ASTRA Agent Programming Language and comparison with JADE

Giulia Lucchi Luca Polverelli

Relazione Sistemi Distribuiti

Anno 2016/2017

Ingegneria e Scienze Informatiche, Università di Bologna

Outline

- ASTRA Agent Programming Language
- ASTRA and JADE
- PingPong Example

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent System
  - EIS Environment
- ASTRA and JADE
- PingPong Example

ASTRA Agent Programming Language: Introduction

ASTRA, an implementation of AgentSpeak(TR), is a logic-based agent programming language that combines AgentSpeak(L) with Teleo-Reactive functions. [Russell, 2015]

- A Teleo-Reactive (TR) program is one whose actions are all directed toward achieving goals (the Teleo part), in a way appropriate to the current, perceived state of the environment (the Reactive part). [Nilsson, 1994]

ASTRA is an Agent Programming Language for people who wish to create intelligent distributed/concurrent systems that are built on the Java platform. It is designed to be easy to learn and familiar to developers who are experienced in using mainstream Object-Oriented Programming Languages. [ASTRA, 2017]

Why use ASTRA?

- to develop multi-agent programs,
- to do this using a familiar and simple to use language,
- Java-style syntax,
- typed language,
- rapidly prototype solutions to problems using an agent programming language,
- possibility to integrate agent technology into your existing platform without the need to redesign the whole system.

ASTRA Agent Programming Language

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
    - AgentSpeak(L)
    - AgentSpeak(L) programming constructs
    - Programming constructs in ASTRA
    - Differences between ASTRA and AgentSpeak(L)
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent System
  - EIS Environment
- ASTRA and JADE
- PingPong Example

From AgentSpeak(L) to ASTRA: AgentSpeak(L) I

ASTRA is based upon AgentSpeak(L) in that it provides all of the same basic functionality as AgentSpeak(L), but then augments this basic functionality with a range of additional features that result in a more practical Agent Programming Language.

AgentSpeak(L) is an agent programming language, that is based on **Belief-Desire-Intention (BDI)** theory.

Belief-Desire-Intention (BDI) [ASTRA, 2017]

The theory BDI models rational decision making as a process of state manipulation. The current state of the world can be represented as a set of **beliefs** and the ideal state of the world can be represented as a set of **desires**. The theory maps out how a rational decision maker achieves its desires – that is, how it changes the world so that its desires become its beliefs. The way in which BDI theory achieves this is by adding a third state – **intentions** – defined as a subset of desires that the decision maker is committed to achieving. The decision-makers are resource bounded entities and may not be able to achieve all their desires due to a lack of sufficient resources. As a result, they must select a subset of those desires that they "believe" they are capable of realising given their resource constraints – these are their intentions. Once selected, the decision maker attempts to make its intentions into beliefs by identifying and following an appropriate course of action. The identification of this course of action is known as a **plan**.

From AgentSpeak(L) to ASTRA: AgentSpeak(L) II

AgentSpeak(L) adopts three refinements to BDI theory:

- The plan is based on selection of the plan from a library of pre-written plans;
- The concept of a goal is often introduced as a replacement for desires;
  - Goals are defined as a mutually consistent set of desires.
  - AgentSpeak(L) adopts goals as the representation of future state.
- Intentions are represented as the plan that will bring about that state (intention-to-do).

ASTRA Agent Programming Language

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
    - AgentSpeak(L)
    - AgentSpeak(L) programming constructs
    - Programming constructs in ASTRA
    - Differences between ASTRA and AgentSpeak(L)
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent System
  - EIS Environment
- ASTRA and JADE
- PingPong Example

From AgentSpeak(L) to ASTRA: AgentSpeak(L) programming constructs I

AgentSpeak(L) defines a set of programming constructs, encoded using a specific syntax and supported by a corresponding interpreter algorithm that is based on a Belief-Goal-Intention(to-do) model of rational decision making. The core constructs provided are:

Beliefs

Predicate formulae representing facts about the state of the agents environment.

Goals

Predicate formulae (prefixed with a bang operator (!)) identify what the agent wants to do. Goals are not stored explicitly in the agent state, instead, they are declared as required and mapped contextually to a behaviour that will realise the goal. The mapping is achieved through the use of events and associated event handler, known as plan rules.

From AgentSpeak(L) to ASTRA: AgentSpeak(L) programming constructs II

Events

Events drive the behaviour of an agent. Internally, the agent contains an event queue. Each iteration of the interpreter, one event is selected from the event queue and processed through contextual mapping of the event to an event handler (plan rule). AgentSpeak(L) includes events for: the adoption of new beliefs, the retraction of existing beliefs, and the adoption of goals.

Plan rules

Plan Rules define the core behaviours of the agent, contextually mapping those behaviours to the events that trigger them. Behaviours are specified as a sequence of plan operators; which support the following functionality: belief adoption/retraction, sub goal adoption, belief querying and private actions.

ASTRA Agent Programming Language

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
    - AgentSpeak(L)
    - AgentSpeak(L) programming constructs
    - Programming constructs in ASTRA
    - Differences between ASTRA and AgentSpeak(L)
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent System
  - EIS Environment
- ASTRA and JADE
- PingPong Example

From AgentSpeak(L) to ASTRA: Programming constructs in ASTRA

Beliefs, Goals and Events

In terms of **beliefs**, **goals**, and **events**, the main difference is that terms and variables are typed. The type system employed by ASTRA is based on the Java type system.

