MASS Java Manual

August 31, 2016

1. Intro

This document is written to define the third draft version of the MASS library, a parallel-computing library for Multi-Agent Spatial Simulation. As envisioned from its name, the design is based on multi-agents, each behaving as a simulation entity on a given virtual space. The library is intended to parallelize a program that particularly on multi-entity interaction in physical, biological, social, and strategic domains. The examples include major physics problems (including molecular dynamics, Schrödinger's wave equation, and Fourier's heat equation), neural network, artificial society, and battle games.

2. Table of Contents

3. Programming Model

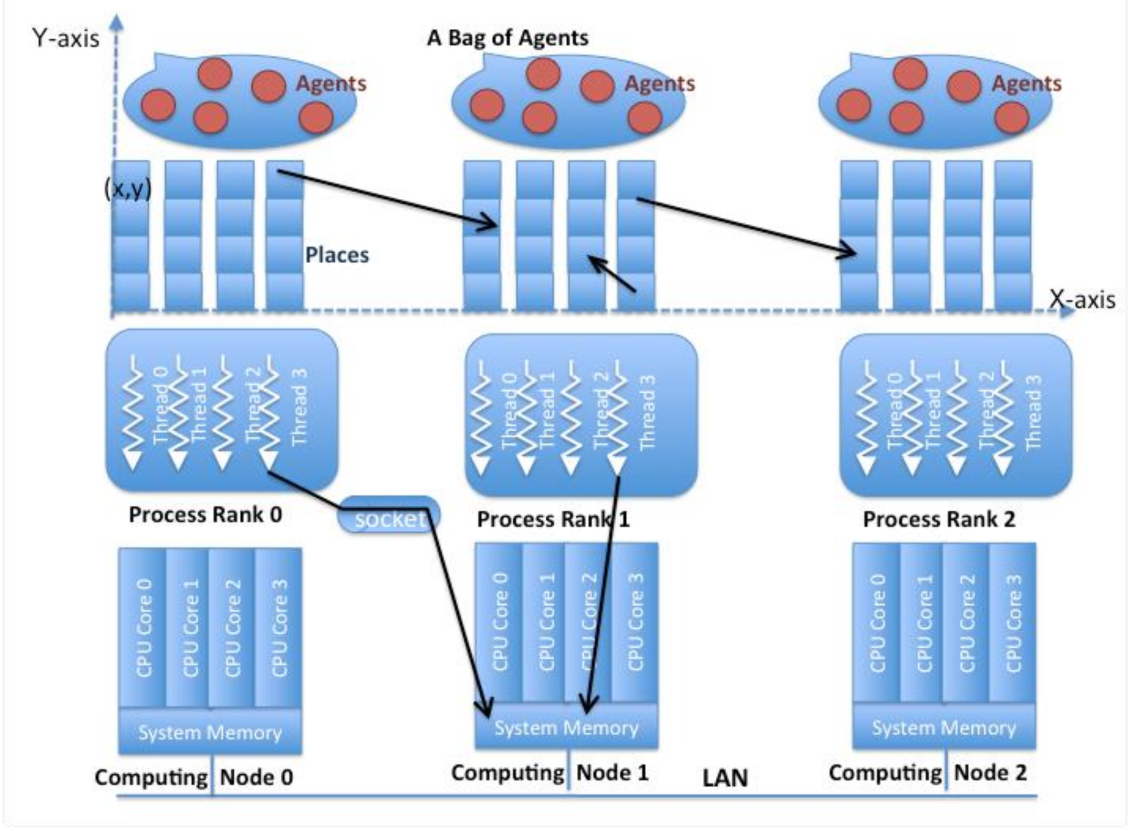
3.1. Components: Places and Agents

Places and **Agents** are keys to the MASS library. **Places** is a matrix of elements that are dynamically allocated over a cluster of computing nodes. Each element, called a **Place**, is pointed to by a set of network-independent matrix indices, and is capable of exchanging information with any other **Place**. On the other hand, **Agents** is a set of execution instances that can reside on a place, migrate to any other **Place** with matrix indices, (thus as duplicating themselves), and interact with other **Agents** as well as multiple **Places**.

An example of **Places** and **Agents** in a battle game could be territories and military units respectively. Some applications may need only either **Places** or **Agents**. For instance, Schrödinger's wave simulation needs only two-dimensional places, each diffusing its wave influence to the neighbors. Molecular dynamics needs only agents, each behaving as a particle since it must collect distance information from all the other particles for computing its next position, velocity, and acceleration.

Parallelization with the MASS library assumes a cluster of multi-core computing nodes as the underlying computing architecture, and thus uses a set of multi-threaded communicating processes that are forked over the cluster and managed under the control of typical message-passing software infrastructure such as sockets. The library spawns the same number of threads as that of CPU cores per node or per process. Those threads take charge of method call and information exchange among places and agents in parallel.

Places are mapped to threads, whereas **Agents** are mapped to processes. Unless a programmer indicates his/her places-partitioning algorithm, the MASS library divides **Places** into smaller stripes in vertical, or in the X-coordinate direction, each of which is then allocated to and executed by a different thread. Contrary to **Places**, **Agents** are grouped into bags, each allocated to a different process where multiple threads keep checking in and out one after another **Agent** from this bag when they are ready to execute a new **Agent**. If **Agents** are associated with a particular **Place**, they are allocated to the same process whose thread takes care of this **Place**.



