DeCoAgent

Deliberative Coherence Driven Agent

User Guide

DeCoAgentLibrary v1.0

30 June 2014

This document presents a user manual for DeCoAgent Library package. Any person who want to create a DeCoAgent can use this tutorial. In the first part of the user manual environment configuration of the DeCoAgent Library will be presented. In the second part, a Cleaner World agent, a famous example in intelligent agent literature, design and implementation with the usage of DeCoAgent Library will be presented.

A.1 Development and Execution Environment Configuration

1. Prerequisites

- Download and install a recent Java environment from http://java.sun.com/javase/downloads (if not installed).
- Download and install a suitable Eclipse distribution (>=3.5) from http://www.eclipse.org/downloads (if not installed).
- Download Jadex 2.3 distribution .zip from http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/Download/Available+Packages and unpack it to a place of your choice.
- Download DeCoAgent package (DeCoAgentLib.jar) from https://sourceforge.net/projects/decoagentlibrary

2. Eclipse Environment Setup

Start Eclipse IDE and start the 'New Java Project' wizard, disable the 'Use default location' checkbox and browse to the directory, where you unpacked the Jadex distribution.

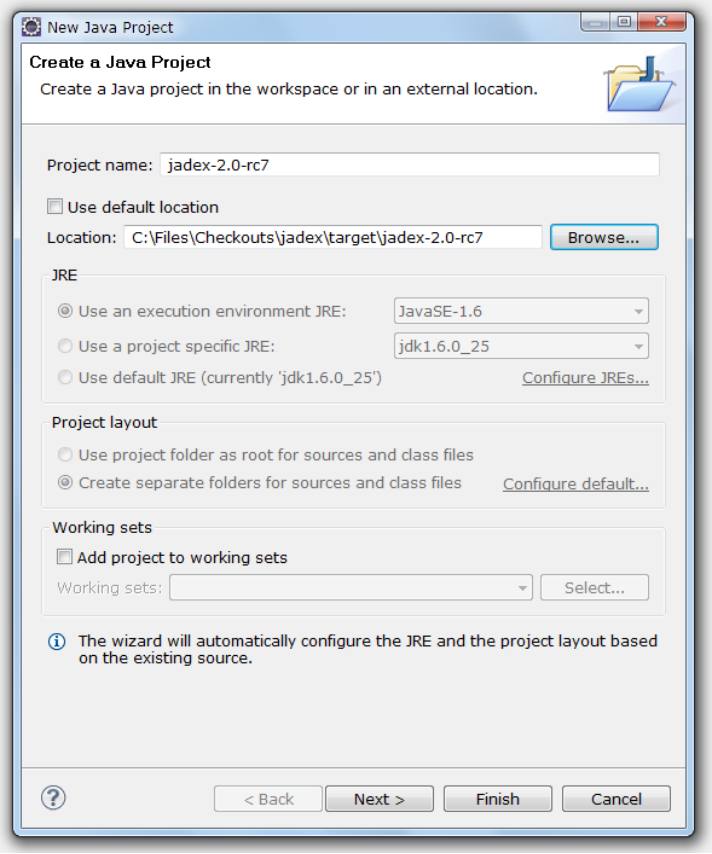

Figure 1 Creation of Java Project in Eclipse

Click 'Finish' – then the project will be created. Your project should now appear IN your package explorer. This project will be used as a basis for your own development projects. To make the Jadex libraries accessible to other projects it is necessary in Eclipse to export them. Right-click

on project, choose ‘Build Path -> Configure Build Path...’. Go to the ‘Order and Export’ tab, choose ‘Select All’ and hit ‘Ok’.

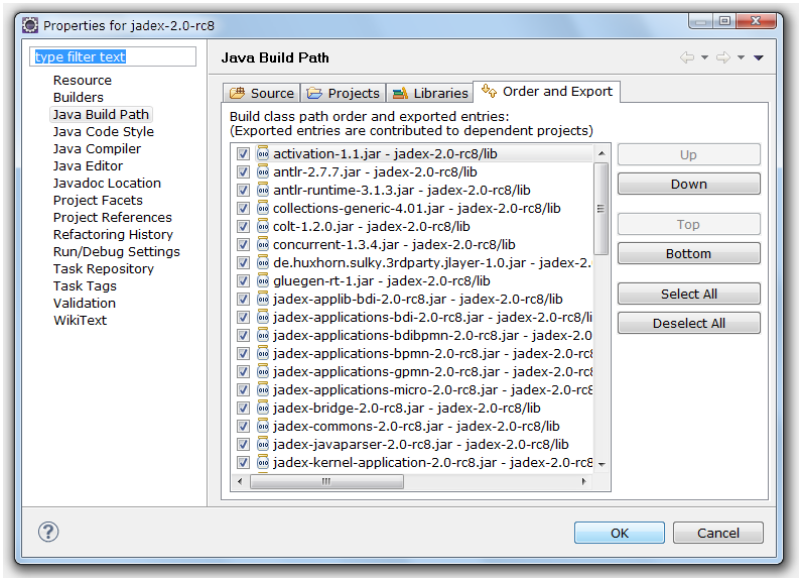

Figure 2 Export .jars from Build Path

To start the Jadex platform right-click on the Jadex project, which is in the package explorer. Then choose ‘Run As’ > ‘Java Application’ from the pop-up menu. Eclipse will search for main types (startable main classes). Select *Starter* from the package *jadex.base* in the dialog.

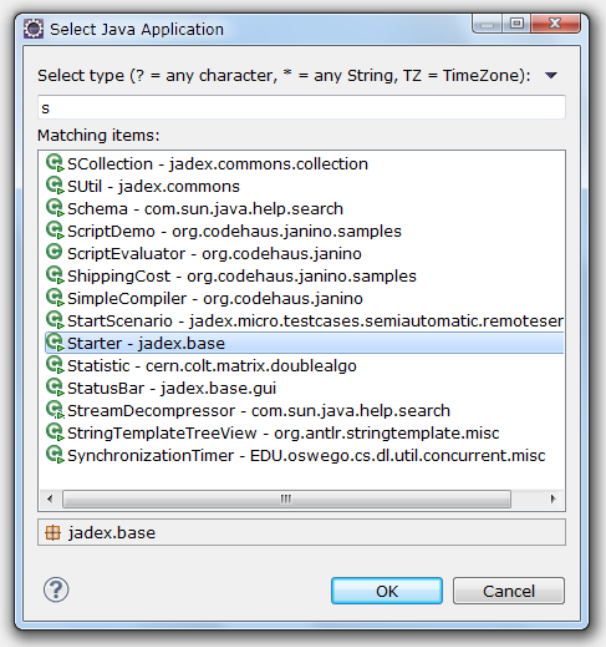

Figure 3 Select *Starter* Class for Starting Jadex Platform

Hit ‘Ok’ to start the Jadex platform. The next time you want to start the platform, you do not have to repeat the above steps. Just choose the ‘Starter’ entry from the run history, which eclipse

generates automatically.

If you successfully start the Jadex platform, the Jadex control center (JCC) will appear in the screen as seen in Fig. 4. The JCC is a management and debugging interface for the Jadex platform and components that run on it. To execute any application you need to add the corresponding path to the JCC project. We will now set up the platform for starting some examples. Right-click in the upper left area (called the model explorer, as it is used to browse for models of e.g. processes) and choose ‘Add Path’.

