The standard models of society

Daniel Tang

July 3, 2024

Here we define a heirarchy of standard models which we can use to perform numerical experiments on societies. The models form a heirarchy in that they make increasingly strong commitments but provide a minimal model that can be used as a starting point taking those commitments as given. This helps us make our commitments explicit while providing a modular library that helps us rapidly build models.

1 Root model

For any social simulation we need multiple agents that can interact. Each agent is represetned as an object which implements some number of standard interfaces. Interaction is by one agent executing a method on another agent. We write a call as $\sigma.m(A)$ which signifies that method m on agent σ is called with arguments A. We'll call an instance of a call an *event*. Each method is a function from (σ, A) to an ordered list of events. Agents change their state by calling methods on themselves.

There also exists two special events

Creation event which we write $T(\sigma)$. This creates a new agent of type T in state σ and returns a reference/pointer to the newly created agent.

Deletion event which we write $delete()$. This deletes the calling agent.

This model admits the possibility of dangling pointers (calls to agents that no longer exist). The result of such a call is the termination of the whole simulation.

1.1 The environment

Agents may also interact with non-sentient objects that are thought of as part of the "environment". These are also modelled as objects, but with different interfaces.

1.2 Remote execution dispatch

There are a number of ways of allowing an agent to execute a method on another agent:

Direct execution An agent has a pointer/reference to another agent and directly calls the method. This has the adventage of being fast and simple, and calls can easily return results. However, it can only be used when we're sure there will not be very deep recursion, since each method may call another method. It also makes a commitment about the time ordering of calls.

Thread pool / task queue A method call is packaged as an executable task and added to a queue. The task can be timestamped or the queue itself can be the definition of time ordering. Tasks on the queue are then assigned to a thread from a pool of threads. Tasks can either have void return, or can return a future.

Per agent task queue The tasks can be stored on the called agent itself, along with some way of assigning agents to threads on a pool [although it's not clear how large, if any, efficiency gain there is here].

1.3 Observation

We may wish to allow an agent to observe its environment. An observation is a transfer of information from an observed agent to an observing agent so can be implemented as remote dispatch where the observed agent calls the observer. In this way, observation is a type of agent interaction. An agent can be given an ability to make a certain type of observation by implementing an interface that has a method to receive that observation type. An agent becomes observable by an interface if it implements the code to call agents that have that interface. An observation interaction may be constrained to agents that have certain states, so we define an observation function $o(\sigma_1, \sigma_2)$ that gives a rate or probability that an observation will occur between agents in states $\sigma_1, and\sigma_2$.

1.4 Space and interaction constraints

We may want to restrict the set of agent pairs that can interact depending on their states. That is, there is a set of agent-state pairs, (σ_1, σ_2), that can interact, and agents in other state pairs cannot interact.

If there exists a function $\pi(\sigma)$ such that agents in states σ_1 and σ_2 can interact iff $\pi(\sigma_1) = \pi(\sigma_2)$ then we call $\pi(\sigma)$ the spatial position or location of the agent.

1.5 Time

We assume the existance of an initial event which has no simulated cause. This event causes an ordered list of events, so the cause preceeds the first event in the list and each item in the list preceeds the next. Each of these events in turn cause other lists of events. So, an initial event generates a set of events with a partial ordering which we call the "causal ordering".

An "execution ordering" is a per-agent ordering of method calls. An execution ordering defines a computation. In general, there are a number of execution orderings that satisfy the causal ordering (i.e. that doesn't generate any cycles). The different orderings represent resolutions to the "race conditions" that would exist if each agent had their own processor.

Every physical execution on a real-world computer implies a complete ordering of events given by the real-time ordering of the calls. Note that a program that starts multiple threads may not specify a complete ordering. When executed the resulting simulation may be a draw from a probability distribution over race conditions. A given execution ordering can, in general, be realised by a number of complete orderings, but they all compute the same function.

A theory of simulated time is a function from an initial event to a set of events with an execution ordering. This function may be stochastic. It may be more convenient to define a stochastic function over complete orderings or any other ordering that implies an execution ordering.

1.6 Space-time

One way of generating execution orderings is to have a spatial simulation where each location is associated with a *local time*. Since only agents that share a location can interact, each event takes place at a point in space-time (i.e. at a defined location and local-time) and we have a complete ordering of events in each location, we'll call this a location ordering.

If an agent moves from one location to another (i.e. changes its state to another location) and then interacts with an agent at the new location, the location change event must preceed the interaction event. The location-change call occurs at a local time of the departure location, so we interpret this event as the time of the agent's departure, but we must somehow compare the departure time with the subsequent interaction time at the destination and define some concept of ordering. To do this we define a metric between space-time positions and require that the distance between the departure event and any interaction at the destination must be non-negative. If the distance is positive, this gives the time experienced by the agent between departure and interaction at the new location.

A simple metric is to give each location a coordinate, say (x, y), and define distance as $ds^2 = dx^2 + dy^2 - dt^2$. If we also assume an agent's arrival time is equal to the departure time plus $\sqrt{dx^2 + dy^2}$, then the agent doesn't experience any passage of time during the journey (i.e. we define a frame of reference w.r.t. the locations and assume the agents travel at the "speed of light") and then we need only require that any interaction occurs after an agent's arrival time.

This way of defining the relation between local times has the advantage that an event can only affect interactions within that event's light cone, and this helps when we cone to use multiple processors to run a single simulation.

Even more generally, it is not agents but events (interactions) that have space-time points. We define three functions

1. $P(\pi'|\sigma, \pi)$ which gives the probability that an agent in state σ at point π will enter an interaction at space-time point π'

2. $\rho(R|\sigma_1, \sigma_2, \pi)$ which is the rate that an agent in state σ_1 co-located with an agent in state σ_2 at π will interact and produce a set of agents R at π.

3. $\rho(R|\sigma, \pi)$ the rate that an event will occur that turns an agent in state σ into the agents in R at π.

So, for example, given boundary conditions of two agents in states