazione dell'esecuzione

Ogni behaviour può chiamare il metodo `Agent.doDelete()` per ordinare all'agente la terminazione dell'esecuzione.

Il metodo `Agent.takeDown()` viene eseguito quando l'agente viene posto in stato DELETED. Questo metodo può essere sovrascritto dal programmatore per implementare una qualsiasi funzionalità di pulizia. Quando questo metodo è in esecuzione l'agente può continuare ad essere registrato all'interno dell'AMS e può comunque inviare messaggi ad altri agenti. Quando invece questo metodo termina, l'agente viene cancellato dall'AMS e il suo thread ucciso.

```
package PingPong;

import jade.core.Agent;

public class Ping extends Agent {
  protected Agent myAgent =  this ;

  public void setup() {

    PingSendBehaviour psb=new PingSendBehaviour(this);
    this.addBehaviour(psb);
    PingReceiveBehaviour prb=new PingReceiveBehaviour(this);
    this.addBehaviour(prb);
    PingSendReceiveMessage psrm =
    new PingSendReceiveMessage(this);
    this.addBehaviour(psrm);

  }
}
```

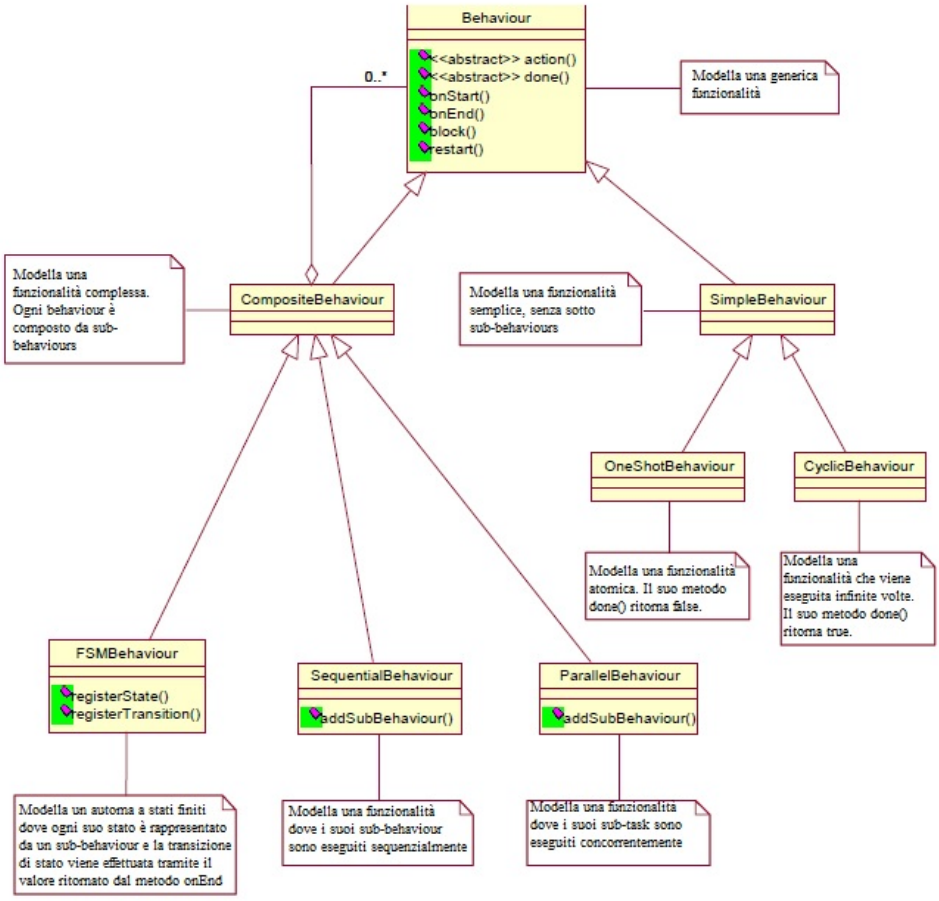
1.2.7 I Behaviour

Un agente, come è stato accennato in precedenza, deve aver la possibilità di definire diversi comportamenti concorrenti fra loro per poter rispondere a eventi esterni di qualsiasi entità. Ogni agente è composto da un singolo thread e ogni sua funzionalità può essere implementata come un oggetto Behaviour.

Lo scheduler di questi behaviours è implementato all'interno della classe `Agent` e nascosto al programmatore. Consiste in uno scheduling con politica round-robin non-preemptive tra tutti i behaviurs disponibili in stato pronto all'interno della coda. Un behaviour può anche mettersi in attesa per un messaggio in arrivo.

Nel dettaglio: lo scheduler dell'agente esegue il metodo `action()` di ogni behaviour presente in stato pronto nella lista. Quando il metodo `action()` termina viene chiamato il metodo `done()`, dopodiché il behaviour viene rimosso dalla lista.

I behaviours lavorano come thread cooperanti, ma non vi è alcuno stack dove salvare il loro risultato. Per questo motivo ogni behaviour deve mantenere lo stato del calcolo in variabili del behaviour stesso e dell'agente associato.



La classe Behaviur

Questa classe è astratta e provvede a modellare l'insieme di base delle funzionalità del behaviour. Sono presenti sei principali funzionalità:

- *action*(): questo metodo viene eseguito dallo scheduler dell'agente quando il behaviour è in stato pronto;
- *done*(): questo metodo viene eseguito dopo che il metodo action() è terminato;
- *onStart*(): questo metodo viene eseguito prima dell'esecuzione del behaviour;
- *onEnd*(): questo metodo viene eseguito dopo l'esecuzione del behaviour;
- *restart*(): questo metodo può essere chiamato dal behaviour stesso per resettare lo stato del behaviour;

- *block*(): questo metodo può essere eseguito per mettere in attesa un behaviour, solitamente per la ricezione di un messaggio.

La classe SimpleBehaviour

Questa classe astratta definisce modelli di behaviour semplici. Il suo metodo reset() non fa nulla, ma può essere implementato dalle sottoclassi.

Le classi OneShotBehaviour e CyclicBehaviour

La classe astratta OneShotBehaviour definisce un behaviour che sarà eseguito una sola volta, quindi il metodo done() ritornerà sempre true. La classe astratta CyclicBehaviour invece definisce un behaviour che verrà eseguito infinite volte e ritornerà false.

```
package esempi.pingpong;

import jade.core.behaviours.CyclicBehaviour;

public class PingSendReceiveMessage extends CyclicBehaviour{

  private int conta;

  public PingSendReceiveMessage(Ping ping) {
    super(ping);
  }

  @Override
  public void action() {
    ACLMessage msg = this.myAgent.receive();
    System.out.println("Receive:"+msg.getContent()+"␣from>␣"
        +msg.getSender().getName());

    ACLMessage msgRsp = new ACLMessage(ACLMessage.INFORM);
    msgRsp.addReceiver(msg.getSender());
    msgRsp.setContent("Ping");

    this.myAgent.send(msgRsp);
  }
}
```

La classe CompositeBehaviour

Questa classe astratta definisce modelli di behaviour che sono stati progettati per contenere un insieme variabile di altri behaviour.

La classe FSMBehaviour

Questa classe astratta modella l'esecuzione di un automa a stati finiti, di qualsiasi tipo, definito dall'utente. Ogni stato dell'automa a stati finiti è definito come un sub-behaviour; che utilizzerà il valore ritornato dal metodo onEnd() per effettuare una transizione di stato. Attraverso il metodo registerState() è possibile aggiungere uno stato all'automa e attraverso il metodo registerTransition() è possibile registrare una transizione di stato.

```
package examples.FSM;

import jade.core.Agent;

public class FSMAgent extends Agent {
//nomi degli stati
private static final String STATE_A = "A";
private static final String STATE_B = "B";
private static final String STATE_C = "C";
private static final String STATE_D = "D";
protected void setup() {
  FSMBehaviour fsm = new FSMBehaviour(this) {
    public int onEnd() {
      System.out.println("FSM behaviour completato.");
      myAgent.doDelete();
      return super.onEnd();
    }
  };
  //Registro gli stati
  fsm.registerFirstState(new Behaviour1(), STATE_A);
  fsm.registerState(new Behaviour2(), STATE_B);
  fsm.registerState(new Behaviour3(), STATE_C);
  fsm.registerLastState(new Behaviour6(), STATE_D);

  //Registro le transizioni
  fsm.registerDefaultTransition(STATE_A, STATE_B);
  fsm.registerDefaultTransition(STATE_B, STATE_C);
  fsm.registerTransition(STATE_C, STATE_C, 0);
  fsm.registerTransition(STATE_C, STATE_D, 1);
  fsm.registerTransition(STATE_C, STATE_A, 2);

  addBehaviour(fsm);
}}
```

Le classi SequentialBehaviour e ParallelBehaviour

Queste classi astratte definiscono due classici modelli di behaviour ovvero quello sequenziale, dove ogni behaviour viene eseguito solo dopo che quello precedente ha terminato la sua esecuzione; oppure quello parallelo, che esegue i sub-behaviour concorrentemente e termina quando una particolare condizione viene soddisfatta.

1.2.8 Le classi ACLMessage e MessageTemplate

La classe ACLMessage rappresenta il messaggio ACL che può essere scambiato fra due agenti. Questo messaggio contiene un'insieme di attributi definiti dalle specifiche FIPA.

Un agente che desidera inviare un messaggio deve creare un nuovo oggetto ACLMessage, inserire i parametri appropriati, elencati a inizio capitolo nelle specifiche degli standard dell'ACL, e invocare il metodo Agent.send(). Analogamente per il destinatario dovrà chiamare il metodo receive() o blockingReceive(), se non desidera riceverlo.

Un agente potrebbe avere più conversazioni in simultaneo. Siccome la coda dei messaggi in entrata è condivisa a tutti i behaviour, è stata introdotta la classe MessageTemplate che permette di confrontare i pattern dei messaggi destinati all'agente.

Usando il metodo di questa classe il programmatore potrà creare un pattern per ogni ACLMessage. I pattern elementari possono essere combinati fra loro attraverso gli operatori AND, OR e NOT per creare più complesse regole di matching.

```
...
MessageTemplate mt = MessageTemplate.or(
MessageTemplate.MatchPerformative(ACLMessage.INFORM),
MessageTemplate.or(MessageTemplate.MatchSender(Ping),
MessageTemplate.MatchPerformative(ACLMessage.REQUEST)));
...
ACLMessage msg = myAgent.receive(mt);
...
if(mt.match(msg)){...}
else{...}
...
```

1.2.9 L'Ontologia

Quando un agente vuole comunicare con un altro agente, vi sono un certo numero di informazioni che sono trasferite fra i due agenti attraverso un messaggio ACL. All'interno di questo messaggio, l'informazione è rappresentata come una espressione con un proprio linguaggio e codificata in un proprio formato (String). Entrambi gli agenti hanno la possibilità di rappresentare in via differente l'informazione che ricevono o inviano.

I contenuti

Un *content*, contenuto, è il significato che assegniamo a un possibile elemento all'interno di un discorso. Questa classificazione di contenuto è derivata dal linguaggio ACL definito dalle specifiche FIPA che richiedono che siano implementate all'interno di ogni ACLMessage. In primo luogo è necessario distinguere fra Predicato e Termine.

Un Predicato è un'espressione che afferma qualcosa sullo stato del mondo e può essere true o false.

I predicati solitamente sono utilizzati con le performative INFORM o QUERY-IF, in quanto non avrebbe senso utilizzarlo con REQUEST.

Un Termine è un'espressione che identifica un'entità che esiste nel mondo e che comunica con l'agente.

Le principale classificazioni di Termini sono: i concept e le agent actions.

I *Concept*, concetti, sono espressioni che indicano le entità con una struttura complessa che possono essere definite in termini di variabili.

Le *Agent Action* sono speciali concetti che indicano le azioni che possono essere effettuate da alcuni agenti, come può essere la vendita di un libro o l'accensione di una lampadina.

Definire un'ontologia

Per definire un ontologia è necessario:

I estendere la classe base Ontology, package jade.content.onto.Ontology;

II definire un nome dell'ontologia;

III definire un vocabolario;

IV definire l'ontologia come un modello Singleton;

V definire il costruttore: aggiungendo i contenuti all'ontologia attraverso le chiamate al metodo add() e definendone lo schema per ogni contenuto aggiunto ovvero Concept, Predicate o AgentAction.