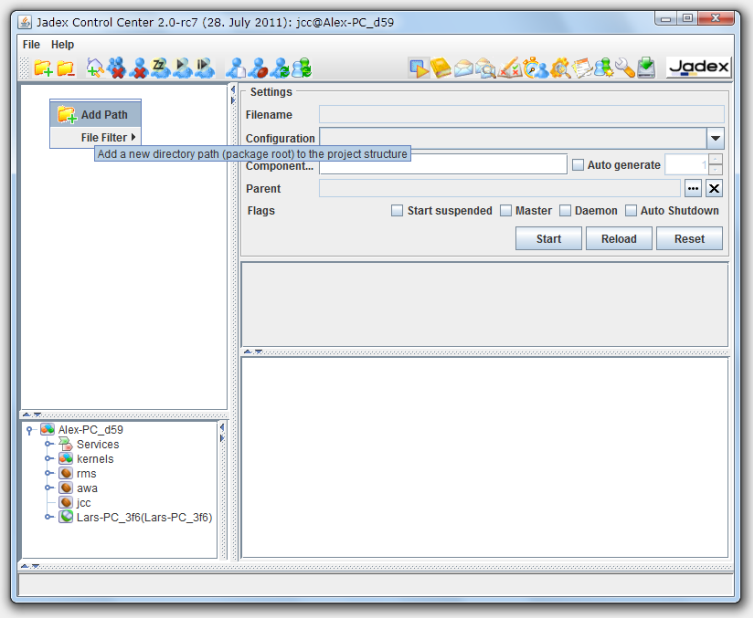

Figure 4 Add Path in JCC

For executing a sample agent open the *lib* directory and select the file *jadex-applications-micro-.....jar*. You can now unfold the contents of the jar file and browse to the helloworld example in the *jadex/examples* package. After you select the *HelloWorldAgent.class* in the tree, you can start the process by clicking 'Start'. The component will be executed and it will print some messages to the (eclipse) console.

3. Create and Setup a DeCo Agent Project

Create a new Java project from ‘File’ -> ’New’ -> ’New Java Project’ and name it as DeCoExamples. Choose your project location or use default one. Click ‘Finish’ – then your empty java project will be created and shown in package explorer.

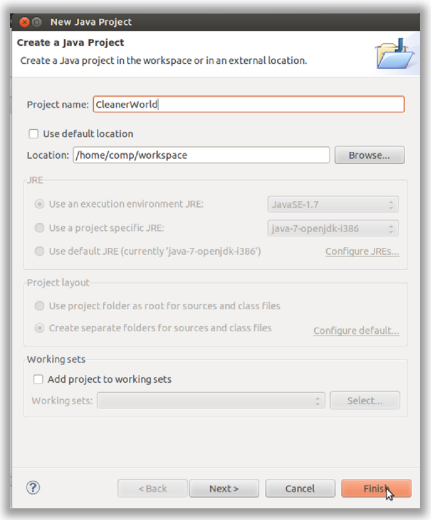

Figure 5 Create DeCoExamples Java Project

DeCo Agent library needs Jadex libraries for execution. Put *DeCoAgentLib.jar* file under the Jadex project source library file. For configuring our DeCoExamples project, right-click on project and choose 'Build Path -> Configure Build Path...'. Go to the 'Libraries' tab, choose 'Add JARs...' and choose Jadex project from appearing window and add all library files to the project. Then choose 'Add External JARs...', browse the *DeCoAgent.lib* file that needs to be added the Jadex lib source and choose 'OK'. After importing the necessary libraries go to the 'Order and Export' tab and change the order of the DeCoAgentLib.jar for it to be above Jadex libraries as shown in Fig. 6.

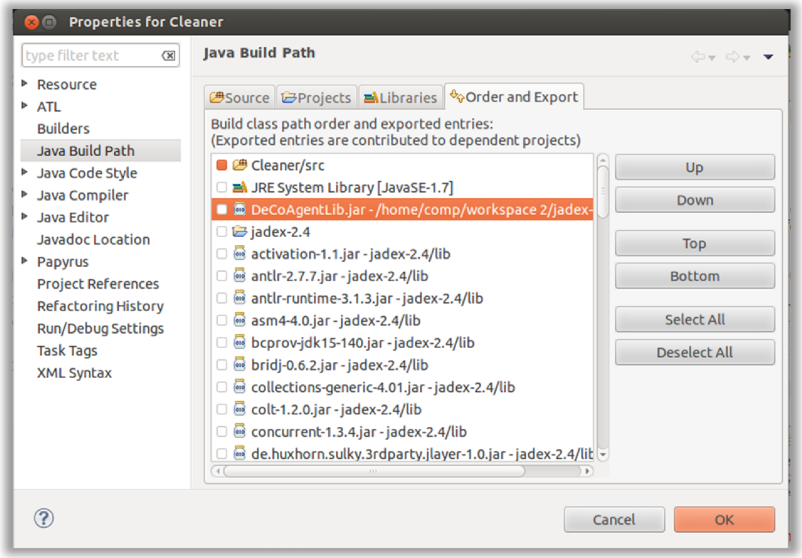

Figure 6 Change the Order of the DeCoAgentLib

The configuration of our environment is completed. Now we can start to implement our first DeCoAgent.

A.2 Implementing a DeCoAgent

1. First DeCo Agent

Choose DeCoExamples project from package explorer and create a new package called 'deco.jadex.examples'. DeCoAgent is a BDIv3 agent and most of its properties belong to Jadex BDIAgent. For BDIAgent detailed documentation, you can visit Jadex Active Components site (http://www.activecomponents.org/bin/view/BDI+V3+Tutorial/01+Introduction). The agent definition file is a normal Java class that uses *@Agent* annotation to state that is an agent. It is required that the Java file ends with "DeCo". Otherwise it will not be recognized as DeCoAgent.

Create a new empty Java class and name it as HelloDeCo.class. Add *@Agent* annotation before HelloDeCo class definition. This annotation states that this class is an agent. Add a variable called 'agent' to the agent class and add annotation *@Agent* above the agent variable. The field should be type of DeCoAgent. This will let the engine automatically inject the DeCo Agent (API) to the plain old java object (pojo) agent class. Add an agent body method that is automatically invoked when the agent is started. Add a println line inside the agent body for console output.

```
package deco.jadex.examples;


import deco.jadex.bdiv3.DeCoAgent;
import jadex.micro.annotation.Agent;
import jadex.micro.annotation.AgentBody;

@Agent
public class HelloDeCo
{
	@Agent
	DeCoAgent agent;

	@AgentBody
	public void body()
	{
		System.out.println("Hello World!");
	}
}
```

Figure 7 First DeCoAgent Example

This first agent is not different from a classic BDIAgent and does not contain any DeCoAgent ability. We implement this agent for testing our environment.

2. Executing DeCoAgent

Jadex Control Center only executes its inner assigned agent types (MicroAgent, BDIAgent…). DeCoAgent is an extension package for Jadex platform and cannot be executed within JCC. But, Jadex is a service based library and we can use its services with Jadex libraries. We will execute our agents with a Java application that contains main function.

Create a new java class file named *ExecuteMain.class*. Add a main method to the class. As explained above, place the following code inside the method body.

This code first defines a string array with default settings as shown below:

- **-gui false** disables the JCC

- **-welcome false** disables printing of the welcome message with the platform setup
- **-cli false** disables the command line interface
- **-printpass** false disables printing of platform password

After that a new array is created. First one is filled with the default arguments and then second one is filled with the args which are supplied to the main method. Therefore, in case of conflicts the arguments supplied from the outside override the default settings. Make sure that in the *createPlatform(...)* method the *newargs* array is passed instead of the *args*. Otherwise the default settings will be ignored.

```
package deco.jadex.examples;

import jadex.base.Starter;
import jadex.bridge.IExternalAccess;
import jadex.bridge.service.RequiredServiceInfo;
import jadex.bridge.service.search.SServiceProvider;
import jadex.bridge.service.types.cms.IComponentManagementService;
import jadex.commons.future.IFuture;
import jadex.commons.future.ThreadSuspendable;

public class ExecuteMain{

       public static void main(String[] args)
       {
              String[] defargs = new String[]
              {
                     "-gui", "false",
                     "-welcome", "false",
                     "-cli", "false",
                     "-printpass", "false"
              };
              String[] newargs = new String[defargs.length+args.length];
              System.arraycopy(defargs, 0, newargs, 0, defargs.length);
              System.arraycopy(args, 0, newargs, defargs.length,
args.length);
              IFuture<IExternalAccess>  platfut =
                Starter.createPlatform(newargs);
              ThreadSuspendable sus = new ThreadSuspendable();
              IExternalAccess platform = platfut.get(sus);
              System.out.println("Started platform:
                "+platform.getComponentIdentifier());

              IComponentManagementService cms =
             SServiceProvider.getService(platform.getServiceProvider(),
             IComponentManagementService.class,
             RequiredServiceInfo.SCOPE_PLATFORM).get(sus);

           cms.createComponent("deco.jadex.examples.HelloDeCo.class",null)
                .get(sus);
       }
```

Figure 8 Main Class for DeCoAgent Execution

In order to search component management service and to create an access point for the service mechanism we use the static helper class *SServiceProvider* from package *jadex.bridge.service.search*.

