Tutorial on Using the SSA MASON Extension

Justin Kreikemeyer

November 10, 2021

This tutorial will give a very brief overview of how to define a model using the prototype of the SSA MASON extension introduced in this thesis. The SIR model described in **??** is used as an example.

The first step is very similar to pure MASON: A class is defined, which manages the model and simulation state (Listing 1). We will create a new class called `SirModel`, which inherits from the base class `SSASimState`. This is a subclass of MASON's `SimState`, which would normally be used as a base class. We need to provide a constructor with one argument `seed`, which will also set the SSA to be used by default (In this case, the `FirstReactionMethod` is chosen). The `SSASimState` also provides a second constructor, which can be used to set the simulator when instantiating the model. Furthermore, we need to implement the method `start()`, which will handle the initialisation of the simulation. To run our model, we also provide a main method, like it is usual with MASON.

```
1  package org.tutorial;
2  import org.justinnk.ssamason.extension.*;
3
4  class SirModel extends SSASimState {
5
6      public static void main(String[] args) {
7          doLoop(SirModel.class, args);
8          System.exit(0);
9      }
10
11     /* model parameters/fields... */
12
13     public SirModel(long seed) {
14         super(seed, new FirstReactionMethod());
15     }
16     public void start() {
```

```
17          super.start(); /* model initialisation... */
18      }
19  }
```

Listing 1: The basis of our SIR model.

We now need to add some structures to our model, which provide the environment for our agents (Listing 2). In our SIR model, the environment will consist of a `ContinuousSpace2D` for placing the agents on a grid for visualisation and an (undirected) `Network` of the contact relations. We also use a graph creator from the SSA MASON extension to initialise the edges of our network.

```
1  public Continuous2D world = new Continuous2D(1.0, 100.0,
       100.0);
2  public Network contacts = new Network(false);
3  public GraphCreator graph = new GridGraphCreator();
```

Listing 2: The environment for our agents.

We will also add a couple of parameters to the model (Listing 3). By using getters and setters, these can also be discovered and adjusted through the MASON GUI. The `dom`-prefix can be used to define a maximum slider value for the initially infected humans.

```
1  private double infectionRate = 1.0;
2  public double getInfectionRate() { return infectionRate;
       }
3  public void setInfectionRate(double rate) {
       infectionRate = rate; }
4  private double recoveryRate = 0.5; /* getters and
       setters... */
5  private int numHumans = 100; /* getters and setters...
       */
6  private int initialInfected = 1; /* getters and setters
       ... */
7  public Object domInitialInfected() {
8    return new Interval(0, numHumans);
9  }
```

Listing 3: Parameters of the model.

So far, the process was very similar to MASON. However, defining agents

is a little different. A new agent can be created, by introducing a class, which inherits from the base class `Agent` (Listing 4). In our case, we define the class `Human`, which represents a single human in our SIR model. It will have one public attribute, the `infectionState`. In general, fields that should be tracked by the SSAs that use dependencies must be public fields of an atomic type, like an `Integer`. Public fields within agents, which should not be tracked by the SSA, can be marked with the annotation `@NoAgentAttribute`. In this case, the infection state is of an `enum` type, which has the entries `SUSCEPTIBLE`, `INFECTIOUS` and `RECOVERED`.

```
1  package org.tutorial;
2  import org.justinnk.ssamason.extension.*;
3
4  class Human extends Agent {
5      public enum InfectionState {
6          SUSCEPTIBLE, INFECTIOUS, RECOVERED
7      };
8
9      public InfectionState infectionState =
           InfectionState.SUSCEPTIBLE;
10
11     public Human(SSASimState model) {
12         super(model); /* actions go here */
13     }
14 }
```

Listing 4: The basis of our Human agent.

The human agent has no actions yet, so it won't contribute to the simulation. We change that by adding two actions "infection action" and "recovery action" in the constructor (Listing 5).

```
1  /* cast the state for easy access */
2  SirModel state = (SirModel)model;
3  /* get infected with a certain rate depending on the
       contacts */
4  this.addAction(new Action(
5      () -> this.infectionState == InfectionState.
           SUSCEPTIBLE,
6      () -> {
7          log("caught␣infection␣at␣" + model.schedule.
               getTime());
8          this.infectionState = InfectionState.INFECTIOUS;
```

```
9       },
10      () -> {
11          return state.getInfectionRate() *
                getInfectiousNeighbours();
12      },
13      "infection action"
14  ));
15  /* Recover at a certain rate if infected. */
16  this.addAction(new Action(
17      () -> this.infectionState == InfectionState.
            INFECTIOUS,
18      () -> {
19          log("recovered at " + model.schedule.getTime());
20          this.infectionState = InfectionState.RECOVERED;
21      },
22      () -> state.getRecoveryRate(),
23      "recovery action"
24  ));
```

Listing 5: The actions of our Human agent.