Di seguito le dichirazione del vocabolario e identificativo dell'ontologia.

```
package esempi.pingpong;

import ...

public class OntologyPingPong extends Ontology{
//identificativo
public static final String NOME_ONTOLOGIA = "PP-Ontology";

//vocabolario
public static final String PERSONA = "Persona";
public static final String PERSONA_NOME = "Nome";
public static final String PERSONA_ETA = "Eta";

public static final String PROPRIETARIO = "Proprietario";
public static final String PROPRIETARIO_ID = "Aid";
public static final String PROPRIETARIO_PERSONA = "Persona";
public static final String PROPRIETARIO_MESSAGGIO = "Msg";

public static final String AGENTE = "Agente";
public static final String AGENTE_ID = "AID";
public static final String AGENTE_PERSONA = "A_Persona";
```

Un'ontologia deve essere un Singleton, ovvero deve essere creata una ed una sola istanza. Di seguito la definizione del pattern Singleton.

```
private static Ontology istanza = null;

public static Ontology getIstance(){
  if(istanza == null){
    istanza = new OntologyPingPong();
  }
  return istanza;
}
```

Di seguito il costruttore dell'ontologia.

```
private OntologyPingPong() {
  super(NOME_ONTOLOGIA, BasicOntology.getInstance());
  try{
    add(new ConceptSchema(PERSONA), Persona.class);
    add(new PredicateSchema(PROPRIETARIO),
            Proprietario.class);
    add(new AgentActionSchema(AGENTE),
            GestoreAgente.class);

    ConceptSchema cs =
    (ConceptSchema) getSchema(PERSONA);
    cs.add(PERSONA_NOME,
    (PrimitiveSchema) getSchema(BasicOntology.STRING));
    cs.add(PERSONA_ETA,
    (PrimitiveSchema) getSchema(BasicOntology.INTEGER),
                                  ObjectSchema.OPTIONAL);


    PredicateSchema ps =
    (PredicateSchema) getSchema(PROPRIETARIO);
    ps.add(PROPRIETARIO_ID,
              (ConceptSchema) getSchema(BasicOntology.AID));
    ps.add(PROPRIETARIO_PERSONA,
               (ConceptSchema) getSchema(PERSONA));
    ps.add(PROPRIETARIO_MESSAGGIO,
    (PrimitiveSchema) getSchema(BasicOntology.STRING),
                                   ObjectSchema.OPTIONAL);

    AgentActionSchema as =
    (AgentActionSchema) getSchema(AGENTE);
    as.add(AGENTE_PERSONA,
               (ConceptSchema) getSchema(PERSONA));
    as.add(AGENTE_ID,
              (ConceptSchema) getSchema(BasicOntology.AID));
  }
  catch(OntologyException e){
    e.printStackTrace();
  }
}}
```

Capitolo 2

Xtext

Xtext copre tutti gli aspetti di una completa infrastruttura di un linguaggio, comprese le implementazioni di un compilatore o un interprete.

2.1 DSL: Domain Specific Language

Un DSL è un linguaggio di programmazione dedicato alla risoluzione di particolari problemi all'interno di un dominio.

L'idea è quella che il concetto e la notazione sono il più vicino possibile a quello che lo sviluppatore ha in mente quando pensi alla soluzione in quel dominio. Chiaramente relativi a problemi che hanno soluzione.

L'opposto di un DSL è il GPL, ovvero General Purpose Language, che come Java, o qualsiasi altro linguaggio di programmazione, può risolvere un qualsiasi tipo di problema.

Un esempio di DSL è il linguaggio SQL che è concentrato sulle interrogazioni delle basi di dati relazionali. Un altro esempio di DSL è il linguaggio MathLab o XML.

2.2 Come lavora Xtext

Xtext dispone di un'insieme di DSL e API per descrivere i differenti aspetti di un linguaggio di programmazione. Sulla base di questi aspetti dà un'implementazione completa di questo linguaggio per poter essere eseguito sulla JVM (Java Virtual Machine). Le componenti del compilatore sono indipendenti da Eclipse e possono essere utilizzate in un qualsiasi ambiente Java. Queste componenti includono: parser, i type-safe abstract syntax tree (AST), il serializer e il code formatter, lo scoping framework e il linking, i controlli del compilatore e le analisi statiche conosciute come validation e il codice del generatore o dell'interprete. Questi componenti runtime sono integrati con l'EMF (Eclipse Modeling Framework), che permette a Xtext di essere utilizzato con altri EMF presenti.

Oltre a questo tipo di architettura runtime, Xtext fornisce un IDE Eclipse specifico per il linguaggio che si vuole sviluppare. Questo IDE dispone di tutte le funzionalità che normalmente sono presenti in quello originale di Eclipse permettendo di configurare o cambiare le funzionalità in modo molto semplice ed efficacie.

Questo editor conosce anche le parole chiave del linguaggio e, grazie alla grammatica che è stata definita, dove possono posizionarsi.

Xtext è utilizzato in molte industrie di diversi settori. É utilizzato nel campo dei dispositivi mobile, nello sviluppo automobilistico, in sistemi embedded e nello sviluppo dei videogame. Il linguaggi Xtext-based guidano i generatori a linguaggi specifici come Java, C, C++, C#, Python...

Xtext è un progetto Open-Source, per il progetto Eclipse.org, sviluppato da itemis, una società con sede in Germania, di consulenza specializzata e sviluppo di software model-based.

2.3 Definire una grammatica

Una delle prime funzionalità introdotte da Xtext è stata quella di definire un domain-specific language per la descrizione di linguaggi testuali. É un linguaggio chiamato "Grammar Language" che come idea principale ha quella di descrivere concretamente la sintassi e come viene mappata una sua rappresentazione in memoria ovvero il modello semantico.

2.3.1 La sintassi

Per poter definire una grammatica utilizzando il DSL messo a disposizione da Xtext, è necessario rispettare i vincoli sintattici che sono stati inseriti per la definizione dello stesso.

Il primo passo è quello di scrivere un header significativo dove si possono quindi trovare alcune proprietà fondamentali della grammatica. Dopo la parola chiave "grammar" è necessario quindi inserire il nome del linguaggio, dove è stato mantenuto da Xtext lo stesso meccanismo di path di Java. Il nome del file che conterrà la grammatica del linguaggio che andiamo a definire, verrà chiamato come il linguaggio stesso con estensione .xtext e dovrà essere contenuto nel package da noi indicato quando abbiamo inserito il nome del linguaggio.

Il secondo aspetto che si tiene in considerazione quando definiamo l'header è quello di poter riutilizzare un'altra grammatica già esistente. Dopo aver scritto la parola chiave "with" si scrive il package che contiene la grammatica che vogliamo importare.

```
grammar org.xtext.example.MyDsl
with org.eclipse.xtext.common.Terminals
```

La dichiarazione generate nel Grammar Language avvisa il framework di generare un package, in particolare Epackage con il nome scelto dal programmatore, in questo caso "mydsl".

```
generate mydsl 'http://www.eclipse.org/mydsl'
```

In alternativa, o in aggiunta, a questo metodo è possibile importare un package preesistente attraverso l'istruzione import con anche la possibilità di assegnare un nome ad esso.

```
import "http://www.xtext.org/example/Domainmodel"
as dmodel
```

2.3.2 Regole sintattiche

Il parsing si può suddividere in quattro fasi distinte: lexing, parsing, linking e validation.

Regole terminali

Il primo passo è chiamato lexing, ovvero una sequenza di caratteri che è trasformata in una sequenza di token. Un token consiste in uno o più caratteri che corrispondono a una particolare regola terminale, o a una parola chiave e in ogni caso rappresenta un simbolo atomico. Solitamente il nominativo delle regole terminali sono scritte in maiuscolo.

```
terminal ID :
('^')?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_
    '|'0'..'9')*;
```

Una regola terminale è definita formalmente come segue.

```
TerminalRule :
    'terminal' name=ID ('returns' type=TypeRef)? ':'
    alternatives=TerminalAlternatives ';' ;
```

Gli operatori per la cardinalità

Ogni espressione può definire una cardinalità. Ci sono quattro differenti possibilità:

- esattamente un elemento. Operatore: nessuno;
- uno o nessun elemento. Operatore: ?;
- zero o più elementi. Operatore: *;
- uno o più elementi. Operatore: +.

Parole chiave

Una parola chiave è una regola terminale letterale. Può contenere qualsiasi carattere ed ha lunghezza arbitraria.

```
terminal ID : '^'? 'keyword'* ..... ;
```

Range di caratteri

Un range di caratteri si dichiara utilizzando l'operatore '..'.

```
terminal INT returns ecore::EInt: ('0'..'9')+;
```

Negazione del token

Tutti i token possono essere negati utilizzando il carattere '!'.

```
terminal BETWEEN_HASHES : '#' (!'#')* '#';
```

Or token

In Grammar Language è possibile definire più opzioni nel file di input. Il carattere è '|'.

```
terminal ALTERNATIVE: ('one'|'two'|'three')*;
```

2.3.3 Regole di parsing

Le regole di parsing sono un'insieme di sequenze terminali e un insieme di passi (walkthrough). Una regola di parsing non produce un token terminale, ma un albero di token non terminali e terminali, chiamato *parse tree*.

L'assegnamento

L'assegnamento è usato per assegnare informazioni a una variabile dell'oggetto che stiamo producendo.

L'oggetto, un Eclass, è specificato nel tipo dell'oggetto ritornato dalla regola di parsing. Se non è stato specificato alcun tipo di ritorno allora il nome del tipo dichiarato sarà uguale al nome della regola. Il tipo della funzione assegnata viene dedotto dal lato destro dell'assegnamento.

```
State :
    'state' name=ID
    ('actions' '{' (actions+=[Command])+ '}')?
    (transitions+=Transition)*
    'end' ;
```

"name=ID" è un'assegnamento. Il lato sinistro fa riferimento a una caratteristica *name* dell'oggetto che viene creato.

Esistono tre differenti operatori di assegnamento:

- '='. Utilizzato quando la caratteristica deve aver un solo elemento;
- '+='. Utilizzato quando viene implementata una lista di elementi;
- '?='. Utilizzato quando viene implementato un valore booleano.

Cross-reference

Una caratteristica unica di Xtext è l'abilità di dichiarare cross-link nella grammatica. Tradizionalmente nella costruzione del compilatore i cross-link non sono stabiliti nella fase di parsing, ma nella fase di linking.

Questo vale anche in Xtext, ma è permesso specificare le informazioni di cross-link nella grammatica. Queste informazioni sono usate da linker nella fase successiva.

La sintassi per un cross-link è:

```
CrossReference :
    '[' type=TypeRef ('|' ^terminal=
        CrossReferenceableTerminal )? ']' ;
```

La transizione viene creata fra due cross-reference. L'operatore utilizzato è '=>'.

```
Transition :
    event1=[Event] '=>' event2=[Event] ;

Event returns MyEvent : ..... ;
```

Gruppi non ordinati

Gli elementi di un gruppo non ordinato possono apparire, banalmente, in qualsiasi posizione, ma ogni elemento deve apparire una sola volta. In Xtext un gruppo non ordinato è separato dal carattere '&'.

```
Modifier:
    static?='static'? & final?='final'? & visibility=
        Visibility;

enum Visibility:
    PUBLIC='public' | PRIVATE='private' | PROTECTED='
        protected';
```

Azioni semplici

L'oggetto che deve essere ritornato da una regola di parsing è solitamente creato al primo assegnamento. Il suo tipo è determinato dal tipo di ritorno della regola il quale è dovuto al nome della regola se non è stato esplicitamente specificato un tipo di ritorno.

Le azioni semplici possono essere utilizzati per forzare la creazione di un'istanza con un tipo specifico.

```
MyRule returns TypeA :
    "A" name=ID |
    "B" {TypeB} name=ID ;
```

Nell'esempio utilizzo la action per obbligare il TypeB ad essere un sottotipo di TypeA.

Con la notazione {TypeB} verrà creata un'istanza di TypeB e verrà assegnata al risultato della regola di parsing.

Regole di chiamata non assegnate

Come suggerisce il nome, le regole di chiamata non assegnate sono regole di chiamata per altre regole di parsing, che non sono usate con un assegnamento. Il valore ritornato della chiamata diventa il valore ritornato dalla regola di parsing chiamata, se non è assegnato ad una funzione.

```
AbstractToken :
    TokenA |
    TokenB |
    TokenC ;
```

Un AbstractToken può ritornare un'istanza di TokenA, TokenB e TokenC. AbstractToken è il super tipo per tutte e tre queste istanze.