The result of the *getService(...)* method is a future of the corresponding service type. The application is still running on the Java main thread, therefore it is safe to use the thread suspendable again for blocking until the search result is available.

We pass our DeCoAgent class name with package to the *createComponent(...)* function of the component management service.

Finally we run ExecuteMain.class as a java application and our HelloDeCo agent is started to run on platform Jadex. The execution's output of the agent will be shown in Eclipse console panel as "Hello World!".

3. DeCoAgent Design and Implementation

DeCoAgent implementation needs a deliberative coherence model. Before implementing our DeCoAgent, we will describe the scenario of the cleaner world. After the definition of goals and actions, we will design our deliberative coherence model and finally we will implement our DeCoAgent. In this document we will explain DeCoAgent capabilities within four exercises in expanding manner. In each step we will add additional abilities to our agent.

a. Cleaner World General Scenario Description

A cleaner is an autonomous robot that can randomly move around, can sense the dirtiness, can clean the dirtiness, and can recharge itself. The cleaner agent description according to PEAS criteria presented in Table 17. The environment of the agent will be a room that have no object.

Table 1 PEAS Description of Cleaner World

Performance	Clean	Cleaner tries to keep its environment clean.
	Keep Battery Loaded	When battery level below than a defined value, it should return charging station to keep battery loaded.
Environment	Partially observable	Cleaner can only sense current position of the environment, where it possess.
	Static	Environment is not changing in Cleaner reasoning process.
	Stochastic	The next state of the environment cannot be determined with current state.
	Continues	Steering is a continues-state and continues-time problem.
Actuators	Steering	For moving around.
	Vacuum	For cleaning the dirtiness.
Sensors	Dirtiness Sensor	For sensing the dirtiness.

b. Case Study-1

(1) Scenario Description

According to the first case study, cleaner will have no performance criteria, a steering actuator and no sensor. It will only move around randomly.

(2) DeCoModel Design

Firstly we will create an agent that is only moving around. Our goal will be MoveAround and our action will be Move. MoveAround will only create a random position and post it to Move action. Our deliberative coherence model will be as in Fig. 9. When MoveAround goal priority is 1 both of the units will be activated (left side model) and when MoveAround goal priority is 0 both of the units will be deactivated (right side model).

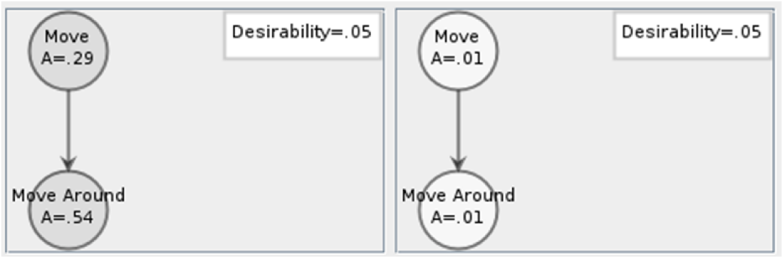

Figure 9 Exercise 1 Cleaner DeCo Model

According to our created model when MoveAround priority is 1 MoveAround goal activation will be 0.54 and Move action activation will be 0.29. When MoveAround priority is 0 MoveAround goal activation will be 0.54 and Move action activation will be 0.29.

(3) Implementation

Create a new package and name it as "deco.jadex.examples.cleaner". Right click on the package, create new java class file and name it as "CleanerDeCo.java". Add `@Agent` annotation before the class definition. Add DeCoAgent API as an attribute for the agent. Our agent currently will only move around. We need two beliefs for keeping agent coordinates (xPos - yPos). We add our beliefs to agent by `@Belief` annotation. We generate getter and setter functions of these beliefs from menu 'Source -> Generate Getters and Setters'.

Imported DeCoAgent libraries are sensitive and may be interfered with Jadex BDIv3 libraries. Keep in mind that when you are implementing a DeCoAgent, you should not import any BDIv3 library.

```
package deco.jadex.examples.cleaner;

import deco.jadex.bdiv3.DeCoAgent;
import deco.jadex.bdiv3.annotation.Belief;
import jadex.micro.annotation.Agent;

@Agent
public class CleanerDeCo
{
	@Agent
	DeCoAgent agent;

	@Belief
	private double xPos;

	@Belief
	private double yPos;

	public double getxPos()
	{
		return xPos;
	}

	public void setxPos(double xPos)
	{
		this.xPos = xPos;
	}

	public double getyPos()
	{
		return yPos;
	}

	public void setyPos(double yPos)
	{
		this.yPos = yPos;
	}
}
```

Figure 10 CleanerDeCo Agent Definition

For now our agent have no goals. We will add our first goal Move to our agent. As we mentioned earlier, DeCoAgent has all of the Jadex BDIv3 abilities. You can define goals with Inner Classes or Classes. We will define our goals with inner classes. Create a new Inner Class and name it as "Move".

Goals create basis of the deliberative coherence model creation. In Jadex, in order to define a goal you must use *@Goal* annotation. In DeCoAgent library *@Goal* annotation have 'relations' and 'priority' fields in addition to the JadexBDIv3 abilities. 'relations' field take an *@Relation* annotation for defining incompatible and facilitation relationships. *@Relation* annotation had 'facilitations' and 'incompatible' fields. 'priority' field defines the inherited priority of the current goal and takes 'double' parameter between 0.0-1.0.

To represent facilitation in DeCo model, an excitatory link is created between goals proportional to the indicated degree. 'facilitations' field take one or more *@Facilitate* annotation object for defining facilitation relationships between goals. It have two fields 'units' and 'degree'. We define goals in 'units' field that facilitate current goal. 'units' field take goal classes as a

parameter and can take one or more parameter. ‘degree’ represents the degree of the facilitation. ‘degree’ field takes ‘double’ parameter between 0.0-1.0.

To represent incompatibility in DeCo model, an inhibitory link is created between goals for the indicated degree. ‘incompatible’ field takes one or more *@Incompatible* annotation object for defining incompatible relationships between goals. It has two fields ‘unit’ and ‘degree’. We define goal in ‘unit’ field that is incompatible with current goal. ‘unit’ field takes goal classes as a parameter and can take only one goal class as a parameter. ‘degree’ represents the degree of the incompatibility. ‘degree’ field takes ‘double’ parameter between 0.0-1.0.

You can void *@Goal* annotation fields. Then no relationship will be defined for current goal and its priority will be assigned as internally.

JadexBDIv3 supports *@GoalCreationCondition*, *@GoalTargetCondition*, *@GoalDropCondition*, *@GoalContextCondition* method conditions for a goal. DeCoAgent Library supports all of these abilities except *@GoalContextCondition*. Because DeCoAgent takes the context handling.

Add the ‘Move’ class in to the agent definition as an inner class.

```
@Goal
    public class Move
    {
            private double xTargetPos;
            private double yTargetPos;

            public Move(double x, double y)
            {
                    xTargetPos=x;
                    yTargetPos=y;
            }
            @GoalTargetCondition(events={"xPos","yPos"})
            public boolean checkTarget()
            {
                    boolean ret=false;
                    double distance=Math.sqrt((xPos-xTargetPos)*(xPos-
                   xTargetPos)-(yPos-yTargetPos)*(yPos-yTargetPos));
                    if (distance<=0.01)
                    {
                            ret=true;
                    }
                    return ret;
            }
            public double getXTarPos()
            {
                    return xTargetPos;
            }
            public double getYTarPos()
            {
                    return yTargetPos;
            }
    }
```

Figure 11 Move Goal Definition

Goal denotes the fact that an agent commits itself to a certain objective and maybe tries all the possibilities to achieve its goal. Plans support goals by achieving agent to some state of affair. A plan defines two aspects. In the head of the plan (i.e. in its *@Plan* annotation) meta information

about the plan is defined. This means that in the plan head, several properties of the plan can be specified, e.g. the circumstances under which it is activated and its importance in relation to other plans. The body of a plan contains the concrete instruction that should be carried out. Plans can be defined as methods, inner classes or classes.

