BRAHMS
An Introduction

António Castro
/DEI/FEUP
September 2010
JADE is meant for building agent systems, but has no simulation infrastructure.

Repast is meant for agent-based simulations and not for developing agent systems.

JADE is not a simulation framework: it has no scheduler, nor a notion of a "clock"; you would have to code any simulation infrastructure.

Repast is not for building MAS, and is not FIPA compliant.

Brahms has all these characteristics: simulation, building and execution environment for MAS...
- **Brahms** agents are Deliberative and BDI (but also supports reactive behaviour)
- **JADE** agents are whatever you design: Reactive, Deliberative (with Jess) or BDI (with Jadex)
- **Repast** agents are Reactive
Agents and Groups

Agents represents a person or an interactive system that has behavior interacting with the world that we want to represent as having the capabilities of awareness, reasoning and a mental state.

Agents are animate – intentional – objects

Agents act on their beliefs, that is, are the triggers of agent’s actions

Agents can conclude beliefs only for himself but can conclude facts about another agent’s attribute

A group represents a collection of agents that can perform similar work and have similar beliefs
GROUPS AND AGENTS - EXAMPLE

package isom sr;
import brahms.base.util.Log;

// OCC SUPERVISOR AGENT
agent OccSupervisor memberOf TapEmployees {
    location: TapOCC;
    attributes:
    relations:
    initial_beliefs:
    initial_facts:
    activities:

    composite_activity monitorTriggers() {
        end_condition: detectable;
        detectables:
        detectable detectTrigger {
            detect((Lisbon_AS.triggerConcept), dc:100);
        }
    }
    workframes:
    workframe wf_monitorTriggers {
        priority: 0;
        do {
            monitorTriggers();
        }
    }
    workframe wf_anomalyDetected {
        repeat: true;
        priority: 10;
        variables:
        ECmessage(string) code;
        when (knowsval(Lisbon_AS.triggerConcept = current.conceptCode)
            and (code = current.conceptCode))
            do {
                Log.info("ANOMALIE DETECTED! (%s)", code);
                conclude((Lisbon_AS.triggerConcept = ""), bc:100, fc:100);
            }
        }
    }
}
Objects and Classes

- **Object** represents a specific artifact in the world
- **Objects** are inanimate – unintentional – objects
- A **Conceptual Object** is used to allow for a user to track things that exist as concepts in people’s minds
- **Objects** act on facts only
- For **Objects** beliefs (can) represent information stored on the object
- **Classes** represents an abstraction of one or more object instances
- **Classes** defines the activities, workframes, initial-facts and initial-beliefs of the class of objects
Classes and Objects - Example

package iowch.sc;

// AIRPORT SCREEN CLASS
class AS {

// AIRPORT SCREEN OBJECT (World)
object Lisbon_AS instanceof AS {
// location: Lisbon;
attributes:
    boolean loadedFlights;
    int currMinute;
    int cfMinute;
    string triggerConcept;
relations:
initial_beliefs:
initial_facts:
    (current.loadedFlights = false);
    (current.currMinute = 0);
    (current.cfMinute = 0);
    (current.triggerConcept = "");
activities:
    primitive_activity waitOneRelativeMinute() {
        random: false;
        max_duration: 60;
    }
workframes:
    workframe wf_loadFlights {
        repeat: false;
        when (knownval (current.loadedFlights = false))
        do {
            java (AirportScreen) as = new AirportScreen();
            as.loadFlights();
            conclude (current.loadedFlights = true), bc: 0, fc: 100;
        }
    }
workframe wf_countMinutes {
    repeat: true;
    when (knownval (current.loadedFlights = true)
        and knownval (current.currMinute < 720)) // TODO: handle user config
    do {
        waitOneRelativeMinute();
        java (BrahmsTime) bt = new BrahmsTime();
        bt.tick();
        conclude (current.currMinute = current.currMinute + 1), bc: 0, fc: 100;
    }
}

// AIRCRAFT MOVEMENT SYSTEM CLASS
class AMS {
resource: false;
attributes:
relations:
initial_facts:
activities:
workframes:
}

// AIRCRAFT MOVEMENT SYSTEMS OBJECTS (HCC and OCC)
object HCC_AMS instanceof AMS {
// location: LisbonAirportHCC;
initial_facts:
initial_facts:
    }

object OCC_AMS instanceof AMS {
// location: TapOCC;
initial_facts:
initial_facts:
    }

Beliefs and Facts

Beliefs are propositions that represent the internal mental state of an agent or object.

Beliefs are always local to the agent or object.

Agents need to have a belief (even about an existing fact) in order to have knowledge about the fact and act upon it.

Fact is meant to represent some physical state of the world or an attribute of some object or agent.

