MultiAgent-based Simulation

Repast Simphony
Recursive Porous Agent Simulation Toolkit
Argonne National Laboratory
University of Chicago
http://repast.sourceforge.net/
mirror : https://seafile.emse.fr/d/0a454872c8/

Amro Najjar
Hubert Curien Laboratory

This course is based on a course by
Mahdi Zargayouna – hamza-mahdi.zargayouna@ifsttar.fr
Repast is a multi-agent platform used to develop, test and execute multi-agent systems
- Written in Java (open-source)
- Very popular in the MAS simulation community, mainly for social and geographic simulations
- Offers an API for the design and implementation of MAS
- Offers a graphical modeling tool (specially for beginners)
  - 2D & 3D visualization
  - Grid management
  - Parameter management
  - Data observation
  - Data can be analyzed and exported from a format to another
- Availability of machine learning libraries (neural networks, regression)
- Import & Export of shapefile
Key Notions: context

- A fundamental concept in Repast Simphony
  - Offers a data structure allowing to organize agents
  - Concretely, contexts are a collection of hierarchically nested agents each containing some of the components of the model
  - These components are java objects
  - Possibility to have other contexts
  - A component can be involved in a multiple contexts at once
  - A component involved in one context is also involved in all contexts having a parent relationship with this context
  - At least there is one master context that contains all the others
Key Notions: projection

- Whereas contests create a collection allowing to organize agents, projections impose a structure on these agents.
- Projections allow to create a structure that define relations (can be spatial or semantic).
- A projection is attached to a particular context and is applied to all the agents belonging to this context.
  - An object must belong to a context before being used in a projection.
  - Multiple projections can be applied to the same context. Thus, we can have a context that contains a grid and to «geographical» spaces (e.g. two representations of the same transport network).
Contexts and Projectons

Context - a “soup” of agents

GIS Projection - gives agents a geometry

Network Projection - gives agents a network
Example: Conway’s Game of Life

- Is a cellular automaton devised by the British mathematician John Horton Conway in 1970
- is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input
- You create an initial configuration and observing how it evolves
- Rules
  - an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead
  - Every cell interacts with its eight neighbours
  - Any live cell with fewer than two live neighbours dies (underpopulation)
  - Any live cell with two or three live neighbours lives on to the next generation
  - Any live cell with more than three live neighbours dies
  - Any dead cell with exactly three live neighbours becomes a live cell
Example: Conway’s Game of life

- New Repast Simphony Project
- GameOfLife
- Perspective java
- In « GameOfLife.rs », « context.xml », create the projection« Grid »

```xml
  <projection type="grid" id="grid"></projection>
</context>
```

- In this configuration file, we can create all the contexts, the subcontexts and projections
Create the Class Agent

```java
import repast.simphony.space.grid.Grid;

public abstract class Agent {
    protected Grid<Agent> grid;
    protected boolean alive;

    public Agent(Grid<Agent> grid) {
        this.grid = grid;
        alive = true;
    }
    // modify the value of alive according to neighborhood
    public abstract void compute();

    // create complementary type of agent
    public abstract void implement();
}
```

- We will have different agent representation based on the agent types, for this we will create two classes extending agents
  - DeadAgent dead cells
  - AliveAgent living cells
Create A ContextBuilder

- The principal class for Repast
- A generic class of which we specify the type of object that it contains

```java
public class ContextCreator implements ContextBuilder<Agent> {

    @Override
    public Context<Agent> build(Context<Agent> context) {
        ...
    }
}
```
Create A projection

- As a general rule, *Projections* are created as follows:
  - Find their « factory »
  - Use the *factory* to create the *projection*

- Each *Factory* creates a projection of a specific type and requires the context to which the projection is associated, the name of the projection as well as some supplementary arguments.

- Projection for GameOfLife

```java
GridFactory gridFactory = GridFactoryFinder.createGridFactory(null);
Grid<Agent> grid = gridFactory.createGrid("grid", context,
new GridBuilderParameters<Agent>(new WrapAroundBorders(),
new SimpleGridAdder<Agent>(), false, 50, 50));
```
Execute GameOfLife Model
Define the principal context

➢ Relancer le modèle

Select Data Source Type
Choose a type for your data source.