3.2. Programming Framework

The following code shows a Java programming framework that uses the MASS library to simulate a multi-agent spatial simulation.

```
import edu.uw.bothell.css.dsl.MASS*;

class Application {
```

```
4:
5:          public void static main( String[] args ) {
6:              // get the max simulation time
7:              int maxTime = Integer.parseInt( args[0] );
8:
9:              // start a process at each computing node
10:             MASS.init();
11:
12:             // distribute places and agents over computing nodes
13:             Places territories
14:                 = new Places( 1, "Territory", null, 100, 100 );
15:             Agents troops
16:                 = new Agents( 2, "Troop", null, territories, 40000 );
17:
18:             // start cyclic simulation in parallel
19:             int[] destination = new int[2]; dest[0] = 0; dest[1] = 1;
20:             for ( int time = 0; time < maxTime - 10; time++ ) {
21:                 Object arg = ( Object )( new Integer( time ) );
22:                 territories.callAll( Territory.compute_, arg );
23:                 territories.exchangeAll( 1, Territory.exchange_, dest );
24:                 troops.callAll( Troop.compute_, arg );
25:                 troops.manageAll();
26:             }
27:
28:             // terminate the processes
29:             MASS.finish();
30:         }
31:     }
```

The behavior of the above code is as follows: it synchronizes all processes with `MASS.init()` and has them spawn multiple threads (line 10). The code thereafter maps a matrix of 100 × 100 “Territory” places as well as 4000 “Troop” agents over these processes (lines 13 – 16). Each process falls into a cyclic simulation (lines 20 – 26) where all its threads repeat calling the following three functions in a parallel fashion:

- `compute()` of the “Territory” places to update each place object’s status
- `exchange()` of the “Territory” places to exchange data among place objects
- `compute()` of the “Troop” agents to update each agent’s status

as well as control the “Troop” agents in `manageAll()` so as to move, spawn, terminate, suspend, and resume agents. At the end, all the processes get synchronized together for their termination with `MASS.finish()` (line 29).

In the following sections, we will define the specification of **MASS**, **Place**, **Place**, **Agents**, and **Agent**.

4. MASS

All processes involved in the same MASS library computation must call `MASS.init()` and `MASS.finish()` at the beginning and end of their code respectively, so as to get started and finished together. Upon a `MASS.init()` call, each process, running on a different computing node, spawns the same number of threads as that of its local CPU cores, so that all threads can access places and agents. Upon a `MASS.finish()` call, each process cleans up all its threads as being detached from the **Places** and **Agents** objects.

4.1.1. MASS Method Detail

init

`public static void init()`

Initialize the MASS library (using settings declared in "nodes.xml"). Calling this method effectively begins computation. Use this init when the nodes and other MASS settings are defined in nodes.xml

finish

`public static void finish()`

Finish computation, terminate remote processes, and perform cleanup and disconnection operations. This method should be called when all computational work has been completed.

setNodeFilePath

`public static void setNodeFilePath( String nodeFilePath )`

Set the filename for the cluster node definition file. Use this function if the nodes are defined in an xml file not named "nodes.xml".

Parameters:
`String nodeFilePath` – Directory of the xml file containing the nodes definition.

debugInit

`public void debugInit( int placesHandle, int agentsHandle, int port )`

Starts the debugger and waits to connect to the GUI on the provided port. This function will block the application until the GUI has connected, allowing the GUI and the debugger application to be executed in any order. After a connection is made some initial data about the **Places** and **Agents** are sent to the GUI. See section 7.1.1.

Parameters:
`int placesHandle` – User defined handle of the **Places** object they wish to debug.
`int agentsHandle` – User defined handle of the **Agents** object they wish to debug.
`int port` – Port with which the user wishes to connect the debugger to.

debugUpdate

`public void debugUpdate()`

This method is used to update the information in the debugger. Should be called at the end of the cyclic simulation loop. See section 7.1.1.

setNumThreads

```
public static void setNumThreads( int numThreads )
```

Set the number of threads to spawn on each node.

Parameters:

`int numThreads` – The number of threads to spawn.

5. Places

Places is a distributed matrix whose elements are allocated to different computing nodes. Each element, (termed a **Place**) is addressed by a set of network-independent matrix indices. Once the main method has called `MASS.init()`, it can create as many **Place** objects as needed, using the **Places** constructor. Unless a user supplies an explicit mapping method in their **Place** definition, a **Places** instance is partitioned into smaller stripes in terms of coordinates[0], and is mapped over a given set of computing nodes, (i.e., processes).

5.1. Class Places

Places manages all Place elements within the simulation space. The class instantiates an array of user-defined **Place** objects shared among multiple nodes. Array elements are accessed and processed by multi-processes in parallel.

5.1.1. Places Constructor Detail

Places

```
public Places( int handle, String className, Object argument, int... size )
```

Instantiates a shared array of **Place** objects with **`size`** from the **`className`** class as passing an argument to the **`className`** constructor. This array is associated with a user-given handle that must be unique over machines.

Parameters:

`int handle` – A unique identifier that designates a group of places. Must be unique over all machines.

`String className` – Name of the user-implemented class **Places** are constructed from.

`Object argument` – Arguments for user-defined **Place** constructor.

`int... size` – Size of the matrix the user wants to create. Given in varargs form, so users can create a matrix of how many dimensions they want. For example, a 2-dimensional matrix of size 100 would use `Places( 1, className, argument, 100, 100 )`, while a 3-dimensional matrix of size would use `Places( 1, className, argument, 100, 100, 100 )`.

Places

```
Places( int handle, String className, int boundaryWidth, Object argument,
          int... size )
```

Instantiates a shared array of **Place** objects with **`size`** from the **`className`** class as passing an argument to the **`className`** constructor. This array is associated with a user-given handle that must be unique over machines.

Parameters:

- **`int handle`** – A unique identifier that designates a group of places. Must be unique over all machines.
- **`String className`** – Name of the user-implemented class **Places** are constructed from.
- **`int boundaryWidth`** – The amount of **Place** objects shared between stripes on different nodes. For example, if there are two computing nodes, sharing 100 **Place** objects in the x direction and boundaryWidth is set to 5, node 0 will be responsible for the **Place** objects at x-coordinates 0 – 54 and node 1 will be responsible for the **Place** objects at 45 – 99. This boundary region can easily be exchanged using `Places.exchangeBoundary()`. Default is 0.
- **`Object argument`** – Arguments for user-defined **Place** constructor.
- **`int... size`** – Size of the matrix the user wants to create. Given in varargs form, so users can create a matrix of how many dimensions they want. For example, a 2-dimensional matrix of size 100 would use `Places( 1, className, 1, argument, 100, 100 )`, while a 3-dimensional matrix of size would use `Places( 1, className, 1, argument, 100, 100, 100 )`.

