M.A.S.L.
(Multi-Agent Simulation Language)

COMSW4115 Programming Languages & Translators

Project Proposal

Jiatian Li	Wei Wang	Chong Zhang	Dale Zhao
jl3930	ww2315	cz2276	dz2242

Overview and Motivation

The Agent-Based Model (ABM) describes a system where the interactions between autonomous agents (individuals) are simulated and the global patterns and effects of such interactions as a whole can be observed and assessed. ABMs have already been employed in various applications including analyzing traffic congestion, modeling social networks, predicting species populations and distributions, etc.

To facilitate building ABMs without having to start from scratch or engaging complex domain toolkits, Multi-Agent Simulation Language, or MASL, is proposed. MASL provides concise syntax and other convenient facilities for users to better focus on describing and solving the problems at hand.

Objectives

Before discussing the objectives we expect to achieve with MASL, here is a summary of essential elements within a single agent in an ABM.

- Properties representing its state
- Actions that may change its properties
- Connections to some other agents in the environment, which may vary over time
- Heuristic rules that trigger certain actions based on the states of other agents connected to it

Given this, some most important objectives of designing MASL are listed below.

- Provide general programming constructs for specifying the states, actions and decision making process of agents, so that the ABMs implemented in MASL will not be limited to certain domains.
- Provide facilities (including succinct syntax and optimized underlying data storage) to define and access connections among agents efficiently.
- Let the users focus on agents and their connections, and handle the details of running simulations behind the scene. Meanwhile, the infrastructure for running simulations should still be configurable via MASL itself.

Features

To achieve the objectives listed in the previous section, the following features are proposed for MASL.

Prototype-Based Programming

It is natural to develop an ABM with the object-oriented paradigm in mind, since every agent can be represented as an object with a set of properties to maintain their internal state, as well as the capabilities to perceive the environment, make decisions and perform actions. In MASL, we take the prototype-based approach to provide support for object-oriented programming, similar to that in JavaScript. Compared with using classes, prototyping enables greater flexibility for adding features to individual agents.

Built-in State Automata

In many ABMs, agents may not only change their properties but also alter or adapt their actions based on their experience in the environment to simulate conditional behaviors or adaptation. This may be achieved via MASL's built-in support for state automata. In MASL, every object is a state automaton with user-defined states, each containing code specifying conditional actions. An object can also transfer from one state to another via such code.

Flexible and Efficient Container Access

There may be a large number of agents and connections in an ABM. Moreover, agents may be spawned or destroyed in the simulation process, and connections may also evolve over time. Thus, flexible and efficient containers are required to store these elements for frequent access, and MASL has a bunch of support for achieving this.

Clear Boundary for Core Language

The core part of MASL only concerns the logic of ABMs. However, MASL also comes with a minimal standard library that provides I/O functionality (such as data visualization and file access) and other utilities (such as basic mathematic library). These features may make it handy to explore many more possibilities with MASL.

Configurable Execution Environment

One big advantage MASL has is that it frees user from implementing the complex process of multi-agent simulation themselves. The users are mainly concerned with the functionality of agents. Meanwhile, the execution environment of the simulation is configurable by within MASL to suit different applications.

Sample and Syntax

To give a taste of MASL, in this section we will provide a sample program written in MASL and highlight some important features of MASL.

Sample: Conway's Game of Life

The cellular automaton is an important form of ABM, and Conway's Game of Life is a classic application of cellular automata (please refer to http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life for a brief introduction). Below is a minimal MASL implementation with comments.

```
/*
 *
 * Conway's Game of Life implemented using MASL.
 * Please refer to http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
 * for a brief description about the rules of this game.
 *
*/

// Define the cell in the game as an object.

object Cell {

	// The initial state of the cell.

	state Init {

		object[] neighbors = container[{
			(x - 1, y - 1), (x - 1, y), (x - 1, y + 1),
			(x, y - 1), (x, y + 1),
			(x + 1, y - 1), (x + 1, y), (x + 1, y + 1)
		}];
	}

	// If the cell is alive, it will only live on with 2 or 3 live neighbors.

	state Live {

		if(countLiveNeighbors() < 2 || countLiveNeighbors() > 3) ->Dead
	}

	// If the cell is dead, only when exactly 3 live neighbors will render it alive.

	state Dead {

		if(countLiveNeighbors() == 3) ->Live
	}

	// The coordinate of the cell.

	int x, y;

	// A routine counting the number of live neighbors.

	int countLiveNeighbors() {

		return neighbors.count(boolean (object cell) {return cell@Live;});
	}
}

// Initialize the container for all the cells as a 2D array.

container = Array {d = 2, size = (60, 80)};

// fill the container with cells in random states.

for((int, int) (x, y) : {(0, 0)..(59, 79)}) {

	object cell;
	if(math.randomInt() % 2 == 0) cell = Cell {.x = x, .y = y}->Live;
	else cell = Cell {.x = x, .y = y}->Dead;
	container[(x, y)] = cell;
}
```

```
// Set the time step (in milliseconds) and start the simulation.

timeStep = 30;
start();
stop(10000);
```