- Context.xml file
- Context.xml file & ModelInitializer
- Custom ContextBuilder Implementation
- Freeze Dried Simulation - Database Format
- Freeze Dried Simulation - Plain Text Delimited Format
- Freeze Dried Simulation - XML Format

Class Name
Please provide the name of the context creator class

Class Name
- gameoflife.ContextCreator
  - gameoflife.ContextCreator
Define the visualization
Agents’ styles
Parametering the Simulation

![Parameter settings in Repast Simphony](image)
Parametering the Simulation

Enregistrer le modèle
public class ContextCreator implements ContextBuilder<Agent> {
    @Override
    public Context<Agent> build(Context<Agent> context) {
        context.setId("GameOfLife");
        int width = RunEnvironment.getInstance().getParameters().getInteger("gridWidth");
        int height = RunEnvironment.getInstance().getParameters().getInteger("gridHeight");

        GridFactory gridFactory = GridFactoryFinder.createGridFactory(null);
        Grid<Agent> grid = gridFactory.createGrid("grid", context, new GridBuilderParameters<Agent>(new WrapAroundBorders(), new SimpleGridAdder<Agent>(), false, width, height));

        for (int x = 0; x < height; x++) {
            for (int y = 0; y < width; y++) {
                boolean b = Math.random() > 0.5 ? true : false;
                Agent a = b ? new AliveAgent(grid) : new DeadAgent(grid);
                context.add(a);
                grid.moveTo(a, x, y);
            }
        }
        return context;
    }
}
There exists three ways to work with Repast Scheduler

- Schedule an action directly:

```java
// Récupérer une référence vers l'ordonnanceur
ISchedule schedule = RunEnvironment.getInstance().getCurrentSchedule();

// Spécifier que l'action devrait commencer au tick 1 et s'exécuter à chaque tick
ScheduleParameters params = ScheduleParameters.createRepeating(1, 2);

// L'agent exécute la méthode move avec les paramètres d'ordonnancement définis
// plus haut
schedule.schedule(params, myAgent, "move");
```

- As you can see, ScheduleParameters can be created using one of its class methods:

```java
// Spécifier que l'action devrait commencer au tick 1 et s'exécuter à chaque tick
ScheduleParameters params = ScheduleParameters.createRepeating(1, 2);
// L'agent exécute la méthode move avec les paramètres d'ordonnancement définis
schedule.schedule(params, myAgent, "move", "Forward", 4);
```
Scheduling in Repast (2/3)

Scheduling with annotations

```java
@ScheduledMethod(start=1, interval=2)
public void deliverPaper()
```

Most of the time, objects with annotations are automatically added to the scheduler. However, if we create an object will the simulation is running, it may be not the case.

For this reason, we can user the « schedule » object to add them

```java
//Ajouter les méthodes annotées de l'agent au schedule.
schedule.schedule(myAgent);
```
public abstract class Agent {
protected Grid<Agent> grid;
protected boolean alive;

public Agent(Grid<Agent> grid2) {
    this.grid = grid2;
    alive = true;
}

@ScheduledMethod(start = 1, interval = 1, priority = 2)
public abstract void compute();

@ScheduledMethod(start = 1, interval = 1, priority = 1)
public abstract void implement();
}
Scheduling in Repast (3/3)

- Scheduling with the « Watchers »

- Watchers are designed to be used to schedule in a dynamic manner.
- A watcher allows an agent to be notified about a change of status of another agent and thus to plan an event that will take place subsequently.
- (e.g. the agent « A » decides that if agent « B » does an action it will make a reaction)

- The watcher is used via an annotation:
  - A request defining the agents to observe
  - A request defining the condition that should be met in order to trigger and execute the action.
watcheeClassName defines the type of agents that this object is going to watch.
watcheeFieldNames defines the attribute to be watched. When this attribute changes, this object will be notified.
query: determines (or filter) the instances of Simple happy agents to be watched
Here, we watch agents that are linked to us.
whenToTrigger specifies if the action should be executed immediately (before other actions scheduled at this instant are executed) or wait for the next tick.
scheduleTriggerDelta defines how many ticks to wait before scheduling the actions.
scheduleTriggerPriority defines the priority of the action.
public class AliveAgent extends Agent{
public AliveAgent(Grid<Agent> grid) {
    super(grid);
    alive = true;
}