5.1.2. Places Method Detail

getHandle

`public int getHandle()`

Returns the handle associated with this array of **Place** objects.

Returns:
The handle associated with this array.

getSize

`public int[] getSize()`

Returns the size of this multi-dimensional array.

Returns:
The size of the array of **Place** objects.

callAll

`public void callAll( int functionId )`

Calls the user-defined method specified with functionId of all array elements. Done in parallel among multi-processes/threads. To be used when there are no arguments in the user-defined function to be called.

Parameters:

`int functionId` – The Id of the user-created function to be called from the overridden `callMethod()` function that is derived from **Place**. The `callMethod()` function is detailed in section 5.3.

callAll

```
public void callAll( int functionId, Object argument )
```

Calls the user-defined method specified with functionId of all array elements as passing an Object argument to the method. Done in parallel among multi-processes/threads.

Parameters:

`int functionId` – The Id of the user-created function to be called from the overridden `callMethod()` function that is derived from **Place**. The `callMethod()` function is detailed in section 5.3.

`Object argument` – Argument passed to the user-defined function.

callAll

```
public Object[] callAll( int functionId, Object[] arguments )
```

Calls the user-defined method specified with functionId of all array elements as passing arguments[**i**] to element[**i**]'s method, and receives a return value from it into Object[**i**]. Done in parallel among multi- processes/threads. In case of a multi-dimensional array, **i** is considered as the index when the array is flattened to a single dimension.

Parameters:

`int functionId` – The Id of the user-created function to be called from the overridden `callMethod()` function that is derived from **Place**. The `callMethod()` function is detailed in section 5.3.

`Object[] argument` – Arguments passed to the user-defined function.

Returns:

An array of objects from each **Place** objects `callMethod()` return value.

exchangeAll

```
public void exchangeAll( int destinationHandle, int functionId )
```

Calls the user-defined method specified with functionId of all destination cells, each indexed with a different Vector element. Each vector element, say destination[] is an array of integers where destination[i] includes a relative index (or a distance) on the coordinate i from the current caller to the callee cell. The caller cell's outMessage is a continuous set of arguments passed to the callee's method. The caller's inMessages[] stores values returned from all callees. More specifically, inMessages[i] maintains a set of return values from the ith callee.

Parameters:

`int destinationHandle –` The unique identifier defined in the **Places** constructor for the **Places** object that is to be interacted with.

`int functionId –` The id of the user-defined method to call in `callMethod()`.

exchangeBoundary

```
public void exchangeBoundary()
```

Exchanges each node's boundary region with that node's neighbors. If there was no boundary defined in the **Places** constructor, then this will do nothing.

5.2. Class Place

Place is the base class from which a user can derive an application-specific matrix of **Place** objects. An actual matrix instance is created and maintain within a **Places** class, so that the user can obtain parallelizing benefits from **Places** `callAll()` and `exchangeAll()` methods that invoke a given method of each matrix element and exchange data between each element and others.

5.2.1. Method Detail

callMethod

```
public Object callMethod( int functionId, Object arguments )
```

Is called from `Places.callAll()` and `Places.exchangeAll()`. Invokes the user-created function pointed to by functionId. An application should override `callMethod()` so as to direct **Places** to invoke an application-specific method.

Parameters:

`int functionId –` Id of the user-defined function to call.

`Object[] arguments –` Arguments passed to the user-defined function.

Returns:

Return value of the user-defined functions.

getAgents

```
public synchronized Set<Agent> getAgents()
```

Returns a set containing the **Agents** that are executing on this **Place**.

Returns:

A set containing the **Agents** that are executing on this **Place.**

getNumAgents

```
public int getNumAgents()
```

Returns the number of Agents that are executing on this Place.

Returns:

The number of Agents that are executing on this Place.

getInMessages

`public Object[] getInMessages()`

Returns the values from this Places neighbors defined in `setNeighbors()` after an `exchangeAll()` call.

Returns:

The values from this Places neighbors defined in `setNeighbors()` after an `exchangeAll()` call.

setNeighbors

`public void setNeighbors( Vector<int[]> neighbors )`

Defined the Places relative to this Place that this Place is to look to when exchanging information between Places in exchangeAll().

Parameters:

`Vector<int[]> neighbors –` The coordinates of Places that are defined to be neighbors if this Place.

getDebugData

`public Number getDebugData()`

Debugger method that allows the user return the data for this **Place** so they can view it in the debugger. To be implemented by the user in their **Place** class. See section 7.1.2.

Returns:

The value the user wants to return for this **Place** as a Number, a Java class Double, Integer, Byte, etc., are derived from.

setDebugData

`public void setDebugData( Number data )`

Debugger method that allows the user to inject their own values for this **Place** in the debugger. To be implemented by the user in their **Place** class. See section 7.1.2.

Parameters:

`Number data –` The value to be added to this **Place.**

5.3. callMethod

Since method names are user-given, it is quite natural to invoke each array element's method through Java reflection, which is intolerably slow for parallel computing. Thus, when deriving from **Place,** the user should override the `callMethod()` function to run user made functions. The `callMethod()` function is a user-provided framework that assists the MASS library in choosing a method to call. The `callMethod()` function should include a switch statement where each function, is identified by a user-defined integer value. This is shown in the code sample below.