ASTRA does also include other event types beyond the ones defined in AgentSpeak(L). These additional event types correspond to:

- message events (receipt of messages),
- EIS and CArtAgO events (environment events) [EIS][CArtAgO],
- ACRE events (conversation management events)[ACRE].

This use of an expanded set of event types is one of the distinguishing features of ASTRA when compared with AgentSpeak(L).

Plan Rules

In ASTRA, **plan rules** are defined using a different syntax to that used in AgentSpeak(L). Specifically, the syntax adopted in ASTRA attempts to be closer to the syntax used in languages like Java and C – the plan body part of a rule is implemented as a code block which itself is made up of a sequence of primitive statements, control flow statements, or sub-blocks.

ASTRA Agent Programming Language

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
    - AgentSpeak(L)
    - AgentSpeak(L) programming constructs
    - Programming constructs in ASTRA
    - Differences between ASTRA and AgentSpeak(L)
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent Systems
  - EIS Environment
- ASTRA and JADE
- PingPong Example

ASTRA Agent Programming Language: Differences between ASTRA and AgentSpeak(L)

ASTRA is different from AgentSpeak(L) in the following ways:

- agent programs are organised into agent classes that can be extended based on a multiple inheritance model
- the primitive actions used must be explicitly declared in the agent program
- ASTRA programs can reference java classes, so support needs to be provided for providing fully qualified class names
- in addition to plan rules, ASTRA programs include partial plans (named plan bodies) in order to improve code/class reusability.
- ASTRA aims to be familiar to developers who know mainstream programming languages, and in particular Java.

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent Systems
  - EIS Environment
- ASTRA and JADE
- PingPong Example

ASTRA Agent Programming Language: ASTRA Control Flow I

Events and Behaviours control flow

ASTRA is an event-based concurrent programming language where global state is modeled using predicate logic and behaviour is encapsulates within a set of contextual event rules.
ASTRA agents react to events. Events arise from changes in the environment, receipt of messages, or as a result of the invocation of a sub goal. Events trigger behaviours that define how the agent should respond to the event. Events can be overloaded and more than one behaviours can be linked to an event. The selection of a specific behaviour is based on rule-order and an associated context that defines when the behaviour should be considered. Behaviours are synonymous with methods/procedures and are encoded in plan rules that combine a triggering event, a context, and the implementation of the behaviour. When an agent selects a behaviour to handle an event, an intention is created. With the exception of sub goal events, all events generate a new intention. [ASTRA, 2017]

Intentions control flow

Intentions are the concept in ASTRA that correlates with method/procedure execution. Intentions are mapped to threads. An intention is basically a stack where is maintained the current state of the intention. Each iteration of the ASTRA interpreter, one intention is selected and the next statement in the behaviour is executed. When there is only a single intention, this means that the behaviour is executed in isolation. When there are multiple intentions, the execution is interleaved. ASTRA adopts a *round-robin policy*, ensuring that all intentions receive an equal opportunity for execution. [ASTRA, 2017]

ASTRA Agent Programming Language: ASTRA Control Flow II

Core interpreter cycle:

1. select an event *e* from the agents event queue.
2. match the event to a plan rule *p* whose triggering event matches *e* and whose context is satisfied.
3. if the event is a belief adoption / retraction event,
   then create a new intention to process the behaviour specified in *p*,
   else update the intention that generated *e* to also process the behaviour specified in *e*.
4. select one intention *i* and execute its next step.
5. return to 1.

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
  - ASTRA Control Flow
  - ASTRA Program Structure
    - ASTRA Examples
  - Multi-Agent Systems
  - EIS Environment
- ASTRA and JADE
- PingPong Example

ASTRA Agent Programming Language: ASTRA Program Structure I

ASTRA programs are a little more structured than AgentSpeak(L) programs.

```
package path.to.folder;

import java.lang.Object;

agent MyAgent {
    // code goes here
}
```

ASTRA program basic structure

Only one class program is permitted per file and each file must have the same name as the agent class and the .astra file extension must be used. For example, the above program must be in a file called MyAgent.astra.

ASTRA is designed to make the link to Java code easy to understand. The language comes with a number of standard libraries, in Javadoc format, that provide commonly used sets of functionality. [API ASTRA]

ASTRA Agent Programming Language: ASTRA Program Structure II

The body of an agent program is made up of some components identified by unique keyword.

KEYWORD	DESCRIPTION	EXAMPLE
module	Links an agent class to a Java module.	`module Console console;`

types	Declares valid formulae	`types eg {`
		`  formula is(string,string);`
		`}`

initial	Declares initial state of the agent	`initial state("alive");` `initial is("rem", "happy"), !init();`

rule	Declares a plan rule	`rule +!init() {`
		`  console.println("hello");`
		`}`

plan	Declares a partial plan	`plan myplan() {`
		`  console.println("plan");`
		`}`

function	Declares a teleo-reactive function	`function myfunction() {`
		`  true -> console.println("function")`
		`}`

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
  - ASTRA Control Flow
  - ASTRA Program Structure
    - ASTRA Examples
  - Multi-Agent Systems
  - EIS Environment
- ASTRA and JADE
- PingPong Example

ASTRA program structure: ASTRA Examples I

At their core, ASTRA programs are executed in exactly the same way as AgentSpeak(L) programs. The main differences come in the syntax, and the ancillary supports, such as modules, multiple inheritance, and typing.

```
!init().

+!init() <- println(hello world).
```

AgentSpeak(L) Hello World example

```
agent Hello {
      module Console console;

      initial !init();

      rule +!init() {
            console.println("hello world");
      }
}
```

ASTRA Hello World example

The !init() statement declares a goal. This goal results in a goal adoption event being added to the agents event queue. The +!init() statement is a plan rule. This rule is designed to handle the goal adoption event generated by the !init() statement. To specify a goal adoption event, the goal is simply prefixed by a + operator. The behaviour contains a single plan operator: a private action that prints out the argument to the console.