I/O functionality is not involved in this simple program. In fact, logic models and I/O such as visualization can be clearly separated while easily cooperated using MASL. The details will be covered in future works.

There are three aspects worth mentioning regarding the code above.

Objects with States

In MASL, every object has one or several states, and is in exactly one of the states at a given time, known as its current state. During each step within the simulation loop, the code in the current state gets executed automatically. An object can enter another state via this code, too.

The existence of states introduces state wide scope. Items defined within a state are invisible outside this state.

To make accessing states easier, we introduced two operators: `->` and `@`. `->` transfers an object to a given state, and `@` is used check if an object is in a given state. `->` can only appear within a state or when initializing an object.

```
cell = Cell {x = x, y = y}->Live;
```

Like in this statement, an object is initialized by setting some of its member variables and optionally specifying the state it shall jump in. Every object is in a special state named `Init` upon initialization. Once initialized, it may go to either the state given in the initialization statement or some state specified within the `Init` state. When both present, the latter dominates. If neither presents, the object will enter the `Null` state and will not be updated in the simulation loop.

Simulation as the Global Object

You may notice that when accessing `container`, `timeStep` and `start()`, we did not specify who they belong to. In fact they are all members of a global object that is not explicitly referred to. This object is known as the `Simulation` object. Actually, a single MASL program usually corresponds to an ABM simulation.

We can control or fine tune various aspects of an ABM simulation process via its members, such as specifying the type of container to be used for storing the agents (using `container`), setting the time elapse between consequent updates (using `timeStep`) and starting or stopping the simulation (using `start()` and `stop()`).

Since the Simulation object can be implicitly referred to, for simple tasks, code in procedural programming style can be written on the topmost level (outside any) of a MASL source file directly. Such code is actually contained in the `Init` state of `Simulation` object.

Support for Various Containers

MASL provides native support for several types of containers commonly used in ABMs.

In the Game of Life program, the cells are best stored in a rectangular 2-D array indexed by 2 integers. Actually rectangular n-D arrays are widely used in cellular automata programs. So we provide convenient syntax for accessing n-D arrays, as seen in

```
object[] neighbors = container[{
      (x - 1, y - 1), (x - 1, y), (x - 1, y + 1),
      (x, y - 1), (x, y + 1),
      (x + 1, y - 1), (x + 1, y), (x + 1, y + 1)
}];
```

where multiple elements stored in an n-D array are retrieved as a 1-D array at a time, and

```
(int, int) (x, y) : {(0, 0)..(59, 79)}
```

for generating a 1-D array of n-D dimensions (similar to generating a `Range` object in Ruby) to be used in the iteration.

MASL also natively supports some other containers, such as one for accommodating agents in a continuous 2-D space and a general purpose list. All the specialized containers are implemented with optimization so that various operations can be performed on them efficiently, such as searching, adding or removing an object.

Convenient constructs for general container (list) manipulation like mapping, filtering, etc. are available by allowing passing an anonymous function as a parameter.

```
return neighbors.count(boolean (object cell) {return cell@Live;});
```

There are two important aspects of all containers. First, all kinds of containers can be iterated, and can be customized by users somehow. In this way, the user can choose any one that is proper for a simulation. Second, since there are many agents to be updated during a time step, the container guarantees this work to be done correctly and efficiently, i.e. it keeps track of all the changes of all the agents to be done in a time step and perform the update consistently, even without the user's knowledge.