Facts are global, with an appropriate detectable any agent can detect a fact.
Workframes

- Are activity rules, that is, execute activities
  - Can be seen as conditional statements or constraints
- Consists of preconditions of beliefs and lead to actions
  - If the conditions of a rule are believed then the associated activities are performed
- It is a declarative description of under what condition the agent/object will perform the activities specified in the body of the rule (in case of an agent, beliefs an agent has or in case of an object, the facts in the world)
Activities

- Abstraction of real-life actions that help to accomplish a task
- There are several types of activities: primitive, communicate, create object, composite, etc.

Thoughtframes

- Are agent’s inference rules, do not execute activities
- Consists of preconditions and consequences
- Can only create new beliefs for agents and objects
- Cannot create new facts in the world
Detectables

- Used for detecting facts and beliefs while the agent is executing a workframe and activity
- Allows for contextual awareness of the agent
- Detect only relevant facts/beliefs for the current activity
- Allows for modeling reactive behavior
Variables

- Can be used in a Workframe or Thoughtframe to write more generic frames
- Before a variable can be used it has to be declared
- The scope of the variable is bound to the frame it is declared in
- Supports quantifiers that affect the way a variable is bound to a specific instance of the defined type (group or object class) of the variable
  - for-each, for-one and collect-all
If three orders are assigned to agent Allen and agent Allen has beliefs for all three orders matching the pre-conditions:

Three *wfi’s* for agent Allen are created and, in each *wfi* the for-each variable is bound to one of the three orders. Allen will work on all the three orders, *one order at a time*.

One *wfi* for agent Allen is created and only one of the three orders gets bound. Allen randomly works on just one of the orders.

One *wfi* for agent Allen is created and binds the collect-all variable to a list of all three orders. Allen will work on all three orders, *at the same time*.
Geography

- It is not a Cartesian Geography
- It is an abstraction based on concepts such as areas and the way those concepts are connected (with paths)

- area_definition: used for defining types of area instances used for representing geographical locations.
- area: is an instance of an area_definition
- path: connects two areas and represents a route that can be taken by an agent or object to travel from one area to another
// generic area definitions
aredef Airport extends BaseAreaDef { } // airport
aredef AP extends BaseAreaDef { } // aircraft parking
aredef CI extends Building { } // check in
aredef HCC extends Building { } // hub control center
aredef ACT extends Building { } // crew terminal
aredef OCC extends Building { } // operational control center

// specific areas
area LisbonGeography instanceof World { }
area Lisbon instanceof City partof LisbonGeography { }
area LisbonAirport instanceof Airport partof Lisbon { }
area LisbonAirportACT instanceof AP partof LisbonAirport { }
area LisbonAirportCI instanceof CI partof LisbonAirport { }
area LisbonAirportHCC instanceof HCC partof LisbonAirport { }
area LisbonAirportACT instanceof ACT partof LisbonAirport { }
area TapOCC instanceof OCC partof Lisbon { }

// relevant paths
path LisAP to_from_LisHCC { 
  area1: LisbonAirportAP;
  area2: LisbonAirportHCC;
  distance: 100; //TODO: confirm
}

path LisCI to_from_LisHCC { 
  area1: LisbonAirportCI;
  area2: LisbonAirportHCC;
  distance: 100; //TODO: confirm
}

path LisACT to_from_LisHCC { 
  area1: LisbonAirportACT;
  area2: LisbonAirportHCC;
  distance: 100; //TODO: confirm
}
Groups are composed of

- Agents having
  - Beliefs and doing
    - Activities executed by
      - Workframes defined by
        - Preconditions, matching agent’s beliefs
        - Primitive Activities
        - Composite Activities, decomposing the activity
        - Detectables, including Interruption (of an activity)
        - Consequences, creating new Beliefs and/or Facts
Brahms Agent Architecture

**Brahms Agent**

- **Beliefs** (atomic formulas)
- **Desires** (workframes thoughtframes belief matching)
- **Plans** (workframes Thoughtframes activities)
- **Intentions** (workframes thoughtframes instantiations)

*Input* ➔ *Engine* ➔ *Output*

*Input* ➔ *Engine* ➔ *Action*
Activities that transfer beliefs to/from one agent to one or several other agents, or to/from an (information carrier) object.

An agent/object has to have the belief before it can communicate (i.e. tell) the belief to another agent/object.