Azioni di assegnamento

Xtext utilizza il parser ANTLR per le azioni di assegnamento. ANTLR usa l'algoritmo LL, ovvero della fattorizzazione a sinistra. Non è quindi permessa una grammatica ricorsiva a sinistra, a meno che non venga disabilitato il controllo all'interno del file MWE2.

```
Expression :
    TerminalExpression ('+' TerminalExpression)? ;

TerminalExpression :
    '(' Expression ')' |
    INT ;
```

2.4 Il file MWE2

L'MWE2, Modeling Workflow Engine 2, è un generatore esterno configurabile, di default generato da Xtext quando inizializziamo un nuovo progetto. In questo contesto, è utilizzato per descrivere il flusso di produzione per ottenere il modello corretto. Il file MWE2 contiene come e quali modelli devono essere caricati, come devono essere convalidati e dove e come la generazione del codice dovrebbe avvenire.

```
module org.xtext.example.jade.GenerateJade

import ...

var grammarURI = "classpath:..."
var fileExtensions = "jade"
var projectName = "org.xtext.example.jade"
var runtimeProject = "../${projectName}"
var generateXtendStub = true
var encoding = "UTF-8"

Workflow {
    bean = StandaloneSetup {...}

    component = DirectoryCleaner {...}
```

```
    component = DirectoryCleaner {...}

    component = DirectoryCleaner {...}

    component = DirectoryCleaner {...}

    component = Generator {
        pathRtProject = runtimeProject
        pathUiProject = "${runtimeProject}.ui"
        pathTestProject = "${runtimeProject}.tests"
        projectNameRt = projectName
        projectNameUi = "${projectName}.ui"
        encoding = encoding
        language = ...
    }
}
```

2.5 Xtend: un dialetto di Java

Xtend è un linguaggio di programmazione che viene tradotto in linguaggio Java. Sintatticamente e semanticamente Xtend ha le sue radici nel linguaggio Java, ma migliora molti dei suoi aspetti come:

- i metodi di estensione: permette di aggiungere nuovi metodi a tipi già esistenti senza modificarli;
- la lambda expression: una sintassi coincisa per funzioni anonime;
- annotazioni attive: permette agli sviluppatori di partecipare al processo di traslazione del codice Xtext al codice Java attraverso librerie;
- overloading di operatori: nuovi operatori rispetto a Java
- lo statement switch: non più limitato ad alcuni tipi e valori, ma può essere utilizzato per qualsiasi oggetto;
- metodo di invocazione polimorfico;
- template expression: consentono una leggibile concatenazione di stringhe attraverso la tripla singola quota (”’);
- nessuno statement: qualsiasi parte del codice è un'espressione che ha un tipo di ritorno.

Diversamente da altri linguaggi per la JVM Xtend non ha problemi di interoperabilità con Java, ovvero qualsiasi cosa venga scritta in Xtend ha il codice Java esattamente come lo si voleva. A differenza di Java Xtend è molto più coinciso, leggibile e espressivo.

```
package sample

import java.util.List

class Greeter {
  def greetThem(List<String> names) {
    for(name: names) {
      println(name.sayHello)
    }
  }

  def sayHello(String name)
     '''Hello <<name>>!'''
}
```

2.6 Xbase

Un DSL in Xtext è abbastanza semplice da implementare, ma un sistema software non consiste unicamente di strutture. Questo tipo di sistema necessità di alcune funzionalità, note come espressioni (Xespression nel linguaggio di Xbase). Le espressioni sono il nucleo di ogni linguaggio di programmazione e non sono di semplice implementazione, non in ogni caso possibile almeno. In ogni caso comunque le espressioni sono ben note e molti linguaggi di programmazione ne condividono alcuni tipi.

Questo è il principale motivo che ha portato gli utenti a non aggiungere i supporti per le espressioni nei loro DSL, ma di risolvere la situazione in maniera differente. La soluzione più utilizzata è quella di definire solamente le informazioni strutturali nel DSL e aggiungere successivamente i comportamenti per estendere o modificare il codice generato. Questa soluzione non è solo complessa da scrivere, capire e mantenere, ma anche se modifichiamo il codice generato è presente la possibilità di riscontrare molteplici problemi.

É stato quindi creato Xbase un linguaggio che funge da libreria e che fornisce un linguaggio di espressioni comuni legato alla piattaforma Java, ovvero alla Java Virtual Machine. Consiste di una grammatica, definita in Xtext, riusabile e adattabile alle diverse implementazioni per diversi aspetti del linguaggio che si vuole implementare. É stato studiato per interagire con l'IDE Eclipse Xtext-based. Concettualmente e sintatticamente Xbase è molto simile a Java, ma con alcune differenze elencate e studiate di seguito.

2.6.1 Identificatori e parole chiave

Gli identificatori sono usati per assegnare un nome a tutti i costrutti. Xbase utilizza gli identifier-syntax di default in Xtext. In confronto a quelli

di Java, sono più semplificati per essere più utilizzabili nei casi comuni perché hanno meno ambiguità. Gli identificatori Xbase iniziano con una lettera o un underscore e sono seguiti da altri caratteri o numeri.

Gli identificatori non devono avere gli stessi caratteri delle parole chiave. Questa limitazione può essere risolta aggiungendo il prefisso '^'.

```
terminal ID:
  '^'? ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_
      '|'0'..'9')* ;
```

2.6.2 Tipi

Tutti i tipi della Java Virtual Machine sono disponibili. Xbase supporta tutti i tipi semplici, ovvero quelli che consistono solo di un nome qualificante, che può essere separato da un punto come in Java (java.lang.String). I tipi parametrizzati sono come in Java, con l'unica differenza che in Xbase un type reference può essere anche un tipo funzione.

Xbase supporta tutte le primitive Java. Le regole di conformità sono esattamente come sono definite nelle specifiche del linguaggio Java.

Gli array non possono essere istanziati arbitrariamente, ma ci sono funzioni della libreria che permettono di creare array con una lunghezza fissa e/o con valori iniziali. Oltre a questa limitazione, possono essere convertiti, su richiesta, in una lista di componenti.

Xbase introduce la lambda expression, e altre funzionalità. A livello JVM le funzioni sono un'istanza del tipo Functions della libreria di Xbase e dipendono dal numero di argomenti.

2.6.3 Espressioni

Le espressioni sono il costrutto principale del linguaggio e sono utilizzate per esprimere comportamenti o computare valori. Il concetto di statement, come accennato in precedenza non è supportato, ma le espressioni vengono utilizzate per gestire situazioni in cui la natura imperativa degli statement sarebbe utile. Un'espressione ritorna sempre un valore, che può essere anche null. In aggiunta, ogni espressione risolta è un tipo statico.

Espressioni Letterali

Un'espressione letterale, literals, denota un valore che non può cambiare nel tempo. Questo tipo di espressioni possono essere riferite a stringhe, numeri, booleani, null e ai tipi di Java supportati,attraverso l'espressione .class (ad esempio List.class).

Lambda Expression

La lambda expression è un'espressione letterale che definisce una funzione anonima. Questa espressione ha il permesso di accedere alle variabili del dichiaratore. Qualsiasi variabile final e parametro visibile in fase di costruzione può essere indicata nel corpo della lambda expression. Questa espressione è nota anche come chiusura.

La sintassi della lambda expression è:

```
myList.findFirst([ e | e.name==null ])
```

dove 'e' è l'elemento da trovare.

La lambda expression produce oggetti funzione. Il tipo è Function Type, parametrizzato con il tipo del parametro della lambda e il tipo di ritorno. Il tipo di ritorno non viene mai specificato in modo esplicito, ma è sempre dedotto dall'espressione. I tipi di parametri possono essere dedotti se la lambda expression è usata in un contesto quando questo è possibile.

If e Switch Expression

L'espressione if è usata per scegliere due valori differenti basati su un predicato. La sintassi Java dello statement è un operatore ternario, dove se il predicato è vero viene eseguita l'espressione dell'if altrimenti quella interna all'else. Il valore ritornato è, banalmente, quello dell'espressione eseguita nell'if o nell'else.

L'espressione switch è un po' differente da quella di Java. Prima di tutto non ci sono salti dove è possibile valutare una sola espressione. La seconda differenza è quella che l'uso dello switch non è limitato a certi valori, ma può essere utilizzato per qualsiasi oggetto.

```
switch myString {
  case myString.length>5 : 'str > 5'
  case 'foo' : "contiene foo"
  default : "E' una corta non-foo string."
}
```

Block

L'espressione block permette di avere sequenze di codice imperativo. Consiste di sequenze di espressioni e un valore di ritorno nell'ultima espressione contenuta all'interno del blocco. Un blocco vuoto ritorna null.

Loop Expression

Come in Java esistono tre principali tipi di cicli: for, while e do-while. Il ciclo for è un'espressione usata per eseguire un certo numero di volte un blocco di

espressioni e per ogni elemento dell'array è istanza di un Iterable. Il valore di ritorno è void. I cicli while e do-while, come in Java, sono utilizzati per eseguire un certo blocco di espressioni finché l'espressione del predicato non è false. La differenza fra i due è che il do-while viene eseguito almeno una volta. Analogamente al ciclo for, il valore di ritorno è void.

2.6.4 Dichiarazione di variabili

La dichiarazione di variabili è permessa solo all'interno dell'espressione block. La variabile dichiarata è visibile in qualsiasi seguente espressione del blocco. Generalmente, l'overriding di variabili fuori dallo scope non è permesso.

Una dichiarazione di variabile inizia con la parola chiave **val** che denota un valore non modificabile, la variabile *final* in Java. Quando si ha la necessità di utilizzare variabili che necessitano di modifiche si utilizza la parola chiave **var**.

```
{
  val max = 100
  var i = 0
  while (i > max) {
    println("Saluti!")
    i = i +1
  }
}
```

2.7 Xbase e Xtext

2.7.1 JVM Types

Un caso comune di sviluppo di un linguaggio è quello di far riferimento a concetti esistenti in altri linguaggi. Xtext rende questo molto semplice per tutti i DSL definiti.

Per questo motivo è utile poter avere accesso ai tipi della Java Virtual Machine. É possibile creare cross-reference alle classi, interfacce e altri campi e metodi.

Per utilizzare i modelli dei tipi JVM è necessario importare l'Ecore model JavaVMTypes

```
import "http://www.eclipse.org/xtext/common/JavaVMTypes"
as jvmTypes
```

Il passo successivo è quello di creare un cross-reference

```
DataType:
    'datatype' name=ID
    'mapped-to' javaType=
    [jvmTypes::JvmType|QualifiedName];
```

Dopo aver rigenerato il codice quindi sarà possibile scrivere codice come questo:

```
datatype Date mapped-to java.util.Date
```

2.7.2 Java Types usando Xbase

Xbase offre una parser rule JvmTypeReference che supporta completamente la sintassi dei type reference di Java. Per poter iniziare a utilizzarle è necessario far ereditare la grammatica di Xbase alla propria.

```
grammar org.eclipse.xtext.example.MyDsl
with org.eclipse.xtext.xbase.Xbase
```

Per poter implementare le funzionalità implementate nella nostra grammatica è necessario costruire un nostro generatore di codice che generi il file Java attraverso l'uso di un compilatore per le espressioni incontrate nel linguaggio. Se abbiamo esteso la grammatica di Xbase sarà anche necessario utilizzare il compilatore e il generatore di codice forniti da Xbase.

```
class DomainmodelGenerator implements IGenerator {

  override void doGenerate(Resource resource,
      IFileSystemAccess fsa) {
    for(e: resource.allContents.toIterable.filter(typeof(
        Entity))) {
      fsa.generateFile(
        e.fullyQualifiedName.toString("/") + ".java",
        e.compile)
    }
  }

  def compile(Entity it) '''...'''
  ...
}
```

Capitolo 3

JADEL1

JADEL1 è un linguaggio di programmazione che facilita la creazione di agenti software JADE.

É stato sviluppato da uno studente del Corso di Laurea in Informatica come lavoro di tesi. Questo linguaggio, analogamente a quello sviluppato durante il mio lavoro di tesi, è stato implementato tramite l'utilizzo di Xtext e Xbase. L'ambiente Xtext, come spiegato nel capitolo precedente, viene utilizzato per lo sviluppo della grammatica e Xbase per la gestione e semplificazione delle espressioni.

La versione del linguaggio che mi è stata assegnata presenta mancanze dal lato di generazione del codice, infatti, come ho iniziato a lavorare su questo progetto si è notata la parziale assenza o assenza di codice per la traduzione e generazione del codice Java. Lo scopo, infatti, è quello di scrivere un linguaggio di programmazione che faciliti la creazione di agenti JADE e che questo programma scritto in JADEL sia tradotto in linguaggio Java corretto e comprensibile.

Nonostante numerose prove ho dovuto riscrivere o estendere il codice, scritto dallo studente che mi ha preceduto, in quanto il framework Xtext, aggiornato alle ultime versioni, potrebbe essere stato anche questo un problema, non riusciva a tradurre alcun programma scritto in linguaggio JADEL. In questo capitolo descrivo il linguaggio JADEL1 ovvero quello su cui ho iniziato a lavorare. Ne descriverò la grammatica e le parti di codice di traduzione e generazione che ho mantenuto ed esteso nel linguaggio di programmazione che ho sviluppato in questo lavoro di tesi denominato JADEL2.

3.1 La grammatica

La definizione della grammatica del linguaggio è data attraverso il Grammar Language fornito da Xtext. Di seguito illustro le parti delle parser rule e elementi che ho mantenuto nella grammatica.