ASTRA program structure: ASTRA Examples II

```
agent QueryExample {
    module Console C;

    types eg {
        formula is(string, string);
    }

    initial !init();
    initial is("rem", "happy");

    rule +!init() {
        C.println("starting");
        query(is("rem", "happy"));
        C.println("first hurdle passed");
        query(is("rem", "sad"));
        C.println("ending");
    }
}
```

ASTRA Belief Queries example

This example shows an ASTRA agent that uses the belief query plan operator. It also includes an initial belief is("rem", "happy") (as well as an initial goal !init()).

The types keyword is used to specify the types of beliefs that are valid for a given program. They are used for type checking. If you use a belief that is not declared in this section of the program, the compiler will generate an error indicating that the belief has not been specified.

The query operator works by checking the agent's beliefs. If the provided belief can be matched to a belief in the agent's belief set, then the operator succeeds, otherwise it fails (and the corresponding intention fails).

ASTRA program structure: ASTRA Examples III

```
agent Uppy {
     module Console C;

     types uppy {
         formula light(string);
     }

     initial light("on");

     rule +light("on") {
         C.println("the light is on, turn it off!");
         -light("on");
         +light("off");
     }

     rule +light("off") {
         C.println("the light is off, turn it on!");
         -light("off");
         +light("on");
     }
}
```

ASTRA Belief Update example

This example illustrates how ASTRA permits the modification of the agent's internal state through the belief update plan operators.

One operator is provided to support the addition of new beliefs and a second operator is provided to support the removal of existing beliefs. The modification of an existing belief is achieved through the retraction of the existing belief and the subsequent adoption of the new belief.

This program includes an initial belief, representing the fact the a light is on, and two rules. The triggering event of the first rule is the event that the agent adopts a belief that the light is on. The body of the rule consists of a sequence of three actions: it prints out a message to the console, it retracts the belief that the light is on, and it adopts a belief that the light is off. The second rule does the opposite of this: it retracts the belief that the light is off and adopts the belief that the light is on.

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent Systems
    - Agent Communication
  - EIS Environment
- ASTRA and JADE
- PingPong Example

ASTRA Agent Programming Language: Multi-Agent Systems I

Agent Programming Languages are designed to support the development of multi-agent systems. In ASTRA this is achieved by writing an agent that creates more agents. The core functionality to support this approach is provided through the System API. [API ASTRA]

METHOD	DESCRIPTION
createAgent(String name, String class)	Action that allows the agent to create another agent.
name()	Term that returns the name of the agent.
sleep(int time)	Action that causes the agent to sleep for a specified period of time
terminate()	Action that allows the agent to terminate itself.
terminateAgent(String name)	Action to terminate another agent (if it exists and the invoking agent is the owner of that agent).

METHOD	DESCRIPTION
getAgents()	Term that returns a list of all the agents on the platform.
getAgentsOfType(String type)	Term that returns a list of all the agents on the platform of a specified type
getType()	Term that returns the type of the agent.
getType(String name)	Term that returns the type of the agent with the given name.
hasType(String type)	Formula that returns true if the given type is available on the agent platform

ASTRA Agent Programming Language: Multi-Agent Systems III

```
package example;

  agent Starter {
      module Console console;
      module System system;

      rule +!main(list args) {
          system.createAgent("worker", "example.Hello");
          system.sleep(1000);
          system.terminateAgent("worker");
      }
  }

package example;

  agent Hello {

      module Console console;
      module System system;

      initial !init();

      rule +!init() {
          console.println("Hello World, "+ system.name());
      }
  }
```

This program creates two agents. The “main” agent creates the “worker” agents, which is an instance of the “example.Hello” class, goes to sleep for a second, and then terminates that agent. An agent is able to terminate another agent only if it was the agent that created the other agent. The “worker” agent prints out “Hello World, worker”.

Up to now, we have been triggering the initial behaviour of the agent by specifying an initial goal !init(). This is not strictly necessary because ASTRA generates an initial goal automatically. The form of this goal is !main(list args) and it is the ASTRA equivalent of the public static void main(String[] args) { } method.

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent Systems
    - Agent Communication
  - EIS Environment
- ASTRA and JADE
- PingPong Example

Working with Multiple Agents: Agent Communication I

In ASTRA, the primary form of interaction between agents is through the use of an **Agent Communication Language (ACL)**. ASTRA supports direct interaction through **FIPA ACL-based message passing**. This ACL supports a range of message type which must be one of the standard *FIPA ACL Performatives* [FIPA ACL, 2002]:

accept-proposal	agree	cancel	cfp	confirm	disconfirm
failure	inform	inform-if	inform-ref	not-understood	propagate
propose	proxy	query-if	query-ref	refuse	reject-proposal
request	request-when	request-whenever	subscribe		

Working with Multiple Agents: Agent Communication II

Sending and Receiving a message

To send a message in ASTRA, you use the **send(…)** statement. This statement takes three arguments:

- a performative,
- the name of the receiver agent,
- the actual content to be sent.