Recipient agent/object will overwrite original beliefs with communicated beliefs.
Communications - Example

```plaintext
communicate communicatePIN(Atm at3) {
    max_duration: 1;
    with: at3;
    about:
        send(current.pinCommunicated = true),
        // send(current.believedPin = current.believedPin);
        send(current.believedPin = unknown);
    when: end;
}

communicate sendRequestAmount(Atm at3) {
    max_duration: 1;
    with: at3;
    about:
        send(current.amountCommunicated = true),
        //send(current.preferredCashOut = current.preferredCashOut);
        send(current.preferredCashOut = unknown);
    when: end;
}
Models a communication event between two actors

Defines a message that is based on the Communicative Act standard defined by FIPA

Specifies an Envelope with the address information (from, to, date, ...) and transport hints

Specifies a Payload for the message content and content properties
COMMUNICATIVE ACT – MY AGENT EXAMPLE

agent MyAgent memberof Communicator, SystemGroup {

workframes:

workframe wf_communicate {

variables:

forone(CommunicativeAct) oRequest;

do {

createCommunicativeAct(
    current, MyOtherAgent,
    REQUEST,
    "createActivity",
    "meeting",
    current,
    oRequest);

setPayloadProperty(oRequest, "type", Activity);

sendCommunicativeAct(oRequest, MyOtherAgent);

delete oRequest;

} // end do

} // wf_communicate

} // MyAgent
agent MyOtherAgent memberof Communicator, SystemGroup {
    activities:
    create_object createObject(string actname, Class acttype, Object actobj) {
        max_duration: 5;
        action: new;
        source: acttype;
        destination: actobj;
        destination_name: actname;
    }

    workframes:
    workframe wf_act {
        variables:
        foreach (CommunicativeAct) oRequest;
        foreach (Class) oClass;
        forone (symbol) actname;
        forone (Activity) actobj;

        when(
            knownval(oRequest.payload("performative") = REQUEST) and
            knownval(oRequest.payload("action") = createActivity) and
            knownval(oRequest.payload("type") = oClass) and
            knownval(oRequest.payload("subject") = actname)
        ) do {
            createObject(actname, oClass, actobj);
            conclude((actobj.name = actname));
            println_c("Creating activity "$1", actobj);
        } // end do
    } // wf_act
}
- **Agent** Model:
  - Represents the groups, agents and their relationships

- **Activity** Model
  - Represents the activities that can be performed by agents and objects

- **Communication** Model
  - Represents the communication between agents and objects

- **Timing** Model
  - Represents the constraints and relationships between activities (if any exist).
- **Knowledge Model**
  - Represents the knowledge (initial beliefs and thoughtframes) of agents and objects

- **Object Model**
  - Represents the (conceptual) classes and objects in the world, used as resources by agents or used to track information flow (environment with resources)

- **Geography Model**
  - Represents the geographical environment in which agents and objects perform their activities
agent Kim_Agent memberof Student {
  location: SouthHall;

  \n  \n  initial_beliefs:
  \n  (current contains Kim_Cash);
  (current contains Kim_BankCard);
  (Kim_Account.balance = 30.00);
  (Kim_Account.code = 1312);
  (Kim_Account.pin = 1331);
  (Kim_Account.openedWithBank WF_Bank);
  (Kim_Cash.amount = 18.00);
  (current.howHungry = 10.00);
  (Campanile_Clock.time = 1);
  (current.perceivedtime = 1);
  (current hasCash Kim_Cash);
  (current hasBankCard Kim_BankCard);
  (Blakes_Diner.foodcost = 6.0);
  (Raleigh_Diner.foodcost = 3.0);
  (current.preferredCashOut = 12.0);
  (current.needCash = false);
  (current.chosenDiner = Raleigh_Diner);
  (current.chosenBank = WF_Bank);
  (current.chosenBank = WF_Bank);
  (current.hasAccount Kim_Account);
  (current.correctPin = 1331);
  (current.believedPin = 1331);
  (current.male = false);
  (current.contains Kim_Cash);
  (current.contains Kim_BankCard);

  initial_facts:
  \n
C:\Users\acastro>bc

[ ]

NASA Ames Research Center Brahms Compiler Version 3.0.4
Copyright (c) 1997-2009 NASA Ames Research Center. All Rights Reserved.

Usage: bc [options] <brahms file> <brahms file> ...

Options:

-1p <library-path> set library path
-source <source-path> set the source code path
-d <destination-path> set the path where to write the compiled code to.
-bar <archive file> the archive file in which to store the files with the compiled code.
-ddt <dtd path> set the path where DTD files can be found
-ontology <destination-path> set the path where the ontology files need to be stored
-ontologybase <url> set the base URL used to reference ontology files
-uml generate XMI-based UML for the model
-cp set the Java class path to find Java activities
-nowarn suppress all warnings
-compatible enable backwards compatibility