Now create a new Java class and name it as "MovePlan.java". A plan must start with *@Plan* annotation in Jadex. *@PlanCapability* defines the API which is used this plan. *@PlanAPI* defines the plan api of the library and the *@PlanReason* defines the aim of the current plan. Every plan must have a body. Each plan body must be annotated with *@PlanBody*. Plan body defines the path of an agent that must follow for achieving its goal.

Add the following MovePlan definition to the MovePlan.java file.

```
package deco.jadex.examples.cleaner;

import deco.jadex.bdiv3.annotation.Plan;
import deco.jadex.bdiv3.annotation.PlanAPI;
import deco.jadex.bdiv3.annotation.PlanBody;
import deco.jadex.bdiv3.annotation.PlanCapability;
import deco.jadex.bdiv3.annotation.PlanReason;
import deco.jadex.bdiv3.runtime.IPlan;
import deco.jadex.examples.cleaner.CleanerDeCo.Move;
import jadex.commons.future.DelegationResultListener;
import jadex.commons.future.Future;
import jadex.commons.future.IFuture;


@Plan
public class MovePlan
{
	@PlanCapability
	protected CleanerDeCo capa;

	@PlanAPI
	protected IPlan rplan;

	@PlanReason
	protected Move goal;

	public MovePlan()
	{
		//
	}

	@PlanBody
	public IFuture<Void> body()
	{
		return moveToDestination();
	}
	...
}
```

Figure 12 Move Plan Definition

Our move plan takes the target x and y coordinates from Move goal and tries to lead the agent there. To satisfy this, agent must change its current location in every step of execution through to the target position.

Add the following moveToDestinationFunction() after the body of the MovePlan.

```
protected IFuture<Void> moveToDestination()
{
	final Future<Void> ret = new Future<Void>();

	double destX = goal.getXTarPos();
	double destY = goal.getYTarPos();
	double posX = capa.getxPos();
	double posY = capa.getyPos();

	if(!isNear(posX,posY,destX,destY))
	{
		oneStepToTarget().addResultListener(new
    DelegationResultListener<Void>(ret)
		{
			public void customResultAvailable(Void result)
			{
				moveToDestination().addResultListener(new
          DelegationResultListener<Void>(ret));
			}
		});
	}
	else
	{
		ret.setResult(null);
	}

	return ret;
}
```

Figure 13 MovePlan moveToDestination() Function Definition

At every execution step, agent will get closer to the target position until it reaches to the target. Add the following oneStepToTarget() function after the moveToDestination() function.

```
        protected IFuture<Void> oneStepToTarget()
        {
                final Future<Void> ret = new Future<Void>();

            double destX = goal.getXTarPos();
               double destY = goal.getYTarPos();
               double posX = capa.getxPos();
               double posY = capa.getyPos();

            double d = getDistance(posX,posY,destX,destY);
               double r = 0.00004*100;
               double dx = destX-posX;
               double dy = destY-posY;
               double rx = r<d? r*dx/d: dx;
               double ry = r<d? r*dy/d: dy;

            capa.setxPos(posX+rx);
               capa.setyPos(posY+ry);
               rplan.waitFor(100).addResultListener(new
                  DelegationResultListener<Void>(ret)
                      {

                  });
               System.out.println("Cleaner moved to position
X:"+capa.getxPos()+
                  "Y:"+capa.getyPos());

               return ret;
```

Figure 14 MovePlan oneStepToTarget() Function Definition

Add the following necessary functions to the MovePlan.

```
public double getDistance(double x, double y, double tarX, double tarY)
{
        return Math.sqrt((y - tarY) * (y - tarY) + (x - tarX) * (x - tarX));

}

public boolean isNear(double x, double y, double tarX, double tarY)
{
        return getDistance(x, y, tarX, tarY) <= 0.01;
`
```

Figure 15 Other Necessary Functions

Now we completed the MovePlan implementation. Close the MovePlan.java and open the CleanerDeCo.java file.

Our second goal is the MoveAround goal. MoveAround goal must be facilitated by Move goal according to our deliberative coherence model. As we mentioned earlier, we define relations with *@Relation* annotation at 'relations' field of the goal and we define priority of the goal at 'priority' field. We add facilitation relationship into the 'facilitations' field of the *@Relations*. Fig. 16 shows *@Facilitate* annotation usage. This type of usage means that Move.java goal facilitates MoveAround.java goal and this is what we want to create with deliberative coherence model.

Add the following MoveAround goal definition to our agent definition file CleanerDeCo.java.

```
@Goal(excludemode=ExcludeMode.Never, succeedonpassed=false,
        relations=@Relations(facilitations=
        {
                @Facilitate(units={Move.class},degree=1)
        }), priority=1.0)
public class MoveAround
{
        public MoveAround()
        {
                //
        }
}
```

Figure 16 MoveAround Goal Definition

@Goal annotation fields and constants except relations and priority fields are about BDIAgent and its documents can be found at http://www.activecomponents.org .

Now we must implement a plan that can be executed, when MoveAround goal is adopted. Create a new java document and name it as MoveAroundPlan.java. We want our agent to pick a random position and move there iteratively. So our plan will create a random position and create a new Move goal to this target position.

In a plan body we can dispatch sub goals for execution by using plan API. In Fig. 17 we define 'rplan' as our plan API and we dispatch Move sub goal using it. This dispatching means that Move goal which is a subgoal of the MoveAround goal will be executed.

Add the following MoveAroundPlan definition to our MoveAroundPlan.java file.

```
package deco.jadex.examples.cleaner;

import deco.jadex.bdiv3.annotation.Plan;
import deco.jadex.bdiv3.annotation.PlanAPI;
import deco.jadex.bdiv3.annotation.PlanBody;
import deco.jadex.bdiv3.annotation.PlanCapability;
import deco.jadex.bdiv3.runtime.IPlan;
import deco.jadex.examples.cleaner.CleanerDeCo.Move;
import jadex.commons.future.ExceptionDelegationResultListener;
import jadex.commons.future.Future;
import jadex.commons.future.IFuture;

@Plan
public class MoveAroundPlan
{
	@PlanCapability
	protected CleanerDeCo capa;

	@PlanAPI
	protected IPlan rplan;

	public MoveAroundPlan()
	{
		//
	}

	@PlanBody
	public IFuture<Void> body()
	{
		System.out.println("Cleaner started to execute Move Around
		  plan.");

		final Future<Void> ret = new Future<Void>();

		double xDest = Math.random();
		double yDest = Math.random();

		IFuture<Move> fut = rplan.dispatchSubgoal(capa.new
		  Move(xDest,yDest));
		fut.addResultListener(new
		  ExceptionDelegationResultListener<CleanerDeCo.Move,
	    Void>(ret)
		{
			public void customResultAvailable(Move amt)
			{
				ret.setResult(null);
			}
		});

		return ret;
	}
}
```

Figure 17 MoveAroundPlan Definition

We implemented all the goals and plans but before running our agent we must bind goals and plans in agent definition file. In CleanerDeCo.java file after *@Agent* annotation, add plans in a *@Plans* annotation. *@Plans* annotation takes one or more *@Plan* annotation as an element. A *@Plan* annotation have two fields. First field is the goal definition class, 'trigger' object of the plan. Second field is the plan definition class, 'body' of the plan.

```
@Agent
@Plans(
{
       @Plan(trigger=@Trigger(goals=CleanerDeCo.Move.class),
      body=@Body(MovePlan.class)),
       @Plan(trigger=@Trigger(goals=CleanerDeCo.MoveAround.class),
      body=@Body(MoveAroundPlan.class))
})
public class CleanerDeCo
{
       ...
}
```

Figure 18 Goal and Plan Binding

When a BDIAgent starts the execution, it needs a top level goal. In our agent definition file, add a body function annotated with *@AgentBody* and dispatch MoveAround goal as a top level goal.

