Demo: Game Of Life

2019.3 Version 1.0

1 Description

1.1 Introduction

This tutorial will introduce a series of examples demonstrating several ways to build an Agent Based Modeling of HPC model. Its purpose is to showcase a set of features included in the Platform Toolkit and to provide a working demo of how to build models that use these features in different ways. Discuss some of the features related to the platform infrastructure, such as how to set and change model parameters, get and record analog output, and implement different forms of random number generation. In particular, it is convenient to construct an ABM model in which agents interact according to certain structural constraints, including spatial placement and network connectivity.。

1.2 Requirement

This tutorial is for anyone who wants to learn more about the platform and is proficient in the ABM model. For those who are not familiar with the HPC environment, it is mainly for people who understand the basics of Agent-Based Modeling (ABM) and are familiar with the X10 language grammar.

2 Demo: Game Of Life

2.1 Introduction

GameOfLife is the most famous set of rules in the Cellular Automaton/Automata (the idea of the rules can be traced back to von Neumann, alias " GameOfLife"). The state of death or viability of each cell is determined by the eight cells surrounding it.

1. "Small population": If there are less than 2 live neighbors, any living cells will die.

2. "Normal": If there are 2 or 3 live neighbors, any living cells will continue to live.

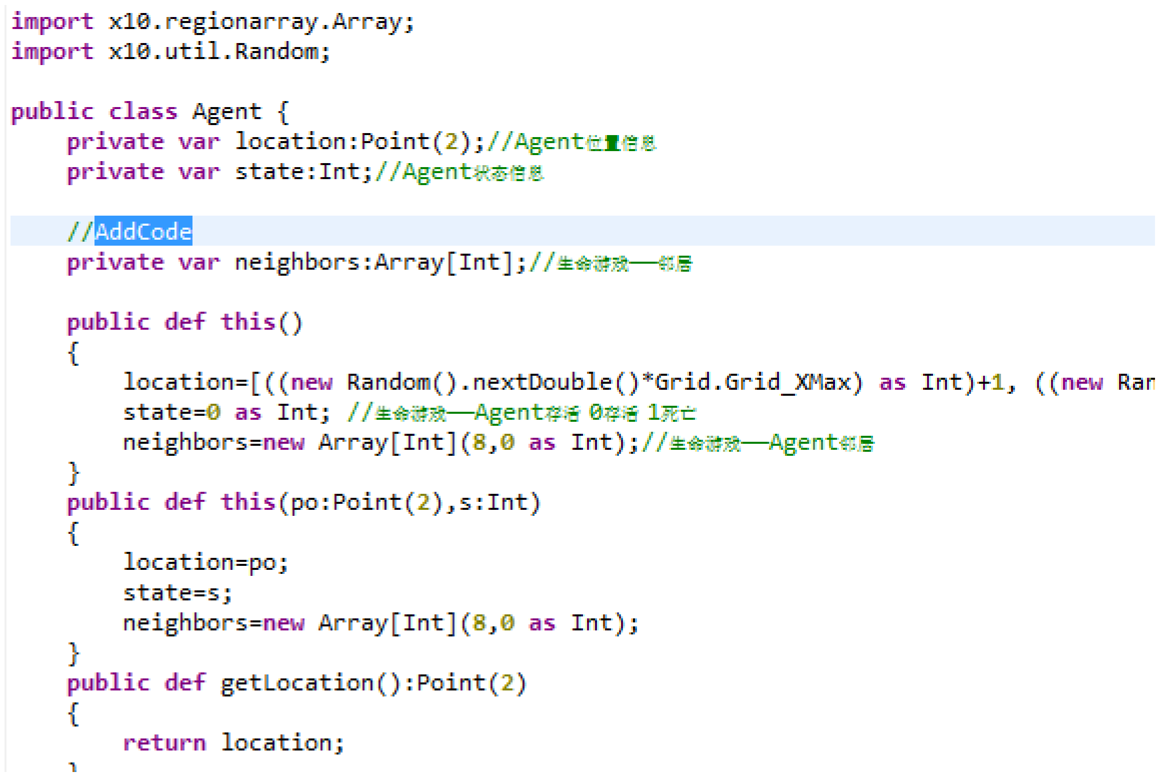
3. "Excessive population": If there are more than 3 live neighbors, any living cells will die.