Messages are received by the agent and converted into events. ASTRA includes a special event to support this, called **@message event**. This event has three parameters:

- the performative (type) of the message,
- the name of the agent that sent the message,
- the content that was sent.

In order for the agent to hear the message, it is necessary to write a rule that handles the event.

Working with Multiple Agents: Agent Communication III

```
package examples;

  agent Master {
      module Console C;
      module System S;

      initial !init();

      rule +!init() {
           S.createAgent("theslave", "examples.Slave");
      }

      rule @message(inform, string sender, state(string X)) {
          C.println(sender + " is alive!");
      }
  }

package examples;

  agent Slave {
      module Console C;
      module System S;

      initial !init();

      rule +!init() {
          send(inform, S.getOwner(), state("alive"));
      }
  }
```

In this example, two programs are defined: Master and Slave. The Master agent creates a Slave agent which sends a message back to the Master when it is created indicating that it is alive.

Running the Master results in an agent called “theslave” being created. This agent then sends a message to its owner (the Master) informing the Master that it is alive (it has been created). The Master is then able to communicate with the Slave and to ask it to perform tasks as necessary.

- ASTRA Agent Programming Language
  - Introduction
  - From AgentSpeak(L) to ASTRA
  - ASTRA Control Flow
  - ASTRA Program Structure
  - Multi-Agent Systems
  - EIS Environment
- ASTRA and JADE
- PingPong Example

ASTRA Agent Programming Language: EIS Environment I

The **Environment Interface Standard (EIS)** [EIS] is an attempt to provide a common interface between agents and the environment they inhabit.

For ASTRA, the approach that has been taken to linking agents to the interface has been based on a core language level integration. This support includes a statement for executing actions in the environment, a formula for querying environment beliefs, and an event for triggering behaviours. Additionally, a custom API has been developed to provide additional support for working with EIS environments. [API ASTRA]

Launching / Connecting to an Environment

- EIS environments are deployed as **jar** files. To use one, you should copy the jar file into the base directory of the Eclipse project (the project folder).
- When working with EIS environments, the first thing that needs to be done is to load the environment into the EIS Interface.
  - In ASTRA, this is done via the ***launch(..)*** action that needs to be performed once by a single agent. Other agents need to connect to the EIS environment by using the ***join(..)*** action
- It is possible for a single agent deployment to interact with more than one EIS environment at a time. Because of this, many of the actions and terms have been overloaded to work with both a specified environment and a **default environment**.
  - The default environment is the last environment that the agent joined. This can be overridden with the EIS API ***setDefaultEnvironment(..)*** action.
  - An agent can get its current default environment identifier using the the ***defaultEnvironment()*** term.

ASTRA Agent Programming Language: EIS Environment II

Linking Agents to Entities

- EIS Environments expose a set of entities that can be controlled by one or more agents. Each entity is associated with *a unique identifier (name)*, *a type* and *an associated set of actions and perceptions*.
- To link an agent to an entity, all you need is the name of the entity and, where more than one environment is involved, the id of the environment. Linking an agent to an entity is supported through ***link(..)*** actions.

Starting an Environment

- To start an environment one of the connected agents must execute one of the ***startEnv(..)*** actions.
  - Some environments start automatically when loaded, while others must be started by an agent.
- Sometimes it is not clear whether the environment needs to be started. In this case, the current state of the environment can be identified by using one of the ***getState(..)*** terms.
  - Normally, an environment will be created in a “paused” state. Once the environment has been started, it moves into a “running” state.

ASTRA Agent Programming Language: EIS Environment III

Working with Complex Environments

- Such environments add more complexity because developers typically want to associate different agent types with different entity types.
- Once an environment has been loaded, and an agent has been linked to it, you are able to retrieve a list of the entities in the environment.
  - To get a list of the entities that are not linked to an agent, you can use one of the ***freeEntities(..)*** terms that return a list of free entities registered to the environment.
  - Once you know the name of the entities, you can check their type by using ***getEntityType(..)*** terms that allow you to query an EIS environment to find out what entities exist and what their types are.

ASTRA Agent Programming Language: EIS Environment IV

```
package examples;

agent Tower {
    module EISAPI eis;

    initial !init();

    rule +!init() {
        eis.launch("tower", "towerenv.jar");
        eis.join("tower");
        eis.link("gripper");
        if (eis.getState() == "paused") eis.startEnv();
    }
}
```

Starting an Environment example

```
package examples;

agent Launcher {
    module EISAPI eis;
    module Prelude P;
    module System S;
    module Console C;

    initial !init();

    rule +!init() {
        eis.launch("soccerworld", "soccer.jar");
        eis.join("soccerworld");

        list l = eis.freeEntities();
        int i = 0;
        while (i < P.size(l)) {
            string entity = P.valueAsString(l, i);
            string type = eis.getEntityType(entity);
            if (type == "goalkeeper") {
                S.createAgent("p"+i, "examples.Goalkeeper");
            } else if (type == "defender") {
                S.createAgent("p"+i, "examples.Defender");
            } else if (type == "midfielder") {
                S.createAgent("p"+i, "examples.Midfielder");
            } else if (type == "attacker") {
                S.createAgent("p"+i, "examples.Attacker");
            } else {
                C.println("Unknown type: " + type + " for entity: " + entity);
            }
            i = i + 1;
        }
    }
}
```