Our deliberative coherence model sets activations of the units according to desirability of the model and we set our DeCoModel desirability as 0.05.

```
       @AgentBody
       public void body()
       {
              agent.setDesirability(0.05);
              agent.dispatchTopLevelGoal(new MoveAround());
       }
```

Figure 19 AgentBody

For execution of our agent, reopen the ExecuteMain.class file and change the create component line as shown in Fig. 20.

```
   cms.createComponent("deco.jadex.examples.cleaner.CleanerDeCo.class",
     null).get(sus);
```

Figure 20 Running the CleanerDeCo

Run ExecuteMain.class file as a java application and observe the console output.

c. Case Study-2

(1) Scenario Description

According to the second case study, cleaner will have dirtiness sensation in addition to the first case study. It will move around randomly and sense the dirtiness from the environment.

(2) DeCo Model Design

We currently have a cleaner agent that can move around. Now we want our agent to look for waste. Firstly our cleaner needs an action ‘Sense’ to sense the dirtiness and move around to look for waste. We will add our model LookForWaste goal and this goal will be facilitated by

MoveAroundGoal and Sense action.

When we add the LookForwaste goal and Sense action to our model, the activations of nodes will be as shown in Fig. 21.

When the priority of the LookForWaste goal is 1, all of the units become activated. When the priority of the LookForWaste goal is 0, all of the units become deactivated. So our agent has the capability of both moving around and sensing the dirtiness.

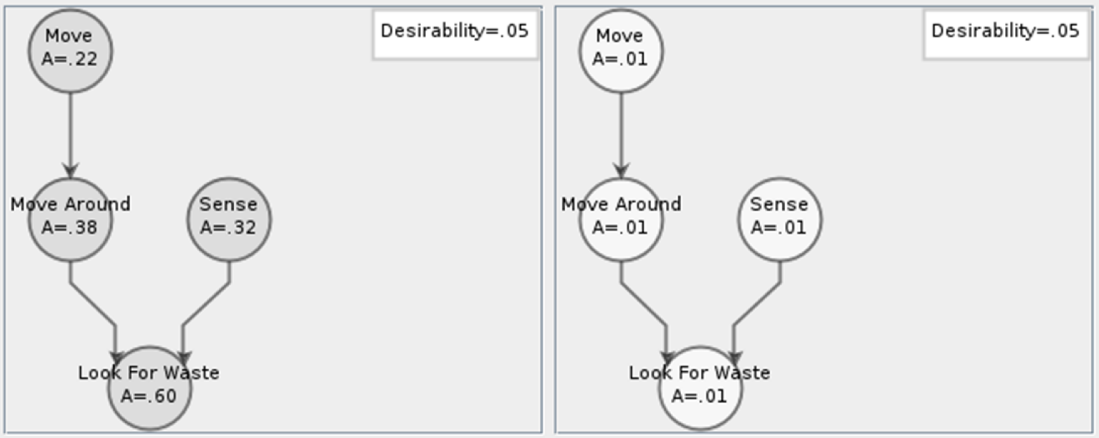

Figure 21 The Appearance of the DeCo Model Activations

(3) Implementation

We have created our DeCo model, now we will implement our model in our agent definition file. Open the agent definition file, CleanerDeCo.java. We need two beliefs which are the sense and the environment, where one of them is related with the amount of the dirtiness and the other is related with the time slot of the sense.

Add the 'sensedDirt', 'time' belief and their setter / getter functions to our agent definition.

```
@Belief
private double sensedDirt;

@Belief(updaterate=200)
private long time=System.currentTimeMillis();

public double getSensedDirt()
{
	return sensedDirt;
}

public void setSensedDirt(double sensedDirt)
{
	this.sensedDirt = sensedDirt;
}
```

Figure 22 Beliefs to Sense the Environment

We have added the necessary beliefs to our agent. Now, we will add the Sense goal. Sense goal will be created for every time event. To make this happen, we use *@GoalCreationCondition* annotation.

Add the following code to the agent definition file.

```
@Goal()
public class Sense
{
	@GoalCreationCondition(events="time")
	public Sense()
	{
		//
	}
}
```

Figure 23 Sense Goal Definition

To create Sense plan, create a new java document named as SensePlan.java. In every execution of the SensePlan, agent will take the environment and will sense the dirtiness of the environment's current position.

Add the following code to the SensePlan.java.

```
package deco.jadex.examples.cleaner;

import jadex.commons.future.Future;
import jadex.commons.future.IFuture;
import deco.jadex.bdiv3.annotation.Plan;
import deco.jadex.bdiv3.annotation.PlanAPI;
import deco.jadex.bdiv3.annotation.PlanBody;
import deco.jadex.bdiv3.annotation.PlanCapability;
import deco.jadex.bdiv3.runtime.IPlan;

@Plan
public class SensePlan
{
	@PlanCapability
	protected CleanerDeCo capa;

	@PlanAPI
	protected IPlan rplan;

	public SensePlan()
	{
		//
	}

	@PlanBody
	public IFuture<Void> body()
	{
		final Future<Void> ret = new Future<Void>();
		Room room=Room.getInstance();
		double dirtiness=room.getDirtiness(capa.getxPos(),
		  capa.getyPos());
		System.out.println("Dirtiness:"+dirtiness);
		capa.setSensedDirt(dirtiness);
		return ret;
	}
}
```

Figure 24 SensePlan.java Definition

We need an instance of environment to store the dirtiness. Create Room.java class file and

add a static instance of the room to get the room class anywhere in synchronized. Add the dirtiness as a double grid array and initialize the dirtiness of the room at the constructor. Add a synchronized vacuum function to the class in order to provide the controlled access.

Add the following code Room.java class file.

```
package deco.jadex.examples.cleaner;

import java.util.Random;

public class Room
{
	protected static Room instance;
	protected double[][] dirtiness;
	protected int size=100;

	public Room()
	{
		//Initialize the room dirtiness
		dirtiness=new double[100][100];
		for(int i=0;i<size;i++)
		{
			Random generator=new
			 Random((int)System.currentTimeMillis());
			for(int j=0;j<size;j++)
			{
				double dirt=(1 + generator.nextGaussian())/2;
				if(dirt<=0)
					dirt=0;
				dirtiness[i][j]=dirt;
			}
		}
	}

	public static synchronized Room getInstance()
	{
		if(instance==null)
		{
			instance = new Room();
		}
		return instance;
	}

	public double getDirtiness(double x, double y)
	{
		int xIdx=(int)(x*size);
		int yIdx=(int)(y*size);
		return dirtiness[xIdx][yIdx];
	}

	public synchronized void vacuum(double x, double y, double value)
	{
		int xIdx=(int)(x*size);
		int yIdx=(int)(y*size);
		dirtiness[xIdx][yIdx]-=value;
	}
}
```

Figure 25 Room.java Definition

Open the CleanerDeCo.java file and add the LookForWaste goal as an inner class. Our LookForWaste goal must be facilitated by the MoveAround and Sense goals. Its priority will be 1.