3.1.1 L'Header e la Parser rule Model

Il frammento di codice seguente riporta l'header della grammatica che dichiara il package del linguaggio, viene importata la grammatica di Xbase e viene dichiarato l'Ecore model di JADEL1.

```
grammar org.xtext.example.jade.Jade
with   org.eclipse.xtext.xbase.Xbase

generate jade "http://www.xtext.org/example/jade/Jade"

import "http://www.eclipse.org/xtext/xbase/Xbase" as xbase
```

La parser rule Model è la prima regola della grammatica ed è usata come iniziale. Contiene la dichiarazione del package attraverso la parola chiave 'package' e il path del package che contiene il file. Contiene anche l'insieme degli import attraverso la regola XimportSection e un numero, che varia da 0 a più di uno, di AbstractElement.

```
Model:
  'package' name = QualifiedName
  importSection = XImportSection?
  elements+=AbstractElement*
;
```

Un AbstractElement come riportato di seguito è composto, in JADEL1, da tre componenti del linguaggio ovvero un SimpleBehaviour, che esprime i concetti relativi ai Behaviour, un AgentJade, che esprime i concetti relativi agli agenti semplici di JADE e AbstractOntology che esprime i concetti relativi alle ontologie.

```
AbstractElement :
  SimpleBehaviour | AgentJade | AbstractOntology
;
```

3.1.2 L'ontologia

A sua volta una AbstractOntology può essere suddivisa in OntologyElement oppure OntologyStructure. La prima rappresenta un elemento base di un'ontologia, ovvero implementa una delle tre interfacce principali: Concept, Predicate e AgentAction. Il campo type della parser rule determina, scegliendo una delle tre possibilità, quale delle tre inferfacce si vuole implementare. Il campo name rappresenta il nome della classe, il campo superType, preceduto dalla parola chiave extends, il tipo esteso dalla classe. Il campo body è una lista di elementi AddField che non sono altro che attributi della classe definiti da un type reference della JVM e da un id. La seconda invece rappresenta la struttura dell'ontologia ed è definita come una lista di elementi definiti nella lista di OntologyElement, denominata elements. Un'ontologia in JADEL1 si definisce tramite la parola chiave ontology dopodiché si definisce un nome e una lista di elementi.

```
AbstractOntology :
  OntologyElement | OntologyStructure
;

OntologyElement :
  type=('concept'|'predicate'|'agentAction')
  name=ValidID ('extends' superType = [OntologyElement])?
  '{'
     body+= AddField*
  '}'
;

AddField :
  'field' Ftype=JvmTypeReference  name=ValidID
;


OntologyStructure :
  'ontology' name=ValidID  '{'
    ('add' elements+= [OntologyElement])*
  '}'
;
```

3.1.3 Il concetto di Behaviour

La parser rule SimpleBehaviour contiene il concetto di Behaviour, che è stato sviluppato e esteso in JADEL2. In JADEL1 si può infatti definire un behaviour tramite la parola chiave behaviour, un id valido e un tipo di SimpleBehaviour (OneShotBehaviour o CyclicBehaviour) definito in seguito alla parola chiave is. Nel seguito della sua definizione vi è una

XblockExpression ovvero un espressione Block della grammatica Xbase utilizzata per rappresentare qualsiasi cosa si possa rappresentare all'interno del blocco. E' necessario notare che attraverso questa forma si può definire esclusivamente il metodo action() del behaviour, causando una limitazione importante sul funzionamento del Behaviour stesso.

```
SimpleBehaviour :
  'behaviour' name=ValidID 'is' typeSB = TypeSB '{'
    (body=XBlockExpression)
  '}'
;

enum TypeSB   :
  OneShotBehaviour | CyclicBehaviour
;
```

3.1.4 Il concetto di Agent

La parser rule AgentJade, in JADEL1, è definita dalla parola chiave agent e un id valido alla quale segue un blocco di istruzioni necessarie se si vuole inserire un'ontologia all'agente, oppure inserire un behaviour nello scheduler dell'agente. Per inserire un'ontologia è necassario, dopo aver scritto le parole chiave add e ontology, scrivere l'id della struttura dell'ontologia che si vuole implementare. Se si vuole inserire behaviours nello scheduler dell'agente bisogna, attraverso la parser rule AddBehaviour, definire un nuovo behaviour attraverso l'id name di tipo SimpleBehaviour.

```
AgentJade:
  'agent' name=ValidID '{'
    ('add' 'ontology' ontology = [OntologyStructure])?
    behaviours += AddBehaviour*
    messages += Message*
  '}';

AddBehaviour :
  'add' 'behaviour' name=ValidID
  'type' type = [SimpleBehaviour|ValidID]
;

Message returns xbase::XExpression:
  SendMessage | ReceiveMessage
;
```

3.1.5 SendMessage

La parser rule SendMessage ritorna una espressione di Xbase creata per rappresentare l'invio di un messaggio attraverso la classe ACLMessage. Questa parser rule è utilizzabile sia in un agente che in un behaviour, essendo una Xexpression. Una regola SendMessage si definisce attraverso un id preceduto dalle parole chiave send e message; il blocco seguente rappresenta tutti i possibili parametri settabili all'interno del messaggio tra i quali vi è: sender, content, ontology, performative e receivers.

```
SendMessage returns xbase::XExpression:
  {SendMessage}
  'send' 'message' name = ValidID '{'

    (('sender=' sender = STRING)? &
    ('content=' (content = STRING | ontologyContent=
        CreateElement ) )? &
    ('language=' (language = STRING | codec?= "
        agentLanguage") )? &
    ('ontology=' (ontology = STRING | agentOntology?= "
        agentOntology") )? &
    ('performative=' performative = Performative) &
    ('receivers' '{'(receivers += (MessageReceivers)+)'}'))
  '}'
;

MessageReceivers :
  'receiver =' receiver = STRING
;
```

3.2 Le classi per l'implementazione

Xtext mette a disposizione un insieme di classi, come accennato nel capitolo precedente, da estendere per definire l'implementazione del proprio linguaggio di programmazione o DSL. Nei prossimi paragrafi descrivo le parti delle classi che ho mantenuto, per poi spiegare nel capitolo successivo le estensioni di codice che ho svolto per poter generare il codice correttamente e estendere il codice. Tutte le classi di cui parlerò sono scritte in linguaggio Xtend.

3.2.1 La classe JadeJVMModelInferrer

La classe JadeJVMModelInferrer estende la classe AbstractModelInferrer che a sua volta implementa l'interfaccia IjvmModelInferrer. L'idea di base è quella di poter traslare le parser rule del proprio linguaggio in un qualsiasi oggetto appartentente al linguaggio Java.

Nel linguaggio che ho sviluppato ho conservato, avendo mantenuto la stessa struttura per la grammatica, lo sviluppo secondo ogni istanza della parser rule Model, tradotta nella classe Model. Attraverso il ciclo for vengono scanditi tutti gli elementi del campo elements della parser rule Model ovvero SimpleBehaviour, AbstractOntology e AgentJade. Ognuno di questi elementi verrà tradotto nel relativo codice Java.

```
package org.xtext.example.jade.jvmmodel

import ...

class JadeJvmModelInferrer extends AbstractModelInferrer {

  @Inject extension JvmTypesBuilder

  @Inject extension IQualifiedNameProvider

  def dispatch void infer(Model element,
                IJvmDeclaredTypeAcceptor acceptor,
                boolean isPrelinkingPhase) {

    for (a : element.elements) {
      switch a {
        SimpleBehaviour : {...}

        AbstractOntology : {
          switch a {
            OntologyElement : {...}
            OntologyStructure : {...}
          }
        }
        AgentJade : {...}
      }
}
```

AbstractElement

Per ognuno degli elementi della parser rule AbstractElement ovvero SimpleBehaviour, AbstractOntology e AgentJade, ho lasciato la struttura iniziale del linguaggio JADEL1 all'interno del mio linguaggio di programmazione. La parte iniziale si riferisce alla creazione della classe Java corrispondente all'elemento.

AbstractOntology

Il tipo AbstractOntology può essere di tipo OntologyElement oppure di tipo OntologyStructure.

```
AbstractOntology : {
  switch a {
    OntologyElement : {
      acceptor.accept(
        a.toClass(a.fullyQualifiedName)).initializeLater [
        documentation = a.documentation
        if (a.superType == null)
          superTypes += a.newTypeRef(
            'jade.content.'+ a.type.toString.toFirstUpper)
        else
          superTypes += a.newTypeRef(
            a.superType.fullyQualifiedName.toString)
    ...
  }
  OntologyStructure : {
    acceptor.accept(a.toClass(a.fullyQualifiedName)).
    initializeLater [
      documentation = a.documentation
      superTypes += a.newTypeRef('Ontology')
      members += a.toField("ONTOLOGY_NAME",
        a.newTypeRef(typeof(String)))[
      visibility = JvmVisibility::PUBLIC
      initializer = [  append (''''<<a.name>>"'''')]
      static = true
    ]
    members += a.toMethod("getIstance",
        a.newTypeRef('Ontology'))[
      static = true
      body = [  append ('''return theInstance;''')]
    ]
    members += a.toConstructor[
      documentation = '''...'''
      parameters += toParameter("base",
        newTypeRef(a, 'Ontology'))
      ...]
}
```

Si può notare che ho conservato la parte della creazione della classe in senso stretto e della definizione del super tipo per entrambi i tipi. Per la classe OntologyStructure ho mantenuto anche la parte della definizione del costruttore e della sua istanza. L'ontologia deve essere definita come un Singleton ovvero deve garantire che venga creata una e una sola istanza, e fornire un punto di accesso globale a tale istanza.

SimpleBehaviour

Analogamente anche per questo tipo ho confermato la creazione della classe e la definizione del super tipo.

```
SimpleBehaviour : {
  acceptor.accept(a.toClass(a.fullyQualifiedName)).
  initializeLater [
    superTypes += a.newTypeRef('jade.core.behaviours.'+
      a.typeSB.toString)
    ...
  ]
  ...
}
```

AgentJade

Per il caso AgentJade invece, oltre alla creazione e definizione del super tipo, che deve estendere la classe jade.core.Agent, ho lascaito la parte relativa alla definizione di alcuni campi come quello relativo al ContentManager contenente la piattaforma dalla quale l'agente viene lanciato, all'ontologia dell'agente. All'interno del metodo setup(), invece è rimasta invariata la parte di registrazione dell'ontologia.

```
AgentJade : {
  acceptor.accept(a.toClass(a.fullyQualifiedName)).
  initializeLater [
    documentation = a.documentation
    superTypes += a.newTypeRef('jade.core.Agent')
    members += a.toField("myAgent",
        a.newTypeRef('jade.core.Agent'))[
      visibility = JvmVisibility::PROTECTED
      initializer = [  append ('''' this '''')]
    ]
    members += a.toField("manager",
        a.newTypeRef("jade.content.ContentManager"))[
      initializer = [
        append ''''(ContentManager) getContentManager() ''''
      ]
    ]
```

```
    if (a.ontology != null){
      members += a.toField("ontology",
          a.newTypeRef("jade.content.onto.Ontology"))[
        initializer = [ append ('''<<a.ontology.name>>.
            getInstance()''')]
      ]
    }
    members += a.toMethod("setup",a.newTypeRef(Void::TYPE))[
      body = [
        if (a.ontology != null){
          append('''manager.registerOntology(ontology);''')}
        ...
      ]
    ]
  ]
}
```

3.2.2 La classe JadeCompiler

La classe JadeCompile che estende la classe XbaseCompiler, necessario per poter utilizzare il compilatore di Xbase in modo efficace e per poter compilare le espressioni presenti nella grammatica di Xbase. Dalla definizione della grammatica si può notare che sono state definite nuove espressioni. Quelle che ho mantenuto nel mio linguaggio sono SendMessage e MyAgent.