Complex Environment example

ASTRA and JADE

- ASTRA Agent Programming Language
- ASTRA and JADE
  - JADE Overview
  - Comparison
- PingPong Example

ASTRA and JADE: JADE Overview I

Jade stands for **Java Agent DEvelopment Framework.**

Jade is a Java-based framework to develop agent-based applications in compliance with the **FIPA specifications** for *interoperable, intelligent, multi-agent systems*. [DISI, 16/17]

JADE goals [DISI, 16/17]

As an **agent-oriented middleware**, JADE pursues the twofold goal of being:

- a full-fledged FIPA-compliant *agent platform*
  - hence, it takes charge of all those application-independent aspects – such as agent lifecycle management, communication, distribution transparency, etc. – necessary to develop a MAS
- a simple yet comprehensive *agent development framework*
  - therefore, it provides Java developers a set of APIs to build their own customisations

ASTRA and JADE: JADE Overview II

JADE & FIPA [DISI, 16/17]

JADE is compliant to FIPA standards:

- a FIPA **agent platform** can be split onto several host. Each host acts as a **container** and provides a complete runtime environment for agent execution. At least one of the container acts as the **main container** that is responsible to maintain a registry of all other containers in the same platform
- for a given platform, a single **Agent Management System (AMS)** exists, which keep track of all other agents in the same platform. The AMS provides the white pages service.
- a single **Directory Facilitator (DF)** exists for each platform that keeps track of all advertised services provided by all the agents in the same platform. The DF provides the yellow pages service.
- for a given platform, a distributed message passing system exists, called **Agent Communication Channel (ACC)**, which controls the exchange of messages within the platform, implements the required facilities to provide for asynchronous communication, and manages all aspects regarding **FIPA ACL** (Agent Communication Language [FIPA ACL, 2002]) message format

ASTRA and JADE

- ASTRA Agent Programming Language
- ASTRA and JADE
  - JADE Overview
  - Comparison
    - Agents
    - Agent Behaviours
    - Communication
- PingPong Example

ASTRA and JADE: Comparison

Java

ASTRA and JADE are both examples of distributed, object-based, agent-oriented systems built on the Java language.

- **ASTRA** is an Agent Programming Language designed to be closely integrated with and influenced by Java
- **JADE** is a agent development framework fully implemented in Java

ASTRA and JADE

- ASTRA Agent Programming Language
- ASTRA and JADE
  - JADE Overview
  - Comparison
    - Agents
    - Agent Behaviours
    - Communication
- PingPong Example

Comparison: Agents

ASTRA

- In ASTRA, individual agents can be viewed as being equivalent to *processes* and the intentions of the agent are *threads*.
- User-defined agents must be declared with the *agent* keyword.
- All ASTRA agents on the same platform have a unique name, which is assigned during their creation.
- Agent business logic is modelled on the *BDI model*.
- ASTRA agents communicate by exchanging *FIPA ACL messages*.

JADE

- An agent is a *Java object* executed by a *Java thread*.
- User-defined agents must extend *jade.core.Agent* class, inheriting some ready-to-use methods.
- All JADE agents have a globally unique name (*agent ID, AID*), which is (by default) the concatenation – by symbol '@' – of their *local name* and of the JADE *platform name*.
- Agents business logic must be expressed in terms of *behaviours*.
- JADE agents communicate by exchanging *FIPA ACL messages*.

ASTRA and JADE

- ASTRA Agent Programming Language
- ASTRA and JADE
  - JADE Overview
  - Comparison
    - Agents
      - Agent states
      - Get an Agent ID
    - Agent Behaviours
    - Communication
    - Get an agent ID
- PingPong Example

Agents: Agent states I

ASTRA

- ASTRA agent states
  - NEW: the agent is created
  - ACTIVE: the agent is executing its behaviour
  - INACTIVE: the agent is ACTIVE and terminates its behaviour
  - RESCHEDULE: the agent is INACTIVE and is added a new agent's event to the event queue
  - TERMINATING: has been called System.terminate()
  - TERMINATED: the agent is terminated

Agents: Agent states II

JADE

- JADE agent states (FIPA states)
  - INITIATED: the agent object has been built, but cannot do anything since it is not registered to the AMS yet—it has no AID even
  - ACTIVE: the agent is registered to the AMS and can access all Jade features—in particular, it is executing its behaviour(s)
  - WAITING: the agent is blocked, waiting for something to happen (and to react to)—typically, an ACL message Suspended the agent is stopped, therefore none of its behaviours are being executed
  - TRANSIT: the agent has started a migration process—it will stay in this state until migration ends
  - UNKNOW: the agent is dead—it has been deregistered to the AMS

- FSM-like lifecycle with public methods to perform state transitions.