Add the following LookForWaste definition to the agent definition file.

```
@Goal(excludemode=ExcludeMode.Never, succeedonpassed=false,
                relations=@Relations(facilitations=
                {
                        @Facilitate(units={MoveAround.class},degree=1),
                        @Facilitate(units={Sense.class},degree=1)
                }),priority=1.0)
public class LookForWaste
{
        public LookForWaste()
        {
                //
        }
}
```

Figure 26 LookForWaste Goal Definition

In order to create LookForWaste plan, create a new java document named as LookForWastePlan.java.

```
package deco.jadex.examples.cleaner;

import jadex.commons.future.ExceptionDelegationResultListener;
import jadex.commons.future.Future;
import jadex.commons.future.IFuture;
import deco.jadex.bdiv3.annotation.Plan;
import deco.jadex.bdiv3.annotation.PlanAPI;
import deco.jadex.bdiv3.annotation.PlanBody;
import deco.jadex.bdiv3.annotation.PlanCapability;
import deco.jadex.bdiv3.runtime.IPlan;
import deco.jadex.examples.cleaner.CleanerDeCo.MoveAround;

@Plan
public class LookForWastePlan
{
	@PlanCapability
	protected CleanerDeCo capa;

	@PlanAPI
	protected IPlan rplan;

	public LookForWastePlan()
	{
		//
	}

	@PlanBody
	public IFuture<Void> body()
	{
		System.out.println("Cleaner started to execute Look For Waste
		  plan.");
		final Future<Void> ret = new Future<Void>();
		IFuture<MoveAround> fut = rplan.dispatchSubgoal(capa.new
		  MoveAround());
		fut.addResultListener(new
		  ExceptionDelegationResultListener<CleanerDeCo.MoveAround,
		  Void>(ret)
		{
			public void customResultAvailable(MoveAround amt)
			{
				ret.setResult(null);
			}
		});
		return ret;
	}
}
```

Figure 27 LookForWastePlan.java Definition

Add the code in the Fig. 27 to the LookForWastePlan.java file.

Until now we have defined our plans and goals. Now open the CleanerDeCo.java file and add the following plan headings in *@Plans* annotation.

```
	@Plan(trigger=@Trigger(goals=CleanerDeCo.LookForWaste.class),
	body=@Body(LookForWastePlan.class)),
	@Plan(trigger=@Trigger(goals=CleanerDeCo.Sense.class),
	body=@Body(SensePlan.class)),
```

Figure 28 Plan Headings

Now run the ExecutionMain.java file as a java application and observe the results.

d. Case Study-3

(1) Scenario Description

According to the third case study, cleaner will have vacuum actuator in addition to the second case study and will satisfy cleaning performance criteria. It will move around randomly, sense the dirtiness from the environment and vacuum the dirtiness.

(2) DeCo Model Design

Until now, our cleaner agent looked for waste by only moving around and sensing the environment. Now, we will add cleaning ability to our cleaner agent. Our cleaning ability will be facilitated by vacuum action and sense actions. Since the cleaner agent will not be able to handle both of LookForWaste and Clean goals, we add incompatible relationship between these goals.

The appearance of the DeCo model activations, when LookForWaste priority is 1, Clean priority is 0(left) and LookForWaste priority is 0, Clean priority is 1 (right), is depicted in Fig. 29.

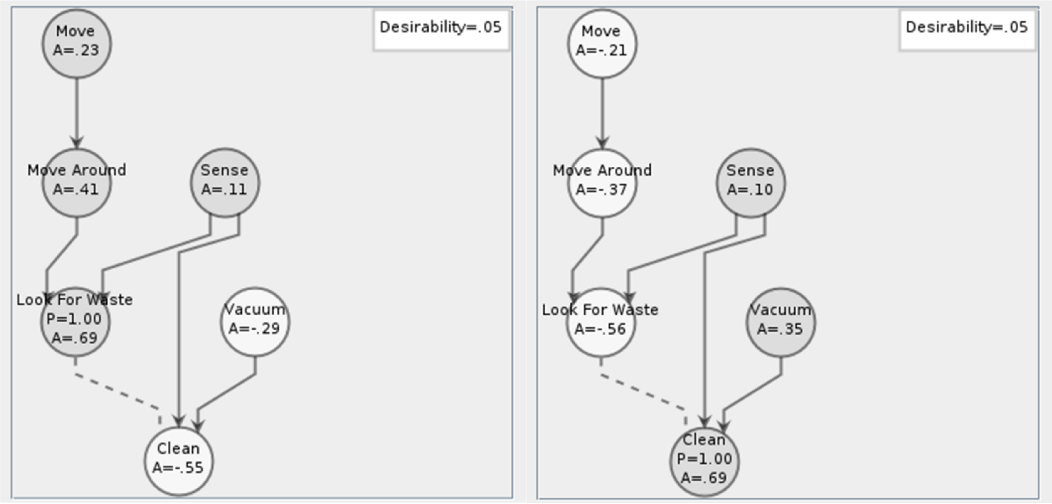

Figure 29 The Appearance of the DeCo Model Activations

(3) Implementation

Open CleanerDeCo.java file and add a clean dynamic belief which is updated on every rate update. This belief will be our creation condition of the clean goal.

```
@Belief(updaterate=200)
private boolean clean=sensedDirt>=1;
```

Figure 30 'clean' Dynamic Belief Definition

Add vacuum goal as an empty goal which is shown in Fig. 30 to the agent definition file.

```
@Goal
public class Vacuum
{
	public Vacuum()
	{

	}
}
```

Figure 31 Vacuum Goal Definition

Create a new java document named as VacuumPlan.java. In VacuumPlan body, we get the current sensation of the agent and check if it is above the dirtiness threshold. If so, agent will vacuum the room with its synchronized function “Vacuum”. When the dirtiness of the room is removed, cleaning goal will be accomplished. After accomplishing the cleaning goal, we must reconfigure our goals’ priorities. We set MoveAround goal priority to 1 and Clean goal priority to 0.

```
package deco.jadex.examples.cleaner;
import jadex.commons.future.Future;
import jadex.commons.future.IFuture;
import deco.jadex.bdiv3.annotation.Plan;
import deco.jadex.bdiv3.annotation.PlanAPI;
import deco.jadex.bdiv3.annotation.PlanBody;
import deco.jadex.bdiv3.annotation.PlanCapability;
import deco.jadex.bdiv3.runtime.IPlan;
import deco.jadex.examples.cleaner.Room;
import deco.jadex.examples.cleaner.CleanerDeCo.Clean;
import deco.jadex.examples.cleaner.CleanerDeCo.LookForWaste;

@Plan
public class VacuumPlan
{
	@PlanCapability
	protected CleanerDeCo capa;

	@PlanAPI
	protected IPlan rplan;

	public VacuumPlan()
	{
	}

	@PlanBody
	public IFuture<Void> body()
	{
		final Future<Void> ret = new Future<Void>();

		Room room=Room.getInstance();
		while(capa.getSensedDirt()>=0.1)
		{
			room.vacuum(capa.getxPos(), capa.getyPos(),0.025);
			rplan.waitFor(100).get();
		}

		capa.agent.setPriority(Clean.class, 0.0);
		capa.agent.setPriority(LookForWaste.class, 1.0);

		return ret;
	}
}
```

Figure 32 VacuumPlan.java Definition

We can reconfigure the agent DeCoModel from any plan by updating the priorities and relationships of the goals. In order to access these functions we must use API of the DeCoAgent.

setPriority(Class<?> goal, double priority) function takes two parameters. First one is for goal class definition and the second one is for priority.

Open CleanerDeCo.java file. Create a new goal as an inner class and name it as “Clean”. Define our DeCoModel relations in ‘relations’ field of the `@Goal` annotation. Our Clean goal must be facilitated by Vacuum, Sense goals must be incompatible with LookForWaste goal. Our Clean goal will be created, once our previously defined belief ‘clean’ creates an event. Our clean goal must be unique and must inhibit the other Clean goals that are dispatched. As can be done in BDIAgent, we can use ‘deliberation’ field of the `@Goal` annotation to define uniqueness.

Add the following Clean class definition to CleanerDeCo.java file.

```
@Goal(unique=true, deliberation=@Deliberation(inhibits={Clean.class}),
                    relations=@Relations(facilitations=
                    {
                           @Facilitate(units={Vacuum.class},degree=1),
                           @Facilitate(units={Sense.class},degree=1)
                    },incompatibles=
                    {
                           @Incompatible(unit=LookForWaste.class,degree=1)
                    }),priority=0.0)
      public class Clean
      {
            @GoalCreationCondition(events="clean")
            public Clean()
            {
                  //
            }

            public int hashCode()
            {
                  final int prime = 31;
                  int result = 1;
                  result = prime * result + getOuterType().hashCode();
                  result = prime * result + 0;
                  return result;
            }

            public boolean equals(Object obj)
            {
                  boolean ret = false;
                  if(obj instanceof Clean)
                  {
                        Clean other = (Clean)obj;
                        ret =
getOuterType().equals(other.getOuterType());
                  }
                  return ret;
            }

            private CleanerDeCo getOuterType()
            {
                  return CleanerDeCo.this;
            }
```

Figure 33 Clean Goal Definition

Create a new java file and name it as CleanPlan.java. Our CleanPlan body will only dispatch Vacuum goal as a sub goal.