Il metodo doInternalToJavaStatement, come afferma il nome del metodo, traduce l'espressione in base al suo tipo. Se riceve una delle nuove espressioni estese da JadeCompiler le gestisce altrimenti vengono gestite dalla super classe come un'espressione di Xbase, gestita dal compilatore di Xbase.

```
package org.xtext.example.jade.jvmmodel

import ...

class JadeCompiler extends XbaseCompiler {

  @Inject extension JvmTypesBuilder
  @Inject extension TypeReferences

  override protected doInternalToJavaStatement(
    XExpression expr, ITreeAppendable it, boolean
        isReferenced) {
    switch expr {
      MyAgent : {...}

      SendMessage : {...}

      ReceiveMessage : {...} //JADEL2
```

```
        default : {
         super.doInternalToJavaStatement(expr,it,isReferenced)
        }
      }
    }
}
```

MyAgent

L'espressione MyAgent viene tradotta nell'attributo myAgent che si riferisce all'agente del behaviour. Questa espressione viene utilizzata all'interno del corpo del SimpleBehaviour per riferirsi all'agente che lo sta utilizzando

```
MyAgent : {
  newLine
  append('''myAgent''')
}
```

SendMessage

L'espressione SendMessage: crea un oggetto ACLMessage e in base ai valori degli attributi di SendMessage vengono settati i vari campi di messaggio da inviare come: il sender, l'ontologia o i receivers.

```
SendMessage : {
  newLine

  append('''
    jade.lang.acl.ACLMessage <<expr.name>> =
      new jade.lang.acl.ACLMessage(
        jade.lang.acl.ACLMessage.<<expr.performative>>);
    <<IF expr.sender != null>>
      <<expr.name>>.setSender(new jade.core.AID (
        "<<expr.sender>>",jade.core.AID.ISLOCALNAME));
    <<ENDIF>>
    <<IF expr.content != null>>
      <<expr.name>>.setContent("<<expr.content>>");
    <<ELSEIF expr.ontologyContent != null>>
      <<createElement(expr.ontologyContent as CreateElement,
          it)>>
       agent.manager.fillContent(<<expr.name>>,
        <<expr.ontologyContent.name>>);
    <<ENDIF>>
    <<IF expr.language != null>>
      <<expr.name>>.setLanguage("<<expr.language>>");
    <<ELSEIF expr.codec != false>>
      <<expr.name>>.setLanguage(agent.codec.getName());
```

```
    <<ENDIF>>
    <<IF expr.ontology != null>>
      <<expr.name>>.setOntology(<<expr.ontology>>);
    <<ELSEIF expr.agentOntology != false>>
      <<expr.name>>.setOntology(agent.ontology.getName());
    <<ENDIF>>
    <<FOR r : expr.receivers>>
      <<expr.name>>.addReceiver(new jade.core.AID (
        "<<r.receiver>>",jade.core.AID.ISLOCALNAME));
    <<ENDFOR>>
    this.myAgent.send(<<expr.name>>);
  ''')
}
```

In caso il content da mandare non sia una stringa ma un elemento dell'ontologia viene chiamato il metodo createElement che si occupa di costruire l'oggetto.

```
def private void createElement(CreateElement prova,
    ITreeAppendable it){
  append('''<<prova.type.name>> <<prova.name>> =
    new <<prova.type.name>>();''')
  newLine
  for (fieldElement : prova.fieldElements){
    append('''<<prova.name>>.
      set<<fieldElement.name.toFirstUpper>>(''')
    switch fieldElement.value {
      XNumberLiteral : {
        append('''<<(fieldElement.value as XNumberLiteral).
            value>>);''')
      }
      XStringLiteral : {
        append('''"<<(fieldElement.value as XStringLiteral).
            value>>");''')
      }
      XBooleanLiteral : {
        append('''<<(fieldElement.value as XBooleanLiteral).
            isTrue>>);''')
      }
    }
    newLine
  }
  for (structureElement : prova.structureElements){
    createElement(structureElement as CreateElement,it)
    append('''<<prova.name>>.set<<structureElement.type.name
        .toFirstUpper>>(<<structureElement.name>>);''')
    newLine
  }
}
```

Il metodo riceve in ingresso un CreateElement che rappresenta l'elemento dell'ontologia da costruire per poi essere inserito nel content del messaggio. CreateElement ha una struttura ad albero, in quanto un elemento complesso di un'ontologia può contenere anche altri elementi dell'ontologia stessa. Per questo motivo il metodo è ricorsivo. Nel primo ciclo for vengono settati tutti gli elementi semplici del nodo, le foglie, mentre nel secondo vengono effettuate le chiamate ricorsive sugli elementi più complessi del nodo. Il risultato è, appunto, un elemento dell'ontologia attraverso la chiamate dei metodi set.

3.2.3 La classe JadeValidator

La classe JadeValidator estende la classe XbaseJadeValidator ed è utilizzata per esprimere vincoli specifici del linguaggio non presenti nelle parser rule della grammatica.

Il primo metodo che ho mantenuto uguale alla versione precedente del linguaggio è checkBehaviourStartsWithCapital che segnala un warning in caso il carattere iniziale del nome di un elemento del SimpleBehaviour sia minuscolo. Questo tipo di vincolo è implementato anche sugli elementi AgentJade e AbstractOntology.

```
@Check
def checkBehaviourStartsWithCapital(SimpleBehaviour it) {
  if (!Character::isUpperCase(name.charAt(0))) {
    warning('Name should start with a capital',it,null)
  }
}
```

Il secondo metodo che ho mantenuto uguale alla versione precedente è checkCreateElement che controlla il tipo dell'inserimento dell'oggetto appartenente agli elementi di un'ontologia nel content del messaggio.

```
@Check
def checkCreateElement(CreateElement ce) {
  var trovato = false
  for (field : ce.fieldElements ){
    trovato = false
    var type = ce.type
    while(type != null && trovato==false){
      trovato = search(field.name,type)
      if (trovato == false)
        type = type.superType
    }
    if (trovato==false)
      warning('the field'+ field.name + 'not exist',ce,null)
  }

  for(element : ce.structureElements){
    trovato = false
```

```
    var type = ce.type
    while(type != null && trovato==false){
      trovato = search(element.name,type)
      if (trovato == false)
        type = type.superType
    }
    if (trovato==false)
      warning('the field '+ element.name + ' not exist',ce,
          null)
  }
}
```

Esiste un terzo metodo in questa classe che ho modificato e lo presenterò nel capitolo riguardante il mio linguaggio. Questo metodo è checkReceiveMessage.

3.2.4 La classe JadeTypeComputer

La classe JadeTypeComputer estende la classe XbaseTypeComputer che si occupa di determinare il tipo delle nuove espressioni di Xbase.

Il metodo computeType riceve un'espressione di Xbase e se questa è una definita all'interno del nuovo linguaggio chiama il relativo metodo _computeTypes, in caso contrario viene chiamato il metodo TypeComputer di Xbase.

```
package org.xtext.example.jade.typesystem

import ...

class JadeTypeComputer extends XbaseTypeComputer {
  override public void computeTypes(XExpression expression,
      ITypeComputationState state) {
    if(expression instanceof SendMessage) {
      _computeTypes(expression as SendMessage, state);
    } else if (expression instanceof ReceiveMessage){
      _computeTypes(expression as ReceiveMessage,state)}
    else if ((expression instanceof MyAgent)){
      _computeTypes(expression as MyAgent,state)}
    else {
      super.computeTypes(expression, state)
    }
  }
  ...
}
```

Capitolo 4

JADEL2

JADEL2, che estende JADEL1, è un linguaggio di programmazione che aiuta il programmatore durante la fase di creazione e sviluppo di un ambiente agent-based, basato su agenti software JADE.

La realizzazione del linguaggio di programmazione *JADEL2* è lo scopo di questa tesi. Rispetto al linguaggio precedente sono state rinforzate le espressioni della grammatica relative: alle ricezione di messaggi, alla classe dell'agente (inserendo una nuova funzionalità ovvero la simulazione di un automa a stati finiti di qualsiasi tipo), alle funzionalità dei behaviour, permettendo di costruire una classe completa e permettendo di costruire un behaviour di qualsiasi tipo, sia semplice che composto, rinforzando il sistema di validazione del codice che verrà generato e riuscendo a generare codice Java corretto. Analogamente a JADEL1, è stato sviluppato con l'ausilio dei plugin Xtext e Xbase.

In questo capitolo verrà trattata l'implementazione del linguaggio JADEL2 descrivendolo nelle parti che differiscono dalla precedente versione. La prima parte tratterà modifiche ed estensioni apportate alla grammatica. La seconda parte tratterà delle parti estese nella porzione di codice relativo alla traduzione e validazione del codice. L'ultima parte, infine, tratterà dei problemi legati alla generazione del codice che ho incontrato durante l'implementazione del linguaggio di programmazione. In appendice verranno presentati alcuni esempi, per mostrare il funzionamento del linguaggio e la semplificazione che porta l'utilizzo di questo linguaggio alla creazione di sistemi agent-based con anche il codice Java generato.

4.1 Grammatica

Come per il linguaggio JADEL1, la grammatica verrà presentata attraverso il linguaggio Grammar Language di Xtext.

4.1.1 L'estensione di AbstractElement

La prima modifica necessaria per poter implementare correttamente le estensioni che si vogliono apportare al linguaggio è quella di estendere la parser rule AbstractElement. All'interno della regola sono state inserite due nuove regole: ComplexBehaviour e FSMBehaviour.

```
AbstractElement :
  SimpleBehaviour | AgentJade | AbstractOntology |
  ComplexBehaviour | FSMBehaviour
;
```

4.1.2 La parser rure SimpleBehaviour

La parser rule SimpleBehaviour è stata estesa rispetto alla precedente versione, dove è stata mantenuta solo la parte relativa al tipo di behaviour esteso.

All'interno del body la regola può incorporare qualsiasi tipo di metodo specifico dei behaviour: restart, block, onStart, onEnd, done e action. Oltre a questi metodi è stata aggiunta la possibilità di dichiarare variabili e definire metodi di qualsiasi genere.

La prima estensione permette, tramite le parole chiave elencate in precedenza, di definire uno o più metodi specifici del behaviour che contengano una determinata espressione di blocco. Ognuno di questi metodi deve o meno avere un determinato valore di ritorno. É il caso dei metodi onEnd(), che ritorna un valore int, e del metodo done(), che ritorna un valore boolean. Questa regola è stata implementata con il nominativo BehaviourMethod e ritorna una espressione di Xbase.

```
BehaviourMethod returns xbase::XExpression:
  {BehaviourMethod}
  type=('restart'|'block'|'onStart'|'onEnd'|'done'|'action')
  (body=XBlockExpression)
;
```

Come accennato in precedenza è possibile definire variabili o costanti di Xbase; questa funzionalità è implementata dalla parser rule XVariableDeclaration.

É possibile anche definire un metodo di qualsiasi tipo all'interno del behaviour. É necessario innanzitutto digitare la parola chiave method, il tipo di ritorno ovvero un tipo della Java Virtual Machine, il nome del metodo, i parametri in ingresso necessari e definire, tenendo conto del parametro di ritorno, un'espressione di blocco di qualsiasi genere. Anche in questo caso la parser rule MethoDeclare ritorna un'espressione di Xbase.

```
MethodDeclare returns xbase::XExpression:
  {MethodDeclare}
  'method' returnType=JvmTypeReference? name=ValidID
   '(' (parameters+=FullJvmFormalParameter (','
     parameters+=FullJvmFormalParameter)*)? ')'
  body=XBlockExpression
;
```

La parser rule, quindi, si definisce in maniera analoga al linguaggio JADEL1, per quanto riguarda la parte relativa all'intestazione per poi definire il corpo della regola in modo completamente differente. Nel linguaggio precedente infatti, qualsiasi cosa venisse scritta all'interno dell'espressione di blocco veniva poi trasferita all'interno del metodo action() del behaviour che veniva creato.

Di seguito è presente la sintassi della parser rule SimpleBehaviour.

```
SimpleBehaviour :
  {SimpleBehaviour}
  'behaviour' name=ValidID 'is' typeSB = TypeSB '{'
  (body+=
    (BehaviourMethod|XVariableDeclaration|MethodDeclare)*)
   '}'
;
```

4.1.3 Le parser rule ComplexBehaviour e FSMBehaviour

La parser rule ComplexBehaviour rappresenta i behaviour complessi, rappresentati nel diagramma UML del capitolo 1, ovvero: SequentialBehaviour e ParallelBehaviour.

```
ComplexBehaviour:
  SequentialBehaviour | ParallelBehaviour
;
```

La parser rule FSMBehaviour è separata dagli altri ComplexBehaviour, nonostante anche essa sia un behaviour complesso, perché la composizione della classe FSMBehaviour non ha così tante analogie con le due classi ParallelBehaviour e SequentialBehaviour, infatti queste due parser rule si discostano solo per la prima parola chiave.

Un SequentialBehaviour o un ParallelBehaviour sono SimpleBehaviour, infatti è permessa la definizione, come nella parser rule SimpleBehaviour, dei metodi specifici dei behaviour, della dichiarazione di variabili e di metodi, che permettono di aggiungere una lista di behaviour, sub-behaviour che vengono eseguiti sequenzialmente o parallelamente a seconda del behaviour che si vuole implementare.

Per aggiungere un sub-behaviour è stata utilizzata la parser rule AddSubBehaviour che permette di aggiungere come sub-behaviour un qualsiasi tipo di behaviour ovvero SimpleBehaviour, ComplexBehaviour o FSMBehaviour.

```
AddSubBehaviour:
  'add' 'subbehaviour' name=[Behaviour|ValidID]
;
```

Per definire un SequentialBehaviour o un ParallelBehaviour è necessario definire il tipo di behaviour che si vuole implementare e fornire un elenco di sub-behaviour che dovranno essere definiti come spiegato in precedenza.

```
SequentialBehaviour:
  {SequentialBehaviour}
  'sequential' 'behaviour' name=ValidID '{'
  'subbehaviours' '{'
    subbehaviours += AddSubBehaviour*
  '}'
  (body+=
    (BehaviourMethod|XVariableDeclaration|MethodDeclare)*)
  '}'
;

ParallelBehaviour:
  {ParallelBehaviour}
  'parallel' 'behaviour' name=ValidID '{'
  'subbehaviours' '{'
    subbehaviours += AddSubBehaviour*
  '}'
  (body+=
    (BehaviourMethod|XVariableDeclaration|MethodDeclare)*)
  '}'
;
```

La parser rule FSMBehaviour invece ha l'obiettivo di simulare il funzionamento di un'automa a stati finiti di qualsiasi tipologia. Questa regola deve contenere l'elenco degli stati e l'elenco delle transizioni. Analogamente agli altri tipi di behaviour, anche l'FSMBehaviour può contenere le dichiarazione dei metodi specifici dei behaviour, le dichiarazione di variabili e di metodi di qualsiasi tipologia.