ASTRA and JADE

- ASTRA Agent Programming Language
- ASTRA and JADE
  - JADE Overview
  - Comparison
    - Agents
      - Agent states
      - Get an Agent ID
    - Agent Behaviours
    - Communication
- PingPong Example

Agents: Get an agent ID I

ASTRA

The ASTRA System API provides terms which allow an agent to get the name of another agent on the platform. This terms are:

- *getAgents()*: returns a list of all the agents on the platform
- *getAgentsOfType(String type)*: returns a list of all the agents on the platform of a specified type
- *getChildren()*: returns a list of all the agents on the platform that are children of the invoking agent
- *getOwner()*: returns the name of the agent's owner

Agents: Get an agent ID II

JADE

JADE provides several ways to get an agent ID:

- by using the agent local name
- from the RMA GUI
- by asking to the AMS
- by asking to the DF

For a given JADE platform:

- a single **Agent Management System** (**AMS**) exists, which keeps track of all other agents in the same JADE platform, and should be contacted by JADE agents prior to any other action. The AMS provides the white pages service.
- a single **Directory Facilitator** (**DF**) exists that keeps track of all advertised services provided by all agents in the same JADE platform, and should be contacted by JADE agents who wish to publish their capabilities. The DS provides the yellow pages service.

ASTRA and JADE

- ASTRA Agent Programming Language
- ASTRA and JADE
  - JADE Overview
  - Comparison
    - Agents
    - Agent Behaviours
    - Communication
- PingPong Example

Comparison: Agent Behaviours I

ASTRA

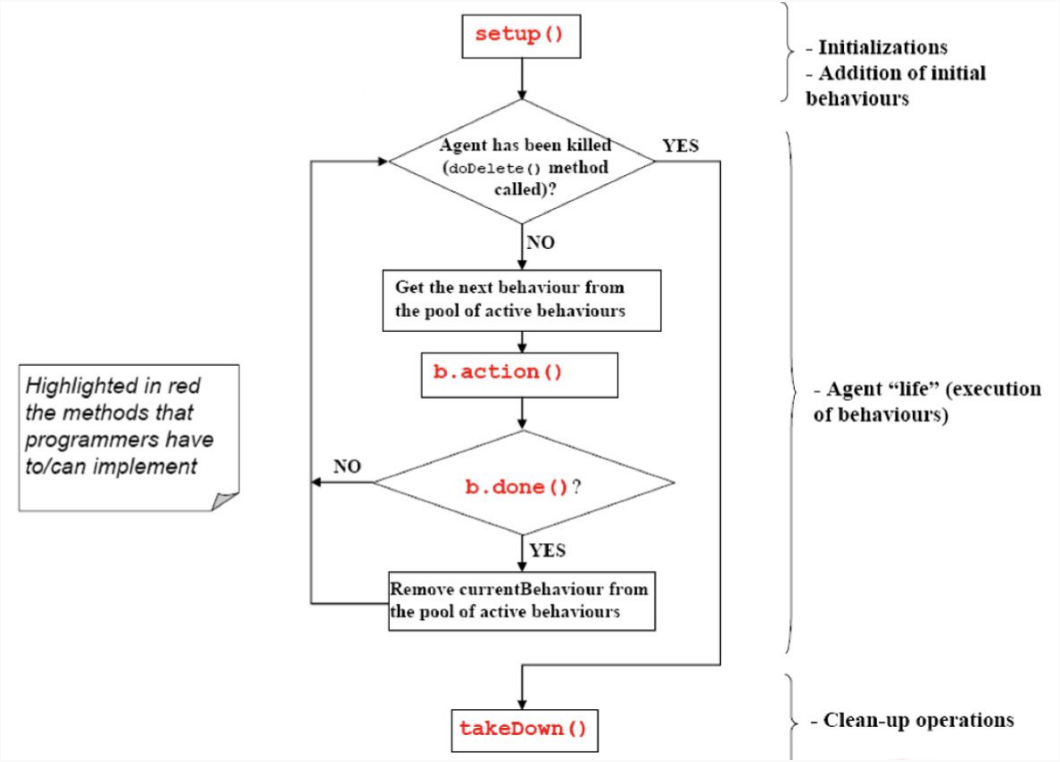
- ASTRA behaviours are encapsulated within a set of contextual event rules. Events trigger behaviours that define how the agent should respond to the event.
- Plan rules define the core behaviours of the agent, mapping those behaviours to the events that trigger them.
- The execution of behaviours is modeled through intentions.
- An agent creates a new intention for every external event that it matches to a plan rule. [Collier, 2015]
- An intention is an execution stack. It contains each action that must be performed in order to achieve the intention. The next action to be performed is sitting on top of the stack. An execution step involves removing the top item from the stack and executing it. Each iteration of the ASTRA interpreter, one intention is selected and the next statement in the behaviour is executed. [Collier, 2015]
- In situations where an agent has multiple intentions, the intention execution is interleaved. On each iteration, one intention is selected and executed. [Collier, 2015]. It is this interleaving of intentions that motivates the mapping of intentions to threads.
- ASTRA adopts a **round-robin policy**, ensuring that all intentions receive an equal opportunity for execution.

Comparison: Agent Behaviours II

JADE [DISI, 16/17]

- JADE implements behaviours as Java objects, which are executed concurrently, by a single thread, using a **non-preemptive, round-robin scheduler**.
- All behaviours are instances of *jade.core.behaviours.Behaviour*-derived classes:
  - Method *action()* should be overridden to carry out the application-specific task
  - Method *done()* should be overridden too to check such task termination condition
- All behaviours are in package *jade.core.behaviours*:
  - *SimpleBehaviour*
    - *OneShotBehaviour* : the behaviour is executed only once
    - *CyclicBehaviour* : the behaviour is executed forever, until agent death
  - *CompositeBehaviour*
    - *SequentialBehaviour* : can be added child behaviours that are scheduled sequentially. The whole behaviour ends when the last child ends
    - *ParallelBehaviour* : can be added child behaviours that are scheduled concurrently. The whole behaviour ends when the last child ends
  - *CompositeBehaviour*
    - *FSMBehaviour* : each child represents the activity to be performed within a state of the FSM. Some of the children can be registered as final states. The whole behaviour terminates after the completion of any of them
  - ...