Add the following code to CleanPlan.java file.

```
package deco.jadex.examples.cleaner;

import jadex.commons.future.ExceptionDelegationResultListener;
import jadex.commons.future.Future;
import jadex.commons.future.IFuture;
import deco.jadex.bdiv3.annotation.Plan;
import deco.jadex.bdiv3.annotation.PlanAPI;
import deco.jadex.bdiv3.annotation.PlanBody;
import deco.jadex.bdiv3.annotation.PlanCapability;
import deco.jadex.bdiv3.runtime.IPlan;
import deco.jadex.examples.cleaner.CleanerDeCo.Vacuum;

@Plan
public class CleanPlan
{
	@PlanCapability
	protected CleanerDeCo capa;

	@PlanAPI
	protected IPlan rplan;

	public CleanPlan()
	{
		//
	}

	@PlanBody
	public IFuture<Void> body()
	{
		System.out.println("Cleaner started to execute Clean plan.");

		final Future<Void> ret = new Future<Void>();

		IFuture<Vacuum> fut = rplan.dispatchSubgoal(capa.new
Vacuum());
		fut.addResultListener(new
		  ExceptionDelegationResultListener<CleanerDeCo.Vacuum,
	    Void>(ret)
		{
			public void customResultAvailable(Vacuum amt)
			{
				ret.setResult(null);
			}
		});

		return ret;
	}
```

Figure 34 CleanPlan.java Definition

Until now we have defined our plans and goals. Now open the CleanerDeCo.java file and add the following plan headings in *@Plans* annotation.

```
	  @Plan(trigger=@Trigger(goals=CleanerDeCo.Clean.class),
	body=@Body(CleanPlan.class)),
	  @Plan(trigger=@Trigger(goals=CleanerDeCo.Vacuum.class),
	body=@Body(VacuumPlan.class)),
```

Figure 35 Plan Headings

Changing the priorities of the goals is the key point of the DeCoAgent goal deliberation process. When a scheduled action has been taken, agent must change the priorities of the goals according to our created deliberative coherence model. According to our agent model, our agent should start to execute the Clean goal after sensing the dirtiness. In order to implement this, open the SensePlan.java file and update the body of the plan as in Fig. 36.

```
@PlanBody
public IFuture<Void> body()
{
	final Future<Void> ret = new Future<Void>();
	Room room=Room.getInstance();
	double dirtiness=room.getDirtiness(capa.getxPos(), capa.getyPos());
	double batteryLevel=capa.getBatteryLevel();
	System.out.println("Dirtiness:"+dirtiness+
	    " BateryLevel:"+batteryLevel);
	capa.setSensedDirt(dirtiness);
	if(dirtiness>=1)
	{
		capa.agent.setPriority(Clean.class, 1.0);
		capa.agent.setPriority(LookForWaste.class, 0.0);
	}
	return ret;
}
```

Figure 36 Updated SensePlan Body

Now run the ExecutionMain.java file as a java application and observe the results.

e. Case Study-4

(1) Scenario Description

According to the fourth case study, cleaner will satisfy keep battery loaded performance criteria in addition to the third case study. It will move around randomly, sense the dirtiness from the environment, vacuum the dirtiness and keep battery loaded.

(2) DeCo Model Design

Until now our cleaner looks for waste in an environment and cleans the dirtiness. From now on to make our agent a real cleaner, we will add the charging ability to the cleaner. We will add KeepBatteryLoaded goal to keep the battery level of the agent above a level. Additionally we will add ReturnHome goal to return the cleaner to the battery station. Our KeepBattery goal will be facilitated by Return Home goal and will be incompatible with LookForWaste and Clean goals. Our ReturnHome goal will be facilitated by Move goal.

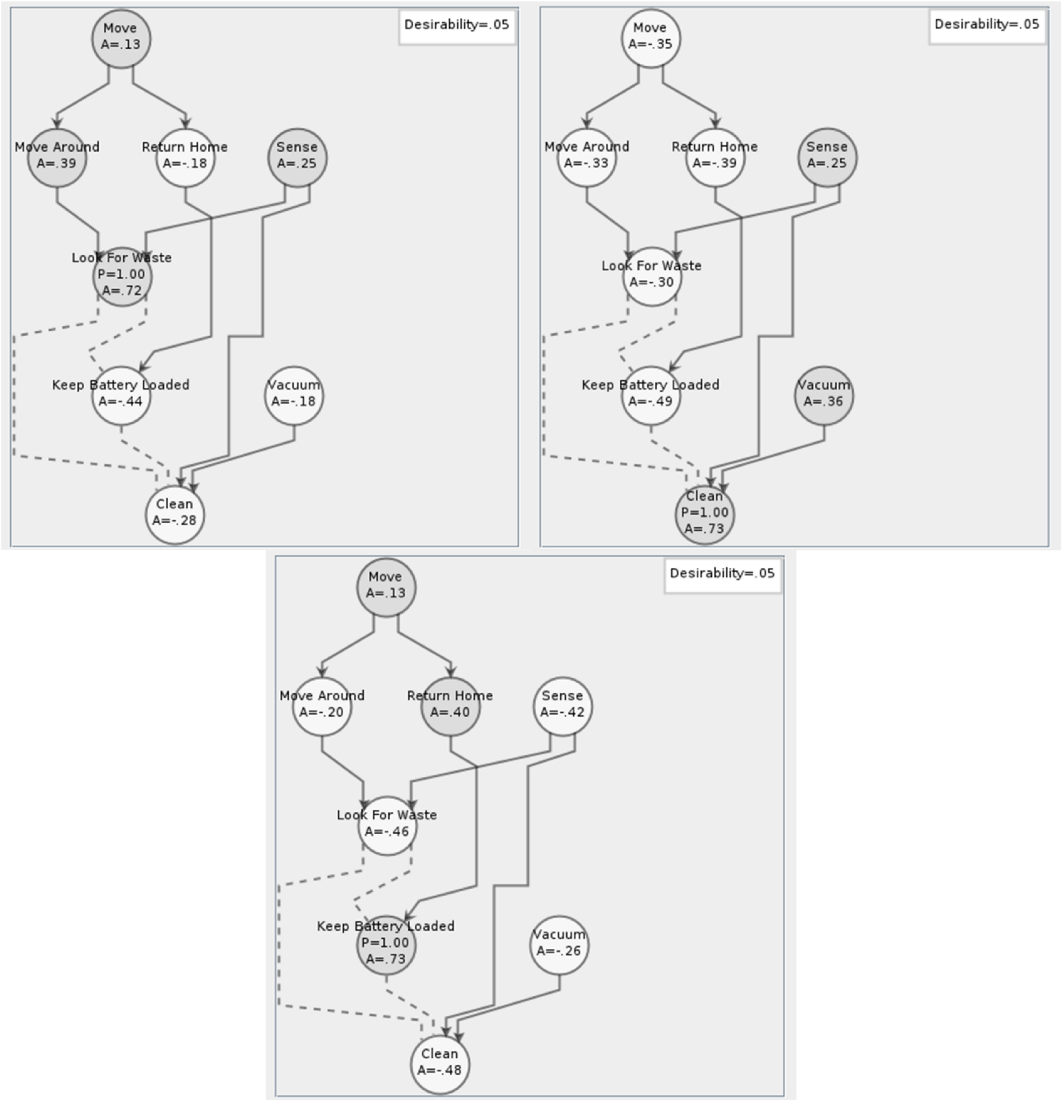

Figure 37 DeCo Model Activations

The appearance of the DeCo model activations. LookForWaste priority is 1, Clean priority is 0 and KeepBatteryLoaded priority is 0 (left). LookForWaste priority is 0, Clean priority is 1 and KeepBatteryLoaded priority is 0 (right). LookForWaste priority.