Per definire l'elenco degli stati è stata definita la parser rule FSMState, permette di aggiungere un nuovo stato che corrisponderà ad un behaviour di tipo qualsiasi, anche un altro FSMBehaviour. Lo stato può essere terminante; definito dalla parola chiave final. Un'automa può avere per definizione più stati finali. Per quanto riguarda lo stato iniziale invece è definito all'interno della parser rule FSMBehaviour in quanto un'automa deve avere uno e un solo stato iniziale. La parola chiave per questa proprietà è initial.

```
FSMState:
  'add'(final='final')?) name=[Behaviour|ValidID]
  'value' value=ValidID
;
```

Per quanto riguardano le transizioni è necessario definire l'id dello stato i-esimo di partenza e dello stato j-esimo di arrivo e il valore con la quale l'automa effettua una transizione di stato dallo stato i allo stato j.

```
FSMTransition:
  'add' 'from' nameFrom=ValidID 'to' nameTo=ValidID
  'with' 'value' value=XNumberLiteral?
;
```

Di seguito la sintassi completa della parser rule FSMBehaviour dove vengono utilizzati le regole descritte in precedenza.

```
FSMBehaviour:
  'fsm' 'behaviour' name=ValidID '{'
  'state' '{' 'initial' initialState = FSMState
  (state += FSMState*) '}'
  'transition' '{' (transition += FSMTransition*) '}'
  (body+=
    (BehaviourMethod|XVariableDeclaration|MethodDeclare)*)
  '}'
;
```

4.1.4 L'estensione della regola XExpressionInsideBlock

La parser rule XExpressionInsideBlock è una delle regole fondamentali della grammatica, infatti tramite questa è possibile definire quali espressioni, è un'espressione di Xbase anch'essa, possono essere contenute all'interno dell'espressione XBlockExpression ovvero l'espressione per un blocco di espressioni di qualsiasi tipologia.

Nel linguaggio JADEL2 le espressioni permesse all'interno di un blocco sono: la dichiarazione di variabili (XVariableDeclaration), espressioni di Xbase di qualsiasi tipo (come cicli, statement for e while...), l'espressione Message ovvero le due parser rule ReceiveMessage e SendMessage e la parser rule MyAgent, già definita nella versione precedente del linguaggio.

```
XExpressionInsideBlock returns xbase::XExpression:
  XVariableDeclaration | XExpression |  Message | MyAgent
;

Message returns xbase::XExpression:
  SendMessage | ReceiveMessage
;
```

4.1.5 La parser rule ReceiveMessage

La parser rule ReceiveMessage si pone l'obiettivo di ricevere e spacchettare correttamente il messaggio che arriva da un altro agente della piattaforma. La regola ritorna un'espressione di Xbase. Può essere inserita all'interno di una qualsiasi altra espressione a blocchi attraverso la parser rule XExpressionInsideBlock descritta in precedenza.

Per definire una ReceiveMessage è necessario, innanzitutto, dichiarare una variabile relativa al nome dell'ACLMessage, che verrà utilizzato e istanziato nella parte successiva. Vi è anche la possibilità di effettuare diversi filtri sul messaggio per poterlo accettare o meno. Il primo è relativo al content e conterrà una determinata Ontologia sul quale apporre i controlli.

La seconda tipologia, invece, è quella relativa ai filtri di tipo MessageTemplate descritta nel capitolo relativo a JADE. Per implementare questo filtro è stato necessario definire altre parser rule.

La parte principale è quella relativa alla parser rule Filter che contiene le regole BodyReceiveMessageElement, ANDFilter, OrFilter e NOTFilter. Questa parser rule è quella contenuta all'interno della regola ReceiveMessage.

```
Filter :
ANDFilter | ORFilter | NOTFilter | BodyReceiveMessageElement
;
```

BodyReceiveMessageElement permette di definire un controllo di qualsiasi tipo all'interno delle specifiche FIPA (performative, sender, content, ontology e language).

```
BodyReceiveMessageElement :
  {BodyReceiveMessageElement}
  'match' type=('performative'|'sender'|'content'|'ontology'
    |'language') '='
  'value' value=XStringLiteral
;
```

Le altre parser rule, invece, sono regole che ritornano a loro volta Filter, essendo la struttura dei filtri ricorsiva. Sono definite dall'operatore (or, and e not) che caratterizza la tipologia di filtro e seguite da un elenco di Filter. L'operatore not può contenere un solo elemento Filter da poter negare.

```
ANDFilter returns Filter:
  operator='and' '{' element+=Filter+ '}'
;

ORFilter returns Filter:
  operator='or' '{' element+=Filter+ '}'
;

NOTFilter returns Filter:
  operator='not' '{' notElement=Filter '}'
;
```

Dopo l'inserimento dei filtri è stata inserita un'espressione di blocco per definire il body di ReceiveMessage. Il corpo di questa espressione rappresenta il codice che verrà eseguito se tutti i filtri posti in precedenza sono stati soddisfatti.

Di seguito la sintassi dell'espressione ReceiveMessage.

```
ReceiveMessage returns xbase::XExpression :
  {ReceiveMessage}
  'receive''message''declaration' vd = XVariableDeclaration
  ('filter' 'on' 'content'
    content=[OntologyStructure|ValidID]
    'name' '=' nameOntology=ValidID)?
  ('where' filter=(Filter))?
  ('do' body= XBlockExpression)
;
```

4.2 L'implementazione

Nei prossimi paragrafi saranno descritte le classi che permettono di tradurre, compilare e validare il codice dal linguaggio JADEL2 al linguaggio Java.

4.2.1 La classe JadeJVMModelInferrer

La classe JadeJVMModelInferrer permette di tradurre i concetti definiti in linguaggio JADEL2 nel linguaggio di programmazione Java. Questa classe utilizza al suo interno altre classi che compilano le espressioni, sia di JADEL2 che di Xbase, presenti nel codice sorgente ritornando statement di codice Java unito dalla classe JadeJVMModelInferrer.

Questa classe estende la classe astratta AbstractModelInferrer che a sua volta implementa l'interfaccia IJvmModelInferrer.

Come nel linguaggio JADEL1, contiene un metodo infer() che riceve come parametri il modello definito nella grammatica, un Acceptor che permetta di accettare il codice generato e un booleano isPrelinkingPhase che indica se il metodo è stato chiamato in fase di prelinking.

All'interno di questo metodo è presente un ciclo for che scorre ciascun elemento del modello. Presenta al suo interno uno statement switch che permette di scegliere quale elemento è ora in fase di traduzione fra quelli possibili, ovvero fra le regole definite all'interno della parser rule AbstractElement.

```
package org.xtext.example.jade.jvmmodel

import ...

class JadeJvmModelInferrer extends AbstractModelInferrer {

  @Inject extension JvmTypesBuilder
  @Inject extension IQualifiedNameProvider

  def dispatch void infer(Model element,
                IJvmDeclaredTypeAcceptor acceptor,
                boolean isPrelinkingPhase) {
    for (a : element.elements) {
      switch a {
        ComplexBehaviour : {...}
        FSMBehaviour : {...}
        SimpleBehaviour : {...}
        AbstractOntology : {...}
        AgentJade : {...}
      }
    }
  }
}
```

In dettaglio verranno descritti i singoli casi dello statement switch.

L'elemento ComplexBehaviour

Il caso ComplexBehaviour, che come descritto in precedenza, contiene le regole relative ai behaviour complessi sequenziali e paralleli.

La prima istruzione è quella relativa all'acceptor che "accetta" il codice tradotto. Al suo interno viene creata la classe relativa al tipo di behaviour complesso definito estendendo SequentialBehaviour se sequenziale e ParallelBehaviour se parallelo. Il nome della classe è quello definito dal programmatore.

Successivamente, in maniera abbastanza standard per qualsiasi behaviour, viene creata una sequenza di dichiarazioni di variabili nel caso siano state definite dal programmatore nella classe creata in linguaggio JADEL2.

```
acceptor.accept(a.toClass(a.fullyQualifiedName)).
    initializeLater [
  if (ParallelBehaviour.isInstance(a))
    superTypes += a.newTypeRef('jade.core.behaviours.
        ParallelBehaviour')
  else
    superTypes += a.newTypeRef('jade.core.behaviours.
        SequentialBehaviour')
  for(bm: a.body) {
    if(XVariableDeclaration.isInstance(bm)){
      members += a.toField((bm as XVariableDeclaration).
          name, (bm as XVariableDeclaration).type) [
        initializer = (bm as XVariableDeclaration).right
      ]
    }
  }
  ...
]
```

Viene poi definito il costruttore della classe con parametro, anch'esso standard, l'agente che contiene il behaviour da lui creato all'interno dello scheduler. Viene chiamato all'interno il costruttore della superclasse come da specifiche JADE.

All'interno del costruttore è stato inserito la costruzione della struttura interna al behaviour complesso. Sono quindi state inseriti in ordine di definizione i sub-behaviour.

```
members += a.toConstructor[
  parameters += toParameter("agent", newTypeRef(a, 'jade.
      core.Agent'))
  body = [append('''
      super(agent);
    ''')
    for (sub: a.subbehaviours) {
      var String app = sub.name.name //semplificazione
      append('''
        this.addSubBehaviour(new <<app>>(agent));
      ''')
    }
  ]
]
```

In maniera altrettanto standard fra tutti i behaviour del linguaggio JADEL2, sono stati tradotti i metodi relativi ai behaviour e quelli definiti dal programmatore. In caso di mancato valore di ritorno o errato valore di ritorno, l'IDE di Eclipse visualizzerà un errore relativo al metodo che conterrà questo tipo di errore, non permettendo alla classe di generare alcun codice Java per quel determinato elemento.

```
for(bm: a.body) {
  if(MethodDeclare.isInstance(bm)){
    if((bm as MethodDeclare).returnType == null)
      (bm as MethodDeclare).setReturnType(a.newTypeRef
        (Void::TYPE))
      members += a.toMethod((bm as MethodDeclare).name,
        (bm as MethodDeclare).returnType)[
      for (p: (bm as MethodDeclare).parameters){
        parameters += p.toParameter(p.name,p.parameterType)
      }
      body = (bm as MethodDeclare).body
    ]
  }
  if(BehaviourMethod.isInstance(bm)){
    if((bm as BehaviourMethod).type.equals('done')){
      members += a.toMethod((bm as BehaviourMethod).type,
          a.newTypeRef(Boolean::TYPE))[
        body = (bm as BehaviourMethod).body
      ]
    }
    else {
      if((bm as BehaviourMethod).type.equals('onEnd')) {
        members += a.toMethod((bm as BehaviourMethod).type,
```

```
                a.newTypeRef(Integer::TYPE))[
              body = (bm as BehaviourMethod).body
            ]
          }
          else {
            members += a.toMethod((bm as BehaviourMethod).type,
                a.newTypeRef(Void::TYPE))[
              body = (bm as BehaviourMethod).body
            ]
          }
        }
      }
    }
]//acceptor
```

L'elemento FSMBehaviour

Il caso FSMBehaviour, come accennato in precedenza, contiene le regole relative al behaviour che ha il compito di simulare un'automa a stati finiti di qualsiasi tipologia. L'automa a stati finiti simulato ha n stati di cui solo uno iniziale e m stati finali; a seconda del tipo di automa a stati finiti può avere k transizioni che possono essere strettamente uguali a 2n oppure anche minori o maggiori. Come visto nella grammatica, l'FSMBehaviour ha al suo interno una lista di stati, che possono essere finali e solo uno iniziale, e una lista di transizioni. Non vi è, nel linguaggio JADEL2, alcuna relazione fra queste due liste presenti all'interno della classe tranne che le transizioni, banalmente, vanno da uno stato i-esimo a uno stato j-esimo con i e j minore o uguale ad n.

All'interno della classe JadeJvmModelInferrer, il caso FSMBehaviour può essere interpretato come un behaviour, con dichiarazioni di variabili e definizioni dei metodi specifici della classe Behaviour e metodi definiti dal programmatore, oltre che come un'automa a stati finiti.

Analogamente a tutte le traduzioni della classe vi è il primo insieme di istruzioni riguardanti l'acceptor e la creazione classe con relativa estensione della classe FSMBehaviour della libreria di JADE.

```
acceptor.accept(a.toClass(a.fullyQualifiedName)).
    initializeLater [
  documentation = a.documentation
  superTypes += a.newTypeRef('jade.core.behaviours.
      FSMBehaviour')
  ...
```

Vi è poi in maniera analoga a tutti i behaviour la definizione delle variabili.