Comparison: Agent Behaviours III



Jade non-preemptive scheduling policy

ASTRA and JADE

- ASTRA Agent Programming Language
- ASTRA and JADE
  - JADE Overview
  - Comparison
    - Agents
    - Agent Behaviours
    - Communication
- PingPong Example

Comparison: Communication I

JADE

The **Agent Communication Channel** (ACC) controls the exchange of messages within the Jade platform.

Following the FIPA specification, Jade agents communicate via **asynchronous message passing**.

- each agent has a *message queue* where the Jade ACC delivers ACL messages sent by other agents
- whenever a new entry is added to the queue, the receiving agent is *notified*.

To interact, Jade agents have a number of ready-to-use methods:

- **send**: to send a message to a recipient agent
- **receive**: to asynchronously retrieve the first message in the mailbox (if any)
- **timed receive**: to perform a timed, synchronous receive on the mailbox—timeout causes agent to resume execution
- **selective receive**: to retrieve a message from the mailbox which matches a given message template—message queue order is bypassed

Comparison: Communication II

ASTRA & JADE FIPA ACL-compliant messages

In both Astra and Jade the primary form of interaction between agents is through the use of an Agent Communication Language (ACL). An ACL message contains:

- *:sender* who sends the message—automatically set
- *:receiver* who the message targets—may be many
- *:performative* the name of the communication act the agents want to carry out—constrained by a FIPA ontology
- *:content* the actual information conveyed by the message
- *:language* the syntax used to encode the :content
- *:ontology* the semantics upon which the :content relies
- others fields.

FIPA performatives

Performatives identify the type of communicative act carried out by the message—thus its semantics and expected response

- CFP (Call For Proposal) to obtain proposals about something
- INFORM to let someone know something
- PROPOSE to propose something
- REQUEST to ask for a service
- SUBSCRIBE to subscribe for notification about something
- AGREE to express consensus about something
- REFUSE to refuse a request
- ...

They are constants to be set for any ACL message exchanged by agents

PingPong Example

- ASTRA Agent Programming Language
- ASTRA and JADE
- PingPong Example
  - PingPong in JADE
  - PingPong in ASTRA
  - Main differences

PingPong Example: PingPong in JADE I

```
package ds.lab.jade.messaging;

import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

public class PingPongAgent extends Agent {
    private class PingPongBehaviour extends CyclicBehaviour {
        private static final long serialVersionUID = 1L;
        private final String content;
        private final MessageTemplate msgTemplate;

        public PingPongBehaviour(final String c, final MessageTemplate t) {
            this.content = c;
            this.msgTemplate = t;
        }

        @Override
        public void action() {
            final ACLMessage msg = this.myAgent.receive(this.msgTemplate);
            if (msg != null) {
                PingPongAgent.this.log("Received message '" + msg.getContent() + "' from agent <" + msg.getSender() + ">.");
                final ACLMessage reply = msg.createReply();
                reply.setPerformative(ACLMessage.INFORM);
               if ("ping".equals(this.content)) {
                    reply.setContent("pong");
                ...
```

PingPong Example: PingPong in JADE II

```
            ...
            } else {
                reply.setContent("ping");
            }
            this.myAgent.send(reply);
        } else {
            this.block();
        }
    }
}

private static final long serialVersionUID = 1L;
private final MessageTemplate template = MessageTemplate.and(
        MessageTemplate.MatchPerformative(ACLMessage.PROPOSE),MessageTemplate.MatchOntology("ping-pong"));

private final MessageTemplate pingTemplate = MessageTemplate.and(this.template, MessageTemplate.MatchContent("ping"));
private final MessageTemplate pongTemplate = MessageTemplate.and(this.template, MessageTemplate.MatchContent("pong"));

private void log(final String msg) {
    System.out.println("[" + this.getAID() + "]: " + msg);
}

@Override
protected void setup() {
    this.log("Started.");
    this.addBehaviour(new PingPongBehaviour("ping", this.pingTemplate));
    this.addBehaviour(new PingPongBehaviour("pong", this.pongTemplate));
}
}
```

PingPong Example

- ASTRA Agent Programming Language
- ASTRA and JADE
- PingPong Example
  - PingPong in JADE
  - PingPong in ASTRA
  - Main differences

PingPong Example: PingPong in ASTRA

```
agent PingPongAgent {

    module Console C;

    types pingPong {
        formula text(string);
    }

    initial !init();

    rule +!init() {
        C.println("Started.");
    }

    rule @message(propose, string sender, text("ping")) {
        C.println("Received message 'ping' from agent <" + sender + ">.");
        send(inform, sender, text("pong"));
    }

    rule @message(propose, string sender, text("pong")) {
        C.println("Received message 'pong' from agent <" + sender + ">.");
        send(inform, sender, text("ping"));
    }
}
```

PingPong Example

- ASTRA Agent Programming Language
- ASTRA and JADE
- PingPong Example
  - PingPong in JADE
  - PingPong in ASTRA
  - Main differences