The `callMethod()` is called by the `callAll()` and `exchangeAll()` functions in **Places** to allow all **Place** objects functions to be called in parallel.

```
public class Wave2D extends Place {
   // constants: each array element's methods are identified by an integer
   // rather than its name.
   public static final int init_ = 0;
   public static final int computeNewWave_ = 1;
   public static final int exchangeWave_ = 2;
   public static final int collectWave_ = 3;
   public static final int startGraphics_ = 4;
   public static final int writeToGraphics_ = 5;
   public static final int finishGraphics_ = 6;

   // automatically called from callAll and exchangeAll.
   // args may be null depending on a calling method.
   public static Object callMethod( int funcId, Object args ) {
      switch( funcId ) {
         case init: return init( args );
         case computeNewWave_: return computeNewWave( args );
         case exchangeWave_: return exchangeWave( args );
         case storeWave_: return exchangeWave( args );
         case startGraphics_: return startGraphics( args );
         case writeToGraphics_: return writeToGraphics( args );
         case finishGraphics_: return finishGraphics( args );
      }
      return null;
   }

   public Object init( Object args ) {
      ...;
   }
   Public Object computeNewWave( Object args ) {
      ...;
   }
}
```

5.4. Example 1: Wave2D

The following Wave2D class is a two-dimensional matrix that simulates Schrödinger's wave diffusion. Each "Wave2D" matrix element maintains the six instance variables listed below:

wave[2]:	The current wave height at each matrix element
wave[1]:	The previous wave height
wave[0]:	Even the one more previous wave height
neighbor[0]:	Through to neighbor[3]: the wave height of north/east/south/west neighbors
time:	The current simulation time
interval:	Time interval to display the ongoing simulation results

In this Wave2D class, callMethod() maps each functionId to the corresponding function (lines 43 – 52); init() initializes instance variables of each place or cell (lines 68 – 76); computeWave( ) simulates wave diffusion per time unit (lines 78 – 120); exchangeWave() receives wave heights from all the four neighbors (lines 122 – 125); collectWave() collects wave heights from all places into place[0][0] that then displays them in graphics (lines 127 – 129).

```
1:     import edu.uw.bothell.css.dsl.MASS.*;
2:     // Library for Multi-Agent Spatial Simulation
3:     // also uses key events so we need this
4:
5:     public class Wave2DMass {
6:        public static void main( String[] args ) throws Exception {
        MASS.init( );

        // create a Wave2D array
        Places wave2D = new Places( 1, Area.class.getName(), null, 100, 100 );
        wave2D.callAll( Area.init_ );

        // now go into a cyclic simulation
        for ( int time = 0; time < 500; time++ ) {
            wave2D.callAll( Area.computeWave_, new Integer( time ) );
            wave2D.exchangeAll( 1, Area.exchangeWave_ );
        }

        MASS.finish( );
    }
}

public class Area extends Place {
    // constants
    public static final int init_ = 0;
    public static final int computeWave_ = 1;
    public static final int exchangeWave_ = 2;
    public static final int collectWave_ = 3;

    // wave height at each cell
    // wave[0]: current, wave[1]: previous, wave[2]: one more previous height
    double[] wave = new double[3];

    int time = 0;

    // wave height from four neighbors: north, east, south, and west
    private final int north = 0, east = 1, south = 2, west = 3;
    double[] neighbors = new double[4];

    // simulation constants
    private final double c  = 1.0; // wave speed
    private final double dt = 0.1; // time quantum
    private final double dd = 2.0; // change in system

    // the array size and my index in (x, y) coordinates
    private int sizeX, sizeY;
    private int myX, myY;

    public Area( Object args ) {
        Vector<int[]> placeNeighbors = new Vector<int[]>();
        placeNeighbors.add( new int[] { 0, -1 } );
        placeNeighbors.add( new int[] { 1, 0 } );
        placeNeighbors.add( new int[] { 0, 1 } );
        placeNeighbors.add( new int[] { -1, 0 } );
        setNeighbors( placeNeighbors );
    }

    public Object callMethod( int funcId, Object args ) {
        switch( funcId ) {
            case init_: return init( args );
            case computeWave_: return computeWave( args );
            case exchangeWave_: return ( Object )exchangeWave( args );
            case collectWave_: return ( Object )collectWave( args );
        }
        return null;
    }

    public Object init( Object args ) {
        sizeX = getSize()[0]; sizeY = getSize()[1]; // size  is the base data members
        myX = getIndex()[0];  myY = getIndex()[1];  // index is the base data members

        // reset the neighboring area information.
        neighbors[north] = neighbors[east] = neighbors[south] = neighbors[west] = 0.0;

        return null;
    }

    public Object computeWave( Object arg_time ) {
        // retrieve the current simulation time
        time = ( ( Integer )arg_time ).intValue( );

        // move the previous return values to my neighbors[].
        if ( getInMessages() != null ) {
            for ( int i = 0; i < 4; i++ ) {
                if (getInMessages()[i] != null){
                    neighbors[i] = ( ( Double )getInMessages()[i] );
                }
            }
        }

        if ( myX == 0 || myX == sizeX - 1 || myY == 0 || myY == sizeY ) {
            // this cell is on the edge of the Wave2D matrix
            if ( time == 0 ) wave[0] = 0.0; //current
            if ( time == 1 ) wave[1] = 0.0; //previous
            else if ( time >= 2 ) wave[2] = 0.0; //previous2
        } else {
            // this cell is not on the edge
            if ( time == 0 ) {
                // create an initial high tide in the central square area
                wave[0] = ( sizeX * 0.4 <= myX && myX <= sizeX * 0.6 &&
                        sizeY * 0.4 <= myY && myY <= sizeY * 0.6 ) ? 20.0 : 0.0;
                        //start w/ wave[0]
                wave[1] = wave[2] = 0.0; // init wave[1] and wave[2] as 0.0
            } else if ( time == 1 ) {
                // simulation at time 1
                wave[1] = wave[0] + c * c / 2.0 * dt * dt / ( dd * dd ) *
                        ( neighbors[north] + neighbors[east] + neighbors[south] +
                        neighbors[west] - 4.0 * wave[0] ); //wave[1] based on wave[0]
            } else if ( time >= 2 ) {
                // simulation at time 2 and onwards
                wave[2] = 2.0 * wave[1] - wave[0] + c * c * dt * dt / ( dd * dd ) *
                        ( neighbors[north] + neighbors[east] + neighbors[south]
                        + neighbors[west] - 4.0 * wave[1] );
                        //wave two based on wave[1] and wave[0]
                wave[0] = wave[1]; wave[1] = wave[2];
                        //shift wave[] measurements, prepare for a new wave[2]
            }
        }
        return null;
    }

    public Double exchangeWave( Object args ) {
        return new Double( ( ( time == 0 ) ? wave[0] : wave[1] ) );
        //wave one used after start
    }

    public Double collectWave( Object args ) {
        return new Double( wave[2] );
    }
}
```