Nel caso FSMBehaviour vengono poi dichiarate le costanti riguardanti ogni singolo stato.

```
if(a.initialState != null)
  members += a.toField("STATE_" + a.initialState.value,a.
      newTypeRef(String))[
    setFinal(true)
    setStatic(true)
    initializer = [ append (''''<<a.initialState.value>>"'''
        )]
  ]
for(s: a.state ){
  members += a.toField("STATE_" + s.value,a.newTypeRef(
      String))[
    setFinal(true)
    setStatic(true)
    initializer = [ append (''''<<s.value>>"'''')]
  ]
}
```

Viene poi inserito il costruttore, che per default riceve come parametro l'agente che lo ha aggiunto allo scheduler dei behaviour. Seguono poi gli insiemi di registrazione di stati e transizioni alla classe.

```
members += a.toConstructor[
  parameters += toParameter("agent",newTypeRef(a,'Agent'))
  body = [  //semplificazione
    append(''' super(agent);
      this.registerFirstState(new
        <<a.initialState.name.name>>(agent),
        STATE_<<a.initialState.value.toUpperCase>>);
      <<if(a.initialState.final != null)
        '''this.registerLastState(new
         <<(a.initialState.name as FSMBehaviour).name>>(
             agent),
         STATE_<<a.initialState.value.toUpperCase>>);'''>>
      <<for(s: a.state)
        if(s.final == null) '''
          this.registerState(new <<s.name.name>>(agent),
              STATE_<<s.value.toUpperCase>>);'''
        else '''
          this.registerLastState(new <<s.name.name>>(agent),
              STATE_<<s.value.toUpperCase>>);'''
       ''')
      append('''<<for(t: a.transition){'''
          this.registerTransition(STATE_<<t.nameFrom.
              toUpperCase>>,STATE_<<t.nameTo.toUpperCase
              >>,<<(t.value as XNumberLiteral).value>>);'''
}>> ''') ]
```

Infine vi è la traduzione dei metodi relativi ai behaviour e quelli definiti dal programmatore.

L'elemento SimpleBehaviour

Il caso SimpleBehaviour come accennato in precedenza è stato già implementato nella versione precedente del linguaggio. Nella mia versione però ho aggiunto, oltre alla dichiarazione della classe e super-classe, le dichiarazioni di variabili e dei metodi relativi alla classe Behaviour e quelli definiti dal programmatore.

L'elemento AbstractOntology

Il caso AbstractOntology, come definito nella versione precedente, può essere un OntologyElement o una OntologyStructure.

Per quanto riguarda il tipo OntologyElement: ho modificato l'aggiunta dei campi relativi all'elemento che viene descritta nella classe definita dal programmatore e la definizione dei metodi setter e getter per il campo definito

```
for (field : a.body){
  members += field.toField(field.name,field.ftype)
}
for (field : a.body){
  members += field.toGetter(field.name.toFirstUpper,field.
      ftype)
  members += field.toSetter(field.name.toFirstUpper,field.
      ftype)
}
```

Per quanto riguarda il tipo OntologyStructure, invece, la prima estensione è stata quella relativa alla definizione delle costanti relative ai campi degli OntologyElement definiti nella struttura.

```
for (f: a.elements) {
  members += a.toField(f.name.toUpperCase,a.newTypeRef(
      typeof(String))) [
    static = true
    final = true
    initializer = [append('''"<<f.name.toFirstUpper>>"''')]
  ]
  for(e: f.body) {
    members += a.toField(f.name.toUpperCase + "_" + e.name.
        toUpperCase,a.newTypeRef(typeof(String)))[
      static = true
      final = true
      initializer = [append('''"<<f.name.toFirstUpper>><<e.
          name.toFirstUpper>>"''')]
    ]
  }
}
```

La seconda estensione è quella che riguarda il costruttore dove viene inserito un costrutto try-catch che conterrà la definizione e l'aggiunta dei parametri relativi all'ontologia.

```
members += a.toConstructor[
  documentation = '''...'''
  parameters += toParameter("base", newTypeRef(a, 'jade.
      content.onto.Ontology'))
  body = [
  append('''
    super(NAME_ONTOLOGY, jade.content.onto.BasicOntology.
        getInstance());

    //define ontology parameter
    try{
    <<for(e: a.elements){
      '''add(new jade.content.schema.<<e.type.toFirstUpper>>
          Schema(<<e.name.toUpperCase>>), <<e.name.
          toFirstUpper>>.class);'''
    }>>
    <<for(e: a.elements){
      '''jade.content.schema.<<e.type.toFirstUpper>>Schema
          <<e.type.toLowerCase.charAt(0)>>s = (jade.content
          .schema.<<e.type.toFirstUpper>>Schema) getSchema
          (<<e.name.toUpperCase>>);
      <<for(f: e.body){
        '''<<e.type.toLowerCase.charAt(0)>>s.add(<<e.name.
            toUpperCase>>_<<f.name.toUpperCase>>,
          (jade.content.schema.PrimitiveSchema) getSchema(
              jade.content.onto.BasicOntology.<<returnType(
              f.ftype.identifier)>>));'''
      }>>
      }>>
      }catch(jade.content.onto.OntologyException e){
        e.printStackTrace();
      }
    ''')
]
```

L'elemento AgentJade

Il caso AgentJade riguarda la traduzione della classe relativa all'agente. Alla versione del linguaggio JADEL1, che è stata descritta nel capitolo precedente, è stata aggiunta al metodo setup() la parte relativa all'aggiunta dei behaviour.

```
members += a.toMethod("setup",a.newTypeRef(Void::TYPE))[
  body = [   //semplificazione
    if (a.ontology != null){
      append('''manager.registerOntology(ontology);''')}
      append('''
      <<FOR b : a.behaviours>>
        <<b.type.name>> <<b.name>> = new <<b.type.name>>(
            this);
        addBehaviour(<<b.name>>);
      <<ENDFOR>>
      ''')
    }
  ]
]
```

4.2.2 La classe JadeCompiler

La classe JadeCompiler, descritta già nel capitolo precedente, contiene un metodo doInternalJavaStatement() che ricevuto come parametro l'espressione da compilare ritorna all'interno dell'albero passato come parametro il codice generato.

Il metodo contiene al suo interno uno statement switch dove sono definite le varie espressioni definite nel linguaggio. In questo caso sono: MyAgent, SendMessage e ReceiveMessage. Le prime due sono già state definite nel linguaggio precedente.

L'espressione ReceiveMessage invece è stata completamente implementata all'interno del linguaggio JADEL2.

La prima parte della compilazione riguarda la definizione della classe MessageTemplate che è necessaria per il controllo del contenuto del messaggio ricevuto e verrà successivamente utilizzato per controllare l'esecuzione del blocco interno allo statement do dell'espressione ReceiveMessage.

```
ReceiveMessage : {
  var vds = expr.vd as XVariableDeclaration
  if (expr.filter != null){
    append (''' MessageTemplate mt = null; ''')
    append (''' mt = ''' )
    append('''<<createTemplate(expr.filter , it)>>;''')
    append(''' jade.lang.acl.ACLMessage <<vds.name>> = this.
        myAgent.receive(mt); ''')
  }
  else {
    append('''jade.lang.acl.ACLMessage <<vds.name>> = this.
        myAgent.receive()''')
  }
  ...
}
```

Attraverso il metodo createTemplate() viene costruito il pattern del controllo del messaggio. É un metodo ricorsivo in quanto la struttura della parser rule Filter è un albero. Il metodo viene iterato finché non si raggiunge una struttura elementare, ovvero un elemento relativo alla classe BodyReceiveMessageElement.

```
def private void createTemplate(Filter f, ITreeAppendable it
    ){
  if(f.operator == null)
    getTemplate((f as BodyReceiveMessageElement),it)
  else{
    if(f.operator.equals("not")){
      append('''MessageTemplate.not(''')
      createTemplate(f.notElement,it)
      append(''')''')
    }
    else{
      append('''MessageTemplate.<<f.operator.toLowerCase>>('
          '')
      for(e: f.element){
        createTemplate(e,it)
        if(f.element.indexOf(e) < f.element.size - 1)
          append(''',''')
      }
      append(''')''')
    }
  }
}
```

Nella traduzione della regola ReceiveMessage, dopo la costruzione del MessageTemplate, viene creato attraverso la ricezione del messaggio l'elemento relativo alla classe ACLMessage.

Vengono poi creati gli elementi dell'ontologia e viene implementato il controllo relativo all'esecuzione dello statement do relativo alla parser rule ReceiveMessage.

```
if(expr.filter != null && expr.content != null){
  append('''jade.content.ContentElement ce = null;''')
  append('''ce = this.myAgent.getContentManager().
       extractContent(<<vds.name>>);''')
  append('''switch(ce) {''')
  append('''<<for(oe: (expr.content as OntologyStructure).
       elements){
  if(oe.type.equals("predicate")){
    '''case <<oe.name>>:
    <<oe.name>> <<oe.name.toLowerCase>> = (<<oe.name>>) ce;
    <<for(f: oe.body){
      '''<<f.ftype.identifier>> <<f.name>> = <<oe.name.
           toLowerCase>>.get<<f.name.toFirstUpper>>();'''
     }>>
    break;'''
    }>>''')
  append('''}''')
}
append('''if(mt.match(<<vds.name>>)) ''')
super.doInternalToJavaStatement(expr.body,it,isReferenced)
append('''} ''')
append('''else { block(); }''')
```

4.2.3 La classe JadeValidator

La classe JadeValidator è utilizzata per esprimere vincoli specifici interni al linguaggio non presente nelle parser rule della grammatica o nella traduzione del codice.

Nella versione del linguaggio JADEL2 è stato inserito il controllo relativo alla performativa, per permettere al programmatore di inserire valori della performativa solo di valore INFORM o REQUEST all'interno dela regola relativa al ReceiveMessage.

```
if(!matchPerformative(it.filter))
  error("Only value 'INFORM' or 'REQUEST' is permitted ",it,
       null,1)
```

Il metodo matchPerformative() che riceve come parametro un elemento Filter controlla ricorsivamente che il valore della perfomativa coincida con i valori INFORM o REQUEST.

```
def private boolean matchPerformative(Filter f){
  if(f.operator == null){
    if((f as BodyReceiveMessageElement).type.equals("
        performative")) {
      var br = (f as BodyReceiveMessageElement)
      if(!(br.value as XStringLiteral).value.equals("INFORM"
          ) && !(br.value as XStringLiteral).value.equals("
          REQUEST"))
        return false
    }
  }
  else{
    if(f.operator.equals("NOT"))
      return matchPerformative(f.notElement)
    else{
      for(e: f.element){
        if(!matchPerformative(e))
          return false
      }
    }
  }
  return true
}
```

4.3 La generazione del codice

Il problema relativo alla generazione del codice è stato il primo problema riscontrato sin dall'inizio del mio lavoro sul progretto di tesi. Il primo problema che ho riscontrato nella generazione del codice è stata quella dell'assenza di registrazioni all'interno della classe JadeRuntimeModule. La classe runtime module è di importanza fondamentale per la scrittura di linguaggi di programmazione tramite Xtext in quanto la classe JadeRuntimeModule permette di registrare i componenti utilizzati al momento di runtime.

Se, banalmente, all'interno di questa classe non è presente alcuna registrazione il framework di Xtext non riuscirà a rintracciare le classi necessarie per la compilazione e validazione del codice.

Di seguito le registrazioni relative all'interno della classe JadeRuntimeModule.

```
package org.xtext.example.jade;

import ...

public class JadeRuntimeModule extends org.xtext.example.
    jade.AbstractJadeRuntimeModule {
  @SingletonBinding(eager = true)
  public Class<? extends XbaseJavaValidator>
      bindAbstractJadeValidator() {
    return JadeValidator2.class;
  }

  @Override
  public Class<? extends ITypeComputer> bindITypeComputer()
      {
    return JadeTypeComputer.class;
  }

  public Class<? extends XbaseCompiler> bindXbaseCompiler(){
    return JadeCompiler.class;
  }
}
```

Si può notare che all'interno della classe JadeRuntimeModule non è presente la registrazione della classe JadeJVMModelInferrer, in quanto la classe viene chiamata automaticamente dal framework Xtext in qualsiasi caso.

Prima di utilizzare la classe Inferrer ho provato a utilizzare la classe JadeGenerator. Questa è concettualmente più semplice della classe JadeJVMModelInferrer, infatti essa genera il codice relativo al modello ricevuto dal framework Xtext. La classe JadeGenerator però non comunica correttamente le espressioni di Xbase alla classe JadeCompiler causando problemi per le espressioni standard del linguaggio di Xbase.

All'interno della classe JadeJVMModelInferrer nella versione del linguaggio JADEL1, non veniva generato codice semanticamente e sintatticamente corretto. Manca ad esempio la parte di traduzione della parser rule AbstractOnotology ovvero la parte di definizione dei campi e metodi setter e getter nella regola OntologyElement e manca la definizione del vocabolario e aggiunta degli elementi nella regolo OntologyStructure.

Conclusioni

Il lavoro di questa tesi ha avuto come scopo principale quello di implementare un linguaggio di programmazione per agenti software JADE utilizzando Xtext. Il lavoro è stato strutturato in tre fasi principali.