4. "Breeding": If there are exactly 3 live neighbors, any dead cells will survive.

2.2 Construct

2.2.1 Step 1: Agent

Add all the attributes of the entity in the Agent.xl0 file. The existing location and status attributes are added. Other attributes are added after AddCode.

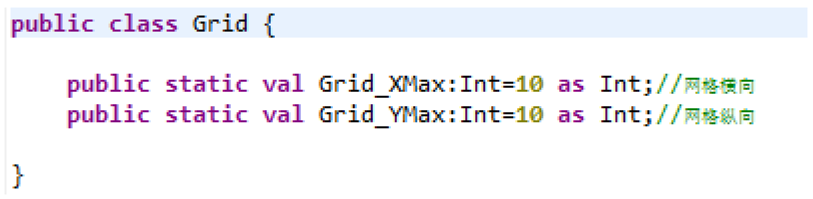


```
import x10.regionarray.Array;
import x10.util.Random;

public class Agent {
    private var location:Point(2);//Agent位置信息
    private var state:Int;//Agent状态信息

    //AddCode
    private var neighbors:Array[Int];//生命游戏——邻居

    public def this()
    {
        location=[((new Random().nextDouble()*Grid.Grid_XMax) as Int)+1, ((new Ran
        state=0 as Int; //生命游戏——Agent存活 0存活 1死亡
        neighbors=new Array[Int](8,0 as Int);//生命游戏——Agent邻居
    }
    public def this(po:Point(2),s:Int)
    {
        location=po;
        state=s;
        neighbors=new Array[Int](8,0 as Int);
    }
    public def getLocation():Point(2)
    {
        return location;
```

In this example, location acquisition, state acquisition, neighbor acquisition, etc. are added as required by the model.

2.2.2 Step 2: Grid

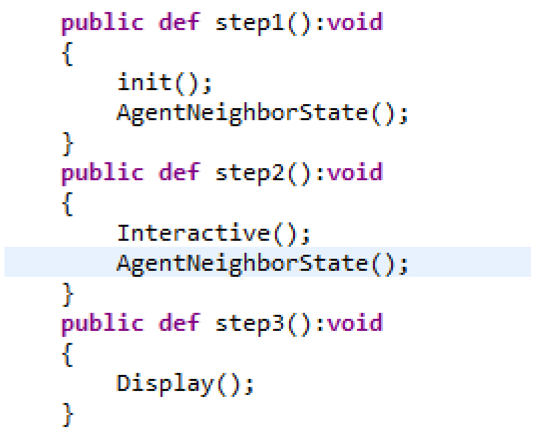
The grid is set in the Grid.x10 file. In this example, the grid is initially set as 10 both horizontal and vertical, for a total of 100 grids.



```
public class Grid {

    public static val Grid_XMax:Int=10 as Int;//网格横向
    public static val Grid_YMax:Int=10 as Int;//网格纵向

}
```

2.2.3 Step 3: Event

The simulation event design is performed in the EventSequence.x10 file. In this Demo, four events are set, which are life game initialization (init()), get agent neighbor information (AgentNeighborState()), and life game evolution according to rules (Interactive( )), agent information display (Display ()) four events, at the same time according to the scheduling steps written to the corresponding step.



```
public def step1():void
{
    init();
    AgentNeighborState();
}
public def step2():void
{
    Interactive();
    AgentNeighborState();
}
public def step3():void
{
    Display();
}
```