PingPong Example: Main differences I

Agent definition

- In **JADE** the PingPongAgent is defined by extending *jade.core.Agent* class
  - `public class PingPongAgent extends Agent { ... }`
- In **ASTRA** the PingPongAgent is defined by the *agent* keyword
  - `agent PingPongAgent { ... }`

PingPong Example: Main differences II

Behaviours definition

- In **JADE** the behaviour PingPongBehaviour is defined by extending the *CyclicBehaviour* class. Two different PingPongBehaviour are then added at the PingPongAgent, one to reply to “ping” and one to reply to “pong”.

  ```
  private class PingPongBehaviour extends CyclicBehaviour {
        …
        @Override
        public void action() { … }
  }

  @Override
  protected void setup() {
        …
        this.addBehaviour(new PingPongBehaviour("ping", this.pingTemplate));
        this.addBehaviour(new PingPongBehaviour("pong", this.pongTemplate));
  }
  ```

- In **ASTRA** a behaviour is defined by a plan rule. Two different plan rules are added, one to reply to “ping” and one to reply to “pong”.

  ```
  rule @message(propose, string sender, text("ping")) { … }
  rule @message(propose, string sender, text("pong")) { … }
  ```

PingPong Example: Main differences III

Receive a message I

- In **JADE** messages are received by the asynchronous, selective *receive(MessageTemplate)* that removes the first message from the mailbox.
  - `final ACLMessage msg = this.myAgent.receive(this.msgTemplate);`

  jade.lang.acl.MessageTemplate are used to get only messages matching a given pattern. Templates are used by the agent to properly distinguish different incoming messages. In the template are specified the performative, the ontology and the content which the message must match.

  - ```
    private final MessageTemplate template = MessageTemplate.and(
          MessageTemplate.MatchPerformative(ACLMessage.PROPOSE),
          MessageTemplate.MatchOntology("ping-pong"));

    private final MessageTemplate pingTemplate = MessageTemplate.and(this.template,
          MessageTemplate.MatchContent("ping"));
    private final MessageTemplate pongTemplate = MessageTemplate.and(this.template,
          MessageTemplate.MatchContent("pong"));
    ```
    ```

PingPong Example: Main differences IV

Receive a message II

- In **ASTRA** messages are received by the agent and converted into events. In order to hear the message, it is necessary to write a rule that handles the event. This rules are used by the agent to properly distinguish different incoming messages.

  ASTRA includes a special event called *@message*. This event takes in input the performative, the name of the agent that sent the message, and the content that was sent as a *formula*.

  - ```
    types pingPong {
           formula text(string);
    }
    ...
    rule @message(propose, string sender, text("ping")) { ... }
    rule @message(propose, string sender, text("pong")) { ... }
    ```
    ```

PingPong Example: Main differences V

Send a message I

- In order to send a message, in **JADE**, the agent should create an *ACL* message. The method *createReply()* automatically sets a number of *FIPA ACL* fields: *:receiver, :language, :ontology, :conversation-id, :protocol, :in-reply-to, :reply-with*.
  - *When sending a message the agent must know the receiver AID.*
  - The message is send by calling the asynchronous *send(...)* method.

```
final ACLMessage reply = msg.createReply();
reply.setPerformative(ACLMessage.INFORM);
if ("ping".equals(this.content)) {
      reply.setContent("pong");
} else {
      reply.setContent("ping");
}
this.myAgent.send(reply);
```

PingPong Example: Main differences VI

Send a message II

- In order to send a message, in **ASTRA,** is used the *send(…)* statement. This statement takes three arguments: a performative, the name of the receiver agent, and the actual content to be sent.
  - *When sending a message the agent must know the receiver name.*
  - Calling the *send(…)* statement ASTRA automatically creates an *AstraMessage* object which represents a FIPA ACL compliant message. This is set with a number of fields: *sender, receiver, performative, content, protocol, conversationId, language*.

  - ```
    rule @message(propose, string sender, text("ping")) {
        …
        send(inform, sender, text("pong"));
    }

    rule @message(propose, string sender, text("pong")) {
        …
        send(inform, sender, text("ping"));
    }
    ```
    ```

References I

- [ASTRA, 2017] *ASTRA Language web site*: http://astralanguage.com/wordpress/
- [API ASTRA] *API ASTRA web site:* http://www.astralanguage.com/api-javadoc/
- [Nilsson, 1994] Nilsson, N.J.,*Teleo-reactive programs for agent control.* JAIR 1 (1994)
- [Russel, 2015] Sean Russel, *Agent-Oriented Programming Languages as a High-Level Abstraction Facilitating the Development of Intelligent Behaviours for Component-Based Applications.* (2015)
- [FIPA ACL, 2002] *Agent Communication Language Specifications.* Foundation for Intelligent Physical Agents (FIPA)
- [EIS] *Environment Interface Standard (EIS) GitHub*: https://github.com/eishub/eis
- [CArtAgO] *CArtAgO web site*: http://cartago.sourceforge.net/
- [ACRE] *The ACRE Guide*: http://astralanguage.com/wordpress/acre-guide/

References II

- [DISI, 16/17] Andrea Omicini, Stefano Mariani, *Agent & Multi-Agent Systems with JADE,* DISI, Università di Bologna (2016/2017)
- [Collier, 2015] Collier, R., Russell, S., Lillis, D., *Exploring AOP from an OOP Perspective*, in Proceedings of the 5th International Workshop on Programming based on Actors, Agents and Decentralized Control (held at SPLASH 2014), Pittsburgh, Pennsylvania, USA (2015)