La prima fase è stata quella dello studio del framework JADE andando a studiarne l'utilizzo e provandone le varie proprietà. É stato particolarmente interessante lavorare con un'ambiente agent-based e studiarne le caratteristiche e le possibili applicazioni. La prova a livello di programmazione si è concentrata sulla scrittura di due agenti che comunicano fra loro utilizzando tutti i concetti principali come agenti, behaviour e ontologia.
La seconda fase è stata quella dello studio del linguaggio JADEL1 e del plugin per Eclipse Xtext. Il primo contatto con questa realtà non è stata delle più favorevoli, infatti il primo problema riscontrato è stato quello relativo alla generazione di codice Java. La generazione di codice corretto è stato poi il passo successivo per far si che il codice generato attraverso Xtext sia il più simile possibile al codice corretto per l'implementazione di un sistema multi-agente.
Lo studio di Xtext e del linguaggio di Xbase è stata una parte molto importante di questa fase perché senza di questa non avrei potuto comprendere completamente cosa era stato implementato nel linguaggio JADEL1.
La terza fase di questo lavoro di tesi è stata quella dello studio delle nuove funzionalità da inserire nel linguaggio JADEL2 e la successiva implementazione, che però senza un'attenta fase di studio non sarebbe potuta esistere.

Il lavoro di questa tesi può essere ampliamente esteso perchè l'attuale implementazione si è focalizzata sulla generazione del codice, sulla comunicazione fra agenti e sui behaviour complessi.

Appendice A

Esempi di codice

In questa appendice verranno mostrati alcuni esempi di programmi scritti in linguaggio JADEL2 con il relativo codice generato.

A.1 Ping-Pong

Viene descritto un programma relativo alla comunicazione fra due agenti: Ping e Pong. Attraverso le nuove feature inserite nel linguaggio JADEL2 ora si può combinare diverse soluzioni per la comunicazione fra i due agenti. É stato descritto solo la parte dell'agente Ping essendo l'agente Pong analogo.

Di seguito la parte di codice in linguaggio JADEL2 relativa all'ontologia dell'agente Ping.

```
package PingPong

import jade.lang.acl.ACLMessage

concept Persona {
        field String nome
        field int eta
}

predicate Proprietario {
        field String persona
        field int anni
}

ontology OntologyPing {
        add Persona
        add Proprietario
}
```

Di seguito il codice generato dell'ontologia dell'agente Ping.

```
package PingPong;
import jade.content.onto.Ontology;
@SuppressWarnings("all")
public class OntologyPing extends Ontology {
  public static String ONTOLOGY_NAME = "OntologyPing";
  private final static String PERSONA = "Persona";
  private final static String PERSONA_NOME = "PersonaNome";
  private final static String PERSONA_ETA = "PersonaEta";
  private final static String PROPRIETARIO = "Proprietario";
  private final static String PROPRIETARIO_PERSONA = "
      ProprietarioPersona";
  private final static String PROPRIETARIO_ANNI = "
      ProprietarioAnni";
  private static Ontology theInstance = new OntologyPing (
      BasicOntology.getInstance());
  public static Ontology getIstance() {
    return theInstance;
  }
  public OntologyPing(final Ontology base) {
    super(NAME_ONTOLOGY, jade.content.onto.BasicOntology.
        getInstance());
    //define ontology parameter
    try{
        add(new jade.content.schema.ConceptSchema(PERSONA),
            Persona.class);
        add(new jade.content.schema.PredicateSchema(
            PROPRIETARIO), Proprietario.class);
        jade.content.schema.ConceptSchema cs = (jade.content
            .schema.ConceptSchema) getSchema(PERSONA);
        cs.add(PERSONA_NOME,
          (jade.content.schema.PrimitiveSchema) getSchema(
              jade.content.onto.BasicOntology.STRING));
        cs.add(PERSONA_ETA,
          (jade.content.schema.PrimitiveSchema) getSchema(
              jade.content.onto.BasicOntology.INTEGER));
        jade.content.schema.PredicateSchema ps = (jade.
            content.schema.PredicateSchema) getSchema(
            PROPRIETARIO);
        ps.add(PROPRIETARIO_PERSONA,
          (jade.content.schema.PrimitiveSchema) getSchema(
              jade.content.onto.BasicOntology.STRING));
        ps.add(PROPRIETARIO_ANNI,
          (jade.content.schema.PrimitiveSchema) getSchema(
              jade.content.onto.BasicOntology.INTEGER));
    }catch(jade.content.onto.OntologyException e){
        e.printStackTrace();}}
}
```

Il codice relativo al behaviour PingSendReceiveBehaviour in linguaggio JADEL2.

```
behaviour PingSendReceiveBehaviour is CyclicBehaviour {
    var int conta = 0
    method boolean inviaMessaggio(){
        send message msg {
          content= "I'm Ping"
          sender= "Ping"
          ontology= "OntologyPing"
          performative= INFORM
          receivers {
              receiver = "Pong"
          }
        }
        return true
    }
    onStart {
        inviaMessaggio
    }
    action {
        receive message declaration val ACLMessage msg =
            null
        filter on content OntologyPing name = ontoPing
        where or {
          match performative = value "INFORM"
          or {
                  match sender = value "Pong"
                  not {
                   match ontology = value "onto"
                }
          }
        }
        do {
          if(conta < 10) {
              conta = conta + 1
                      inviaMessaggio
          }
          else{
              this.^block
          }
        }
    }
}
```

Il codice generato del behaviour PingSendReceiveBehaviour.

```
package PingPong;

import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;

@SuppressWarnings("all")
public class PingSendReceiveBehaviour extends
    CyclicBehaviour {
  private int conta = 0;
  public PingSendReceiveBehaviour(final Agent agent) {
    super(agent);

  }
  public boolean inviaMessaggio() {
    {
      jade.lang.acl.ACLMessage msg = new jade.lang.acl.
          ACLMessage(jade.lang.acl.ACLMessage.INFORM);
      msg.setSender(new jade.core.AID ("Ping",jade.core.AID.
          ISLOCALNAME));
      msg.setContent("I'm Ping");
      msg.setOntology(OntologyPing);
      msg.addReceiver(new jade.core.AID ("Pong",jade.core.
          AID.ISLOCALNAME));
      this.myAgent.send(msg);
      return true;
    }
  }
  public void onStart() {
    this.inviaMessaggio();
  }
  public void action() {
    MessageTemplate mt = null;
    mt = MessageTemplate.or(MessageTemplate.
        MatchPerformative(ACLMessage.INFORM),
    MessageTemplate.or(MessageTemplate.MatchSender(Pong),
    MessageTemplate.not(MessageTemplate.MatchOntology(onto))
        ));
    jade.lang.acl.ACLMessage msg = this.myAgent.receive(mt);
    if (msg!=null){
      jade.content.ContentElement ce = null;
      ce = this.myAgent.getContentManager().extractContent(
          msg);
      switch(ce) {
      case Proprietario:
        Proprietario proprietario = (Proprietario) ce;
        java.lang.String persona =proprietario.getPersona();
        int anni = proprietario.getAnni();
        break;
```

```
        }
      if(mt.match(msg))
      boolean _lessThan = (this.conta < 10);
      if (_lessThan) {
        int _plus = (this.conta + 1);
        this.conta = _plus;
        this.inviaMessaggio();
      } else {
        this.block();
      }}
    else { block(); }
  }
}
```

Il codice in JADEL2 della dichiarazione dell'agent Ping.

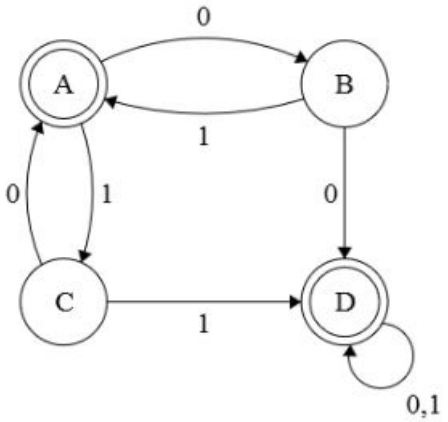
```
agent PingAgent {
        add behaviour psrb type PingSendReceiveBehaviour
}
```

Il codice generato dell'agente Ping.

```
package PingPong;

import jade.content.ContentManager;
import jade.core.Agent;

@SuppressWarnings("all")
public class PingAgent extends Agent {
  protected Agent myAgent =  this ;

  private ContentManager manager = (ContentManager)
      getContentManager();

  public void setup() {

    PingSendReceiveBehaviour psrb = new
        PingSendReceiveBehaviour(this);
    addBehaviour(psrb);

  }
}
```

A.2 FSMBehaviour

Viene descritto un programma relativo alla simulazione dell'automa a stati finiti descritto in figura.



Di seguito il codice, del behaviour, in linguaggio JADEL2.

```
package FSM
behaviour Behaviour is OneShotBehaviour{
        onEnd{
                var int returnValue = 1
                if(Math.random < 0.5)
                        returnValue = 0
                return returnValue
        }
}
fsm behaviour FSMBehaviour{
        state {
                initial add final Behaviour value A
                add Behaviour value B
                add Behaviour value C
                add final Behaviour value D
        }
        transition {
                add from A to B with value 0
                add from B to A with value 1
                add from A to C with value 1
                add from C to A with value 0
                add from B to D with value 0
                add from C to D with value 1
                add from D to D with value 1
                add from D to D with value 0
        }
}
```

Il codice dell'agente che ha aggiunto l'automa a stati finiti.

```
agent FSMAgent{
        add behaviour fsmb type FSMBehaviour
}
```

Il codice generato relativo al behaviour FSMBehaviour.

```
package FSM;

import jade.core.Agent;

@SuppressWarnings("all")
public class FSMBehaviour extends jade.core.behaviours.
    FSMBehaviour {
  private final static String STATE_A = "A";

  private final static String STATE_B = "B";

  private final static String STATE_C = "C";

  private final static String STATE_D = "D";

  public FSMBehaviour(final Agent agent) {

    super(agent);

    this.registerFirstState(new Behaviour(agent),STATE_A);
    this.registerLastState(new Behaviour(agent),STATE_A);
    this.registerState(new Behaviour(agent),STATE_B);
    this.registerState(new Behaviour(agent),STATE_C);
    this.registerLastState(new Behaviour(agent),STATE_D);

    this.registerTransition(STATE_A,STATE_B,0);
    this.registerTransition(STATE_B,STATE_A,1);
    this.registerTransition(STATE_A,STATE_C,1);
    this.registerTransition(STATE_C,STATE_A,0);
    this.registerTransition(STATE_B,STATE_D,0);
    this.registerTransition(STATE_C,STATE_D,1);
    this.registerTransition(STATE_D,STATE_D,1);
    this.registerTransition(STATE_D,STATE_D,0);

  }
}
```

A.3 ParallelBehaviour e SequentialBehaviour

Viene ora mostrato un programma che utilizza i behaviour complessi paralleli e sequenziali. Di seguito il codice in linguaggio JADEL2.

```
package Sequential

behaviour Behaviour is OneShotBehaviour {
        onEnd {
                return 0
        }
}

sequential behaviour SequentialBehaviour {
        subbehaviours {
                add subbehaviour Behaviour
                add subbehaviour Behaviour
        }
}

parallel behaviour ParallelBehaviour {
        subbehaviours {
                add subbehaviour SequentialBehaviour
                add subbehaviour Behaviour
                add subbehaviour SequentialBehaviour
                add subbehaviour Behaviour
        }
}

agent SequentialAgent {
        add behaviour bp type ParallelBehaviour
        add behaviour bs type SequentialBehaviour
}
```

Di seguito viene mostrato il codice generato relativo al behaviour SequentialBehaviour .

```
package Sequential;
import jade.core.Agent;

@SuppressWarnings("all")
public class SequentialBehaviour extends jade.core.
    behaviours.SequentialBehaviour {
  public SequentialBehaviour(final Agent agent) {
    super(agent);
    this.addSubBehaviour(new Behaviour(agent));
    this.addSubBehaviour(new Behaviour(agent));
  }
}
```

Di seguito viene mostrato il codice generatorelativo al behaviour ParallelBehaviour .

```
package Sequential;

import jade.core.Agent;

@SuppressWarnings("all")
public class SequentialBehaviour extends jade.core.
    behaviours.SequentialBehaviour {
  public SequentialBehaviour(final Agent agent) {
    super(agent);
    this.addSubBehaviour(new Behaviour(agent));
    this.addSubBehaviour(new Behaviour(agent));

  }
}
```

Bibliografia

[1] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa
White Paper Of Jade
http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf

[2] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa
Jade programmer's guide
http://jade.tilab.com/doc/programmersguide.pdf

[3] G. Caire
Jade programming for beginners
http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf

[4] F. Bellifemine, G. Caire, T. Truccoi, G. Rimassa, R. Mungenast
Jade Administrator's Guide
http://jade.tilab.com/doc/administratorsguide.pdf

[5] G. Caire, D. Cabanillas
Application-Defined Content Languages and Ontologies
http://jade.tilab.com/doc/applicationdefined.pdf

[6] Xtext Home Page
http://www.eclipse.org/Xtext/index.html

[7] Xtext Documentation
http://www.eclipse.org/Xtext/documentation.html

[8] Xbase Documentation
http://www.eclipse.org/Xtext/documentation.html#xbaseJavaReferences

[9] F. Rastelli
Un Linguaggio di Programmazione per Agenti JADE che Estende Xbase