The actions are constructed by passing three lambda functions and an optional name (which is only used for debugging): the `Guard` expression, the `Effect` and the `RateExpression`.

The *guard* returns a boolean expression determining whether the action is applicable in the current simulation/agent state or not. When using a dependency-based SSA (`NextReactionMethod` or `OptimizedDirectMethod`), it may use access to the edges of a network defined in the model, the agent's own attributes or other agents' attributes. Other agents must be discovered by querying the network. Other than access to constant fields of any variety, like a model parameter, no further dependencies are allowed. When using a parameter adjustable through the gui, also bare in mind that changing it during the simulation is not yet supported. Especially, access to the current simulation time is not allowed in the current version.

The *effect* is a function, which describes how the action alters the simulation state. No restrictions apply regarding the access to other agents or the simulation state. The effect does not return anything (void).

The *rate* is used to determine how often (at which rate) the effect is executed in case the guard evaluates to `true`. The same restrictions as for the guard apply. In the rate function for our human, we use the function `getInfectiousNeighbours`, which returns the number of infectious contacts of our agent (Listing 6).

```
1  /** Retrieve the number of infectious neighbours. */
2  private int getInfectiousNeighbours() {
3      Bag contacts = ((SirModel) model).contacts.getEdges(
           this, null);
4      int numInfectiousContacts = 0;
5      for (int contact = 0; contact < contacts.size();
           contact++) {
6          Human contactPerson = (Human) ((Edge) contacts.
               get(contact)).getOtherNode(this);
7          if (contactPerson.infectionState ==
               InfectionState.INFECTIOUS) {
8              numInfectiousContacts += 1;
9          }
10     }
11     return numInfectiousContacts;
12 }
```

Listing 6: A function for determining the number of infectious contacts of the current agent.

The restrictions mentioned here are likely not exhaustive. Despite the early state of development, some of them are already enforced. For example, an error is thrown if the simulation time is accessed in a rate or guard. However, this first prototype is very limited in terms of notifying the user of mistakes that would break the dependency-based SSAs, so a fundamental understanding of the extension and the dependency maintenance is required. It should be mentioned that other SSAs, like the first reaction method, are not limited by these restrictions, but slower for systems with high locality of the effects (like our grid-based SIR model).

We can use the `log` function of the agent class to print information on the console, which will be prepended by the agents id for identification. A unique id is determined automatically when instantiating and agent.

With the human agent completed, we define how the environment is initialised in the start method of our model (Listing 7).

```
1  world.clear();
2  contacts.clear();
3  for (int i = 0; i < numHumans; i++) {
4      Human h = new Human(this);
5      /* Set initial infection */
6      if (i < initialInfected) {
7          h.infectionState = InfectionState.INFECTIOUS;
```

```
8       }
9       /* (visually) arrange humans on a grid */
10      int rows = (int) Math.sqrt(numHumans);
11      world.setObjectLocation(h, new Double2D(world.
            getWidth() * 0.1 + (int) (i / rows) * 8.0,
12                  world.getHeight() * 0.1 + (i % rows) * 8.0))
                    ;
13      contacts.addNode(h);
14  }
15  /* create the grid-structured graph */
16  graph.create(contacts);
```

Listing 7: The initialisation of our environment in the start method of the model.

A number of humans are added to the contact network and also placed in a world for visualisation. Their contacts are initialised by the graph creator, which will connect every human (except the humans at the edges) to its top, right, bottom and left neighbours.

At this point, the simulation can be executed, but will not yield any results. For this, we can implement methods to retrieve the counts of agents in different states. For infectious agents, this might look like shown in Listing 8.

```
1   public int getNumInfected() {
2       return contacts.getAllNodes().stream().filter(h ->
            ((Human)h).infectionState == InfectionState.
            INFECTIOUS).count();
3   }
```

Listing 8: A method probing the number of infectious agents.

This concludes this brief tutorial. As a next step a visualisation class could be added, which works in the same way as described in MASON's excellent manual, to which the reader is referred for further details. The manual can be found at: `https://cs.gmu.edu/~eclab/projects/mason/manual.pdf`.