-strict require imports for any referenced concepts/Java types
-debug display debug info
-debugcfg <cfgfilename> use specified debug configuration file
-? displays usage text

Brahms file:
.b file to be compiled

C:\Users\acastro>
Virtual Machine

C:\Users\acastro>bvm

Brahms Virtual Machine - Version: 5.1.2
CI Interface - Version: 1.12.9
Copyright © 1999–2010 NASA Ames Research Center. All Rights Reserved.
usage: bvm [-options] [<model>]

where options include:
  -cf <config file> use configuration file
  -username <username> name for vm to register in directory
  -mode <sim | rt | drt> mode of operation, simulation, real-time or distributed
  -date <date and time> the start date and time for the sim "MM/dd/yyyy HH:mm:ss zzzz"
  -time_unit <n> number of seconds represented by each unit of time
  -cp <class path> appended search path for Java class files
  -lp <library path> search path for brahms xml files
  -es <class files> external services to be loaded
  -ui display simulation control panel
  -no_auto_start do not automatically start the model, use the control panel to start model
  -no_auto_stop do not automatically stop the model when the simulation finishes, use the control panel instead

Note 1: virtual machine must at a minimum either get a brahms xml file as a parameter or a configuration file for the vm to register itself with a directory service for remote control by another component.
Note 2: the time as specified in the date parameter must be in military time, the 24 hour clock. The zzzz represents the time zone. The time zone must be specified using its full name, i.e. America/New_York, America/Los_Angeles. For a full list see the Calendar.b file in Models/brahms/base.

C:\Users\acastro>
### Brahms Agent Hosting Environment JAPI

#### JAPI Packages

<table>
<thead>
<tr>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>gov.nasa.arc.brahms.vm.api</td>
</tr>
<tr>
<td>gov.nasa.arc.brahms.vm.api.common</td>
</tr>
<tr>
<td>gov.nasa.arc.brahms.vm.api.components</td>
</tr>
<tr>
<td>gov.nasa.arc.brahms.vm.api.convert</td>
</tr>
<tr>
<td>gov.nasa.arc.brahms.vm.api.events</td>
</tr>
<tr>
<td>gov.nasa.arc.brahms.vm.api.exceptions</td>
</tr>
<tr>
<td>gov.nasa.arc.brahms.vm.api.jac</td>
</tr>
<tr>
<td>gov.nasa.arc.brahms.vm.api.jagt</td>
</tr>
<tr>
<td>gov.nasa.arc.brahms.vm.api.jsvc</td>
</tr>
</tbody>
</table>

#### Library Packages

<table>
<thead>
<tr>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>brahms.base.directory</td>
</tr>
<tr>
<td>brahms.base.system</td>
</tr>
<tr>
<td>brahms.base.util</td>
</tr>
<tr>
<td>brahms.communication</td>
</tr>
</tbody>
</table>
APPLICATION EXAMPLES - AOCC

Used to: Simulating the AOCC and study the impact of organization changes on AOCC global performance
APPLICATION EXAMPLES – VICTORIA MISSION

Used to:
Design (Plan) a robotic mission to the moon
APPLICATION EXAMPLES – MOBILE AGENTS

Used to:
Work as a Distributed MAS to support human teams in their Exploration tasks
Mobile Agents
Architecture for supporting Human-Agent Systems

Mars Desert Research Station Near Hanksville, Utah
The Satellite Dish back to Earth's internet
The EVA Astronaut

HabCom and Hab Robot Operator

The EVA Robotic Assistant
1. Write a **scenario** of what to **model** and define the objectives

2. Using the scenario, make a **list of concepts**, like:
   i. Agents, artifacts (physical objects), areas (locations) and conceptual objects (invoice, for example)

3. Using the scenario, find **all attributes** that say something about **agents** or **objects**
   i. List them with the appropriate agent or object
4. Do the same for relations:
   
i. Find all relations between agent, objects, areas and conceptual objects

   ii. List them assigning it to the appropriate concept listed in 2).

5. From the scenario, make a list of all activities performed by agents and objects
   
i. Place them with the agent or object for which the activity was defined (in the scenario)
6. Make an **hierarchical classification** of the concepts and attributes, relations and activities
   i. Pay attention to attributes that should be more generic

7. For **each activity** and **each agent or object** define the **conditions** under which the **activity** will be **performed**
   i. Include duration, states that can be concluded after being performed and activity priorities
9. Make **Workframes generic using variables**.
   a. Check if an agent or object should detect something in the world

10. For **measurement purposes**, **associate resources with activities**
    i. Resource usage can be used for statistical analysis

11. Simulate the model, fine-tune and simulate again. That’s the debugging cycle.
- Brahms web site and forum
  - http://www.agentisolution.com
  - http://groups.google.com/group/brahms-forum
- Brahms Tutorial
- Brahms book chapter:
  - An Agent-Oriented Language for Work Practice Simulation and Multi-Agent Systems Development
- Maarten Sierhuis PhD Thesis
• Questions?

ajmc@fe.up.pt