6. Agents

Agents are a set of execution instances made up of user-defined **Agent** objects. Each is capable of residing on a place, migrating to any other **Place** objects with matrix indices, and interacting with other agents as well as multiple **Places**.

6.1. Class Agents

Once the main method has called `MASS.init()`, it can create as many **Agent** objects as needed, using the **Agents** constructor. Unless a user supplies an explicit mapping method in their **Agent** definition (see 6.2 Class Agent), **Agents** distribute instances of a given **Agent** class uniformly over different computing nodes.

6.1.1. Agents Constructor Detail

Agents

```
public Agents( int handle, String className, Object argument, Places places,
            int initPopulation )
```

Instantiates a set of **Agent** objects from the `className` class, passes the `argument` object to their constructor, associates them with a given **Place** matrix, and distributes them over these places, based the `Agents.map()` method that is defined within the Agent class. If a user does not overload it by themselves, `Agents.map()` uniformly distributes an `initPopulation` number of **Agent** objects. If a user-provided `Agents.map()` method is used, it must return the number of **Agent** objects spawned at each **Place** regardless of the `initPopulation` parameter. Each set of **Agent** objects is associated with a user-given `handle` that must be unique over all machines.

Parameters:

- `int handle` – A unique identifier that designates a group of **Agent** objects. Must be unique over all machines.
- `String className` – The name of the user-defined class to instantiate
- `Object argument` – The argument to pass to each **Agent** as it is being instantiated
- `Places places` – The **Places** instance that will contain the **Agents**
- `int initPopulation` – The number of **Agent** Objects to create.

6.1.2. Agents Method Detail

getHandle

```
public int getHandle()
```

Returns the handle associated with this **Agent** set.

Returns:
Unique handle for this **Agent** set.

nAgents

```
public int nAgents()
```

Returns the total number of **Agent** objects in this set.

Returns:
Total number of **Agent** objects in this set.

callAll

`public void callAll( int functionId )`

Calls the user-defined method specified with functionId of all **Agent** objects in this this collection. Done in parallel among multi-processes/threads. To be used when there are no arguments in the user-defined function to be called.

Parameters:

`int functionId` – The Id of the user-created function to be called from the overridden `callMethod()` function that is derived from **Agent**. This `callMethod()` function is similar to the one defined in **Places** and detailed in section 5.3.

callAll

`public void callAll( int functionId, Object argument )`

Calls the method specified with functionId of all agents as passing an Object argument to the method. Done in parallel among multi-processes/threads.

Parameters:

`int functionId` – The Id of the user-created function to be called from the overridden `callMethod()` function that is derived from **Agent**. The `callMethod()` function is similar to the one defined in **Places** and detailed in section 5.3.

`Object argument` – Argument passed to the user-defined function.

manageAll

`public void manageAll()`

Updates each agent's status, based on each of its latest `migrate()`, `spawn()`, and `kill()` calls. These methods are defined in the Agent base class and may be invoked from other functions through callAll. Done in parallel among multi-processes/threads.

The following example of code fragment creates a size x size matrix over multiple processes (line 10), distributes a "RandomWalk" agent every four places of this matrix (lines 11 – 12), and simulates random walk of these agents over the matrix using many threads (lines 13 – 17).

```
1:      import MASS;
2:
3:      class RandomWalkDriver  {
4:         public void static main( String[] args ) {
5:            MASS.init();
6:
7:            int size = Integer.parseInt( args[0] );
8:            int maxTime = Integer.parseInt( args[1] );
9:
10:           Places matrix = new Place( 1, "Matrix", null, size, size );
11:           Agents walkers =
12:                  new Agents( 2, "RandomWalk", null, matrix, size * size / 4 );
13:           for ( int time = 0; time < maxTime; time++ ) {
14:              walkers.callAll( RandomWalk.newLocation );
15:
```

```
                walkers.manageAll( );
            }

            MASS.finalize();
        }
    }
```

6.2. Class Agent

Agent is the base class from which a user can derive an application-specific agent that migrates to another **Place**, forks their copies, and terminate themselves.

6.2.1. Agent Method Detail

getPlace

`public Place getPlace()`

Returns the current place where this agent resides.

Returns:

The current user-defined **Place** where this **Agent** currently resides.

map

`public int map( int initPopulation, int[] size, int[] index )`

Returns the number of **Agent** objects to initially instantiate on a **Place** indexed with **`index`**. The **`initPopulation`** parameter indicates the number of **Agent** objects to create over the entire application. The **`size`** argument defines the size of the **Place** matrix to which a given **Agent** class belongs. The system-provided (thus default) `map()` method distributes **Agent** objects over places uniformly as in:

`initPopulation / size.length`

The `map()` method may be overloaded by an application-specific method. A user-provided `map()` method may ignore **`initPopulation`** when creating agents.