2.2.4 Step 4: Model

Schedule design in the Model.x10 file to run the specified event at the specified time.

```
    public var LiveCount:Long;
    public val AgentList:DistArray[Agent](2);
    public val Reg:Region(2);
    private val runner:ScheduleRunner;

    public def this()
    {
        LiveCount=0 as Long;
        Reg = Region.make((0..(Grid.Grid_XMax-1)),(0..(Grid.Grid_YMax-1)));
        val D=Dist.makeBlockBlock(Reg);
        AgentList=DistArray.make[Agent](D,(p:Point(2))=>null);
        runner = new ScheduleRunner();
    }

    public def initSchedule():void
    {

        val eventsum=new GameOfLifeEventSequence();
        runner.scheduleStopEvent(20,new ScheduleStopFunctor(runner));
        runner.scheduleEvent(0, new ScheduleMethodFunctor(eventsum,1n));
        runner.scheduleEvent(1, 5, new ScheduleMethodFunctor(eventsum,2n));
        runner.scheduleEvent(2, 5, new ScheduleMethodFunctor(eventsum,3n));

    }

    public def getRunner(): ScheduleRunner{
        return runner;
    }
```

2.2.5 Step 5: DataCollection

Data collection can occur in any given event or at any time, but you need to customize the data collection event.

1、Class instantiation

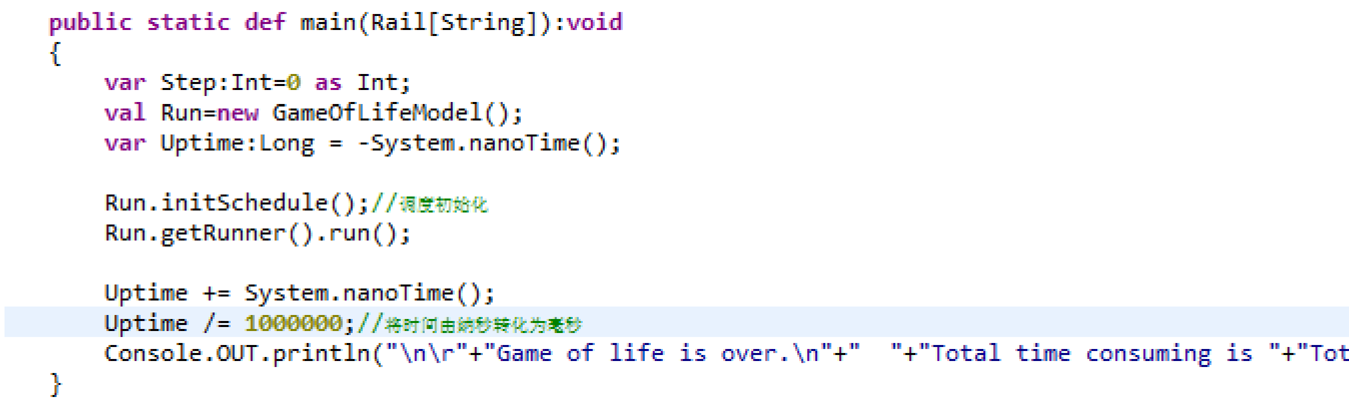
```
public class GameOfLifeEventSequence {
    public var LiveCount:Long;
    public var LiveCountBorn:Long;
    public var LiveCountDead:Long;
    public val AgentList:DistArray[Agent](2);
    public val Reg:Region(2);
    public val DC:DataCollection;
```

2、Class implementation

```
 DC.RecordWrite("  "+"The number of changed Agent (after interactive) is "+ LiveCount +" at Plac
```

2.2.6 Step 6: Run

Count the time in the main function in the Run.x10 file and set the relevant parameters in the Setting.x10 file.。



```
public static def main(Rail[String]):void
{
    var Step:Int=0 as Int;
    val Run=new GameOfLifeModel();
    var Uptime:Long = -System.nanoTime();

    Run.initSchedule();//调度初始化
    Run.getRunner().run();

    Uptime += System.nanoTime();
    Uptime /= 1000000;//将时间由纳秒转化为毫秒
    Console.OUT.println("\n\r"+"Game of life is over.\n"+"  "+"Total time consuming is "+"Tot
}
```