public void compute() {
    MooreQuery<Agent> query = new MooreQuery<Agent>(grid, this);
    int neighbours = 0;
    for (Agent o : query.query())
        if (o instanceof AliveAgent)
            neighbours++;
    if (neighbours != 2 && neighbours != 3)
        alive = false;
}

public void implement() {
    if (!alive) {
        GridPoint gpt = grid.getLocation(this);
        Context<Object> context = ContextUtils.getContext(this);
        context.remove(this);
        DeadAgent a = new DeadAgent(grid);
        context.add(a);
        grid.moveTo(a, gpt.getX(), gpt.getY());
    }
}
public class DeadAgent extends Agent{
    public DeadAgent(Grid<Agent> grid) {
        super(grid);
        alive = false;
    }

    public void compute() {
        MooreQuery<Agent> query = new MooreQuery<Agent>(grid, this);
        int neighbours = 0;
        for (Agent o : query.query())
            if (o instanceof AliveAgent)
                neighbours++;
        if (neighbours == 3)
            alive = true;
    }

    public void implement() {
        if (alive) {
            GridPoint gpt = grid.getLocation(this);
            Context<Agent> context = ContextUtils.getContext(this);
            context.remove(this);
            AliveAgent a = new AliveAgent(grid);
            context.add(a);
            grid.moveTo(a, gpt.getX(), gpt.getY());
        }
    }
}
Executing the model
Run options

- Schedule Options
  - Pause At: 0
  - Stop At: 0
  - Schedule Tick Delay
    - 0 10 20 30 40 50 60 70 80 90 100

- Sparkline Options
  - Sparkline Points: 50
  - Sparklines are Drawn as Line...
public class AgentStyle2D extends DefaultStyleOGL2D {

    @Override
    public Color getColor(Object o) {
        if(RunEnvironment.getInstance().getCurrentSchedule().getTickCount()%2==0)
            return Color.RED;
        else
            return Color.BLUE;
    }
}
public class AgentStyle2D extends DefaultStyleOGL2D {

@Override
public Color getColor(Object o) {

if (o instanceof AliveAgent) {
    if (((AliveAgent) o).getAge() >= 100)
        return Color.BLACK;

if (RunEnvironment.getInstance().getCurrentSchedule().getTickCount() % 2 == 0)
    return Color.RED;
else
    return Color.BLUE;
}
}
Dynamic Agent Styles
Data Sets
Data Observation and Export
Data Observation and Export
Data Observation and Export
Data Observation and Export
Data Observation and Export
Data Observation and Export
Data Observation and Export
Data Observation and Export
Data Observation and Export
Data Observation and Export

- Time Series Editor
  - Configure Chart Properties
    - Title: Dead/Alive Proportions
    - X-Axis: Tick Count
    - Y-Axis: Agents Number
  - Plot Properties
    - Background Color: 
    - Show Grid Lines: 
    - Grid Line Color: 
    - X-Axis Range: -1
    - Show Legend: 

- Scenario Tree
  - GameOfLife
    - Data Collection Initialization
    - Data Loaders
      - ContextCreator
    - Data Sets
      - A Data Set
    - Displays
      - GoDisplay
    - Miscellaneous Actions
    - Model Initialization
      - Schedule Initialization
    - Random Streams
    - Text Sinks
      - A File Sink
      - User Panel
      - User Specified Actions
Data Observation and Export
Exercice

- Create a new project where vehicles move in a continuous space and change their color when they have more than 3 vehicles around them

**Continuous Space:**

```java
ContinuousSpaceFactory spaceFactory = ContinuousSpaceFactoryFinder.createContinuousSpaceFactory(null);
ContinuousSpace<Voiture> space = spaceFactory.createContinuousSpace("space", context,
                      new RandomCartesianAdder<Voiture>(),
                      repast.simphony.space.continuous.WrapAroundBorders(), 50, 50);
```

**Getting the location**

- NdPoint myPoint = space.getLocation(this);

**Move Action**

- Space:
  - NdPoint otherPoint = new NdPoint(50*Math.random(), 50 *Math.random() );
  - double angle = SpatialMath.calcAngleFor2DMovement(space, myPoint, otherPoint);
  - space.moveByVector(this, 1, angle, 0);
- Grid
  - myPoint = space.getLocation(this);
  - grid.moveTo(this, (int) myPoint.getX(), (int) myPoint.getY());