Parameters:

`int initPopulation` – Number of **Agent** objects to create over the entire application.
`int[] size` – Size of the **Place** matrix to which a given **Agent** class belongs.
`int[] index` – Denotes which **Place** to initially instantiate the **Agent** objects on.

Returns:

The number of **Agent** objects initially instantiated on the **Place** indexed with **`index`**.

migrate

`public boolean migrate( int... index )`

Initiates an agent migration to a new **Place** upon a next call to `Agents.manageAll()`. More specifically, `migrate()` updates the calling agent's index[].

Parameters:
`int... index` – Coordinates of the new **Place** this **Agent** will migrate to.

Returns:
True if migration is successful, false otherwise.

spawn

```
public void spawn( int numAgents, Object[] arguments )
```

Spawns a `numAgents` amount of new **Agent** objects, as passing arguments[i] to the i-th new agent upon a next call to `Agents.manageAll()`.

Parameters:
`int numAgents` – Amount of **Agent** Objects to create.
`Object[] arguments` – An array of parameters to give to the new **Agent** objects.

kill

```
public void kill()
```

Terminates the calling **Agent** upon a next call to `Agents.manageAll()`.

callMethod

```
public Object callMethod( int functionId, Object arguments )
```

Is called from `Agents.callAll()`. Invokes the user-created function pointed to by functionId. An application should override `callMethod()` so as to direct **Agents** to invoke an application-specific method.

Parameters:
`int functionId` – Id of the user-defined function to call.
`Object[] arguments` – Arguments passed to the user-defined function.

Returns:
Return value of the user-defined functions.

getDebugData

```
Public Number getDebugData()
```

Debugger method that allows the user return the data for this **Agent** so they can view it in the debugger. To be implemented by the user in their **Agent** class. See section 7.1.2.

Returns:
The value the user wants to return for this **Agent** as a Number, a Java class Double, Integer, Byte, etc., are derived from.

setDebugData

`public void setDebugData( Number data )`

Debugger method that allows the user to inject their own values for this **Agent** in the debugger. To be implemented by the user in their **Agent** class. See section 7.1.2.

Parameters:
  `Number data –` The value to be added to this **Agent.**

6.3. Example 2: RandomWalk

The following RandomWalk application simulates the walking of **Agent** objects, each searching for and moving to the least crowded **Place** over a two-dimensional matrix. Starting from `RandomWalk.main()`, the program instantiates a two-dimensional "Land" array, and populates **Agent** objects over a small square in the middle of this land (lines 9 – 13), and finally goes into a cyclic simulation (lines 16 – 24) where each cell exchanges the number of **Agent** objects with its four neighbors and each **Agent** migrates to a neighbor with the least **Agent** objects. This cyclic simulation is performed in parallel with multiple processes, each with multiple threads. The class "Land" defines this two-dimensional simulation space (lines 32 – 91) and the class "Nomad" describes each **Agent**'s behavior (lines 94 – 158).

```
1:      import edu.uw.bothell.css.dsl.MASS.*;     // Library for Multi-Agent Spatial Simulation
2:      import java.util.*;
3:
4:      public class RandomWalk {
5:          public static void main( String[] args ) throws Exception {
6:              MASS.init( );
7:
8:              // create a Land array.
9:              Places land = new Places( 1, "Land", null, 100, 100 );
10:             land.callAll( Land.init_ );
11:
12:             // populate Nomad agents on the land.
13:             Agents nomad = new Agents( 2, "Nomad", null, land, 1000 );
14:
15:             // now go into a cyclic simulation
16:             for ( int time = 0; time < 500; time++ ) {
17:                 // exchange #agents with four neighbors
18:                 land.exchangeAll( 1, Land.exchange_ );
19:                 land.callAll( Land.update_ );
20:
21:                 // move agents to a neighbor with the least population
22:                 nomad.callAll( Nomad.decideNewPosition_ );
23:                 nomad.manageAll( );
24:             }
25:
26:             // finish MASS
27:             MASS.finish( );
28:         }
29:     }
30:
31:     // Land Array
32:     public class Land extends Place {
33:         // function identifiers
34:         public static final int exchange_ = 0;
35:         public static final int update_ = 1;
36:         public static final int init_ = 2;
37:         public static final int collectAgents_ = 3;
38:
        // the array size and my index in (x, y) coordinates
        private int sizeX, sizeY;
        private int myX, myY;

        // wave height from four neighbors: north, east, south, and west
        private final int north = 0, east = 1, south = 2, west = 3;
        int[][] neighbors = new int[2][2]; // my four neighbors' #agents

        // constructor
        public Land( Object object ) {
            Vector<int[]> placeNeighbors = new Vector<int[]>();
            placeNeighbors.add( { 0, -1 } );
            placeNeighbors.add( { 1, 0 } );
            placeNeighbors.add( { 0, 1 } );
            placeNeighbors.add( { -1, 0 } );
            setNeighbors( placeNeighbors );
        }

        public double collectAgents( Object args ) {
            //if(agents.size() > 0 ) System.err.println(agents.size());
            return ( getAgents().size() );
        }

        public Object callMethod( int funcId, Object args ) {
            switch ( funcId ) {
                case exchange_: return exchange( args );
                case update_: return update( args );
                case init_ : return init(args);
                case collectAgents_ : return ( Object )collectAgents(args);
            }
            return null;
        }

        public Object exchange( Object args ) {
            return new Integer( getAgents().size( ) );
        }

        public Object update( Object args ) {
            int index = 0;
            for ( int x = 0; x < 2; x++ )
                for ( int y = 0; y < 2; y++ )
                    neighbors[x][y] = ( getInMessages()[index] == null ) ?
                            Integer.MAX_VALUE : ( Integer )getInMessages()[index];
            return null;
        }

        public Object init( Object args ) {
            sizeX = getSize()[0]; sizeY = getSize()[1]; // size  is the base data members
            myX = getIndex()[0];  myY = getIndex()[1];  // index is the base data members

            return null;
        }
    }

    // Nomad Agents
    public class Nomad extends Agent {
        // function identifiers
        public static final int decideNewPosition_ = 0;

        // define the four neighbors of each cell
        private Vector<int[]> neighbors = new Vector<int[]>( );

        public Nomad( ) {
            super(  );
        }

        public Nomad( Object object ) {
```

```
        super( );

        int[] north = { 0, -1 }; neighbors.add( north );
        int[] east  = { 1,  0 }; neighbors.add( east );
        int[] south = { 0,  1 }; neighbors.add( south );
        int[] west  = { -1, 0 }; neighbors.add( west );
    }

    public int map( int maxAgents, int[] size, int[] coordinates ) {
        int sizeX = size[0], sizeY = size[1];
        int populationPerCell = (int)Math.ceil( maxAgents / ( sizeX * sizeY * 0.6 ) );
        int currX = coordinates[0], currY = coordinates[1];
        if ( sizeX * 0.4 < currX && currX < sizeX * 0.6 &&
               sizeY * 0.4 < currY && currY < sizeY * 0.6 ) {
            return populationPerCell;
        } else
            return 0;
    }

    public Object callMethod( int funcId, Object args ) {
        switch ( funcId ) {
            case decideNewPosition_: return decideNewPosition( args );
        }
        return null;
    }

    public Object decideNewPosition( Object args ) {
        int newX = 0;                 // a new destination's X-coordinate
        int newY = 0;                 // a new destination's Y-coordinate
        int min = 1;  // a new destination's # agents

        int currX = getPlace().getIndex()[0], currY = getPlace().getIndex()[1];
        int sizeX = getPlace().getSize()[0], sizeY = getPlace().getSize()[1];

        Random generator = new Random();
        boolean candidatePicked = false;
        int next = 0;
        while(!candidatePicked) {
            next = generator.nextInt(4);
            int[] neighbor = neighbors.get(next);
            newX = currX + neighbor[0]; newY = currY + neighbor[1];
            if ( newY < 0 ) continue; // no north
            if ( newX >= sizeX ) continue; // no east
            if ( newY >= sizeY ) continue; // no south
            if ( newX < 0 ) continue; // no west

            candidatePicked = true;
        }
        // let's migrate
        migrate( newX, newY );
        return null;
    }
}
```