(3) Implementation

Open CleanerDeCo.java file and add ‘batteryLevel’ belief to store the battery level of the agent and ‘recharge’ dynamic belief to define creation condition of the MaintainBatteryLoaded goal.

```
	@Belief
	private double batteryLevel=1.0;

	@Belief(updaterate=100)
	private boolean recharge=batteryLevel<=0.4;

	public double getBatteryLevel()
	{
		return batteryLevel;
	}

	public void setBatteryLevel(double batteryLevel)
	{
		this.batteryLevel=batteryLevel;
	}
```

Figure 38 'batteryLevel' and 'recharge' Beliefs

Add the ReturnHome goal to our agent definition file. Since the ReturnHome goal is facilitated by Move goal, define a facilitation relationship between them.

```
@Goal(relations=@Relations(facilitations=
{
	@Facilitate(units={Move.class},degree=1)
}))
public class ReturnHome
{
	public ReturnHome()
	{
		//
	}
}
```

Figure 39 ReturnHome Goal Definition

In order to define our plan for ReturnHome goal, create a new java document named as ReturnHomePlan.java. In our ReturnHomePlan body, we will dispatch a ‘move’ goal with our charging station position.

```
package deco.jadex.examples.cleaner;

import jadex.commons.future.ExceptionDelegationResultListener;
import jadex.commons.future.Future;
import jadex.commons.future.IFuture;
import deco.jadex.bdiv3.annotation.Plan;
import deco.jadex.bdiv3.annotation.PlanAPI;
import deco.jadex.bdiv3.annotation.PlanBody;
import deco.jadex.bdiv3.annotation.PlanCapability;
import deco.jadex.bdiv3.runtime.IPlan;
import deco.jadex.examples.cleaner.CleanerDeCo.Move;

@Plan
public class ReturnHomePlan
{
	@PlanCapability
	protected CleanerDeCo capa;

	@PlanAPI
	protected IPlan rplan;

	public ReturnHomePlan()
	{
		//
	}

	@PlanBody
	public IFuture<Void> body()
	{
		final Future<Void> ret = new Future<Void>();

		double xDest = 0;
		double yDest = 0;

		IFuture<Move> fut = rplan.dispatchSubgoal(capa.new
		  Move(xDest,yDest));
		fut.addResultListener(new
		  ExceptionDelegationResultListener<CleanerDeCo.Move,
		  Void>(ret)
		{
			public void customResultAvailable(Move amt)
			{
				ret.setResult(null);
			}
		});

		return ret;
	}
}
```

Figure 40 ReturnHomePlan.java Definition

Open the CleanerDeCo.java file and add the KeepBatteryLoaded goal as an inner class. Since the KeepBatteryLoaded goal is facilitated by ReturnHome goal, we will define a facilitation relationship between them. Additionally, we will define 2 incompatible relationships between KeepBatteryLoaded, LookForWaste and Clean goals. Our KeepBatteryLoaded goal will be unique and created with ‘recharge’ event.

```
@Goal(unique=true,deliberation=@Deliberation(inhibits=
      {KeepBatteryLoaded.class})
      ,relations=@Relations(facilitations=
       {
              @Facilitate(units={ReturnHome.class},degree=1)
       },incompatibles=
       {
              @Incompatible(unit=LookForWaste.class,degree=1),
              @Incompatible(unit=Clean.class,degree=1)
       }),priority=0.0)
public class KeepBatteryLoaded
{
       @GoalCreationCondition(events="recharge")
       public KeepBatteryLoaded()
       {
              //
       }

       public int hashCode()
       {
              final int prime = 31;
              int result = 1;
              result = prime * result + getOuterType().hashCode();
              result = prime * result + 0;
              return result;
       }

       public boolean equals(Object obj)
       {
             boolean ret = false;
             if(obj instanceof KeepBatteryLoaded)
             {
                    KeepBatteryLoaded other = (KeepBatteryLoaded)obj;
                    ret = getOuterType().equals(other.getOuterType());
             }
             return ret;
       }

             private CleanerDeCo getOuterType()
             {
                    return CleanerDeCo.this;
             }
       }
}
```

Figure 41 KeepBatteryLoaded Goal Definition

After adding our KeepBatteryLoaded goal to our agent definition file. Create a new java document and name it as KeepBatteryLoadedPlan.java. In our plan body we will dispatch a ReturnHome goal and wait for the agent, to return to the charging station. After agent returns to the charging station, agent recharges itself. When recharging is completed, agent will change the priorities of the goals in order to start LookForWaste goal.

```
package deco.jadex.examples.cleaner;

import jadex.commons.future.Future;
import jadex.commons.future.IFuture;
import deco.jadex.bdiv3.annotation.Plan;
import deco.jadex.bdiv3.annotation.PlanAPI;
import deco.jadex.bdiv3.annotation.PlanBody;
import deco.jadex.bdiv3.annotation.PlanCapability;
import deco.jadex.bdiv3.runtime.IPlan;
import deco.jadex.examples.cleaner.CleanerDeCo.Clean;
import deco.jadex.examples.cleaner.CleanerDeCo.KeepBatteryLoaded;
import deco.jadex.examples.cleaner.CleanerDeCo.LookForWaste;

@Plan
public class KeepBatteryLoadedPlan
{
	@PlanCapability
	protected CleanerDeCo capa;

	@PlanAPI
	protected IPlan rplan;

	public KeepBatteryLoadedPlan()
	{
		//
	}

	@PlanBody
	public IFuture<Void> body()
	{
		System.out.println("Battery is low, recharging is needed!");
		System.out.println("Returning to battery charge station");

		final Future<Void> ret = new Future<Void>();

		rplan.dispatchSubgoal(capa.new ReturnHome()).get();

		double batteryLevel = capa.getBatteryLevel();
		while(batteryLevel<1)
		{
			batteryLevel = Math.min(batteryLevel + 0.01, 1.0);
			capa.setBatteryLevel(batteryLevel);

	System.out.println("Charging...."+capa.getBatteryLevel());
			rplan.waitFor(100).get();
		}
		capa.agent.setPriority(KeepBatteryLoaded.class, 0.0);
		capa.agent.setPriority(LookForWaste.class, 1.0);
		capa.agent.setPriority(Clean.class, 0.0);

		return ret;
	}
`
```

Figure 42 KeepBatteryLoadedPlan.java Definition

For goal deliberation process we must control our agent battery level continuously. If battery level drops under a defined threshold we must change goals' priorities to activate designed goals according to our created deliberative coherence model.

Open the SensePlan.java file and update the body of the plan as in Fig. 43.

```
@PlanBody
public IFuture<Void> body()
{
	final Future<Void> ret = new Future<Void>();
	Room room=Room.getInstance();
	double dirtiness=room.getDirtiness(capa.getxPos(), capa.getyPos());
	double batteryLevel=capa.getBatteryLevel();
	System.out.println("Dirtiness:"+dirtiness+
	    " BateryLevel:"+batteryLevel);
	capa.setSensedDirt(dirtiness);
	if(dirtiness>=1)
	{
		capa.agent.setPriority(Clean.class, 1.0);
		capa.agent.setPriority(LookForWaste.class, 0.0);
	}

	if(batteryLevel<=0.4)
	{
		capa.agent.setPriority(Clean.class, 0.0);
		capa.agent.setPriority(LookForWaste.class, 0.0);
		capa.agent.setPriority(KeepBatteryLoaded.class, 1.0);
	}

	return ret;
}
```

Figure 43 Updated SensePlan body.

Open the CleanerDeCo.java file and add the following plan headings in *@Plans* annotation.

```
	@Plan(trigger=@Trigger(goals=CleanerDeCo.KeepBatteryLoaded.class),
      body=@Body(KeepBatteryLoadedPlan.class)),
	@Plan(trigger=@Trigger(goals=CleanerDeCo.ReturnHome.class),
      body=@Body(ReturnHomePlan.class))
```

Figure 44 Plan Headings

Now run the ExecutionMain.java file as a java application and observe the results.