7. Debugger

A debugger and a GUI for that debugger was created to help visualize the values of **Places** and **Agents** as MASS is running. This debugger can also allow the user to pause their application and edit the values of **Places** and **Agents**. To make use of the debugger, a few lines of code has to be added to the user application, and the GUI has to be launched and connected to the MASS debugger.

7.1. User Application Changes

7.1.1. Main Functions

To enable the debugger, the user needs to add two functions to their applications main function.

The `MASS.debugInit( )` function on line 9 will start up the debugger and wait to connect with the GUI on the port passed as the third argument. This function must be declared before the cyclic simulation loop to give the GUI and the debugger a chance to communicate and transfer initial data.

The `MASS.debugUpdate( )` function on line 16 sends the updated **Place** and **Agent** information to the GUI as well as make the changes to the data in the current **Places** and **Agent** that the user specified in the GUI.

```
1:      public static void main( String[] args ) throws Exception {
2:
3:          MASS.init( );
4:
5:          // create a Wave2D array
6:          Places wave2D = new Places( 1, "Area", null, 100, 100 );
7:          wave2D.callAll( Area.init_ );
8:
9:          MASS.debugInit( 1, 0, 9799 );
10:
11:         // now go into a cyclic simulation
12:         for ( int time = 0; time < 500; time++ ) {
13:             wave2D.callAll( Area.computeWave_, new Integer( time ) );
14:             wave2D.exchangeAll( 1, Area.exchangeWave_ );
15:
16:             MASS.debugUpdate();
17:         }
18:
19:         MASS.finish( );
20:     }
```

7.1.2. Place and Agent Functions

To expose the data that is wanted to be shown through the debugger, the user has to implement functions in their classes that are derived from **Place** and **Agent**.

Both of the **Place** and **Agent** classes define a `getDebugData()` function that returns a Number, which is a java abstract class in which Double, Float, Long, Integer, Byte, etc., all extend from. The debugger uses this function to get the values of the **Places** and **Agents** that the user wishes to debug. An example of this function is shown below as it would be used in the Wave2D application from section 5.4. It returns the current value of the wave at any time.

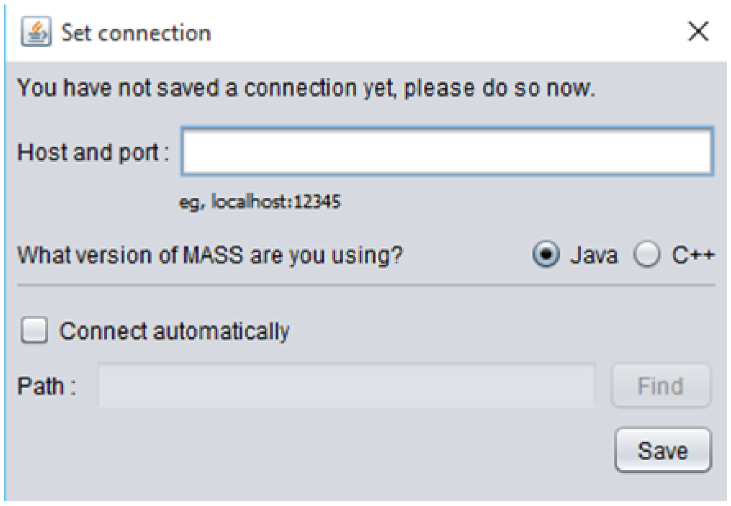
```
public Number getDebugData() {
    Number w = null;
    if ( time == 0 ) w = new Double( wave[0] );
    else if ( time == 1 ) w = new Double( wave[1] );
    else w = new Double( wave[2] );
    return w;
}
```

Place and **Agent** also define a `setDebugData()` function that allows the user to inject new values into existing **Places** and **Agents**. If these functions are not implemented, then the option to inject the new values won't show up on the GUI. This function takes in a Number as an argument, and it is up to the user to cast it to the correct data type. An example from Wave2D is shown below.

```
public void setDebugData( Number argument ){
    Double d = (Double)argument;
    wave[1] = d.doubleValue();
}
```

7.2. Debugger GUI

The debugger is designed to be launched separately from the MASS application and connect to the application over a network. Upon startup, for the first time, the debugger asks the user for the host address and the port the MASS application is listening on. It also asks whether the user is using the Java or C++ version of MASS. When the user enters this information and clicks save, this information is saved for future uses of the debugger, and the main debugger GUI is opened. This first save form is pictured in figure 2.



Set connection

You have not saved a connection yet, please do so now.

Host and port :

eg, localhost:12345

What version of MASS are you using? [✓ Java] C++

Connect automatically

Path :

Figure 2: UI element that opens when the user starts the debugger for the first time.

The bottom right of the GUI features an element that tells the status of the connection with the MASS application. When the application is first started, it will read “Not Connected,” until the user clicks the connect button on the top right of the GUI. The connection status will read “Connected” if there are not problems connecting. Errors connecting are also reported here. The “Not Connected” and “Connected” status are shown in figure 3.

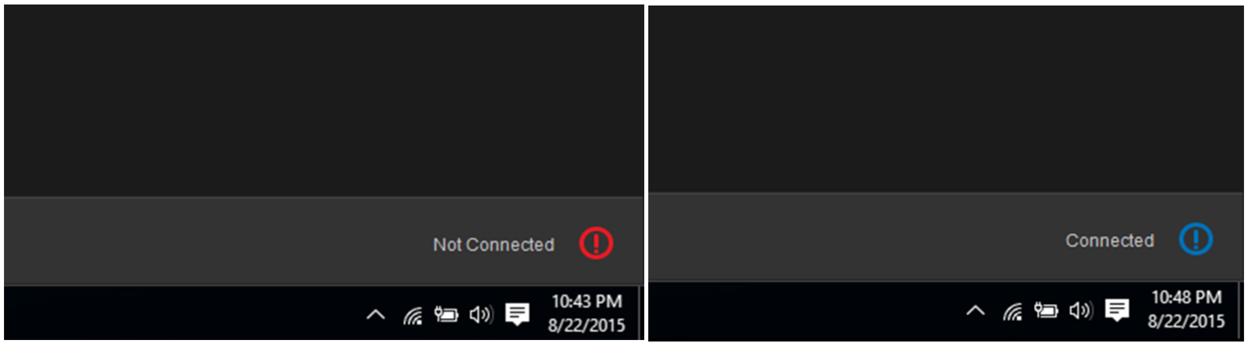


Figure 3: Connection statuses for “Not Connected” (Left) and “Connected” (Right).

The bottom left of the GUI features an element that allows the user to pause their MASS application and continue it. Clicking the pause symbol will pause the application, clicking the play symbol will resume the application until it finishes or is paused, and the skip symbol will advance the application one iteration if it is paused. This element is shown in figure 4.

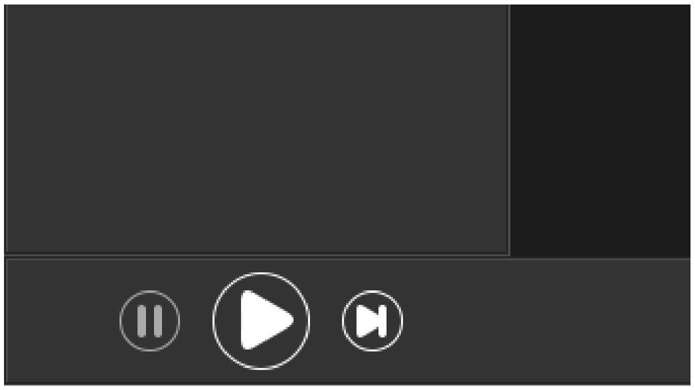
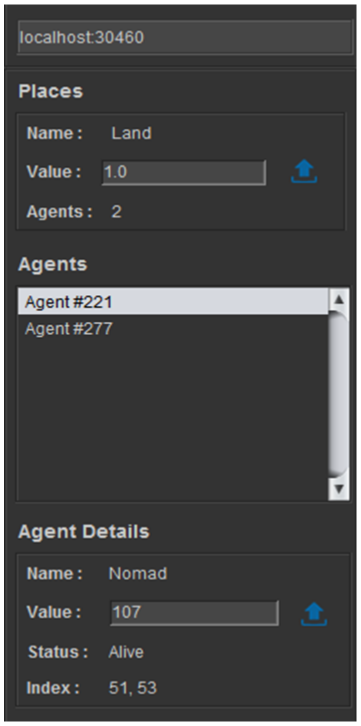


Figure 4: UI element for pausing, resuming, stepping forward once.

The UI element on the left shows the **Place** the user has most recently clicked on and the value associated with that **Place** the was returned from the `getDebugData()` function. It also shows the **Agents** currently residing on that **Place**, and the values associated with that **Agent** that are returned from the `getDebugData()` function. The blue arrow symbols appear if the `setDebugData()` functions have been overridden in the user **Places** and **Agents**. The user can put in their own values and click the arrow to set their new values into their application. This element is shown in figure 5.



localhost:30460

Places

Name : Land

Value : 1.0

Agents : 2

Agents

Agent #221

Agent #277

Agent Details

Name : Nomad

Value : 107

Status : Alive

Index : 51, 53

Figure 5: UI element showing the data in **Places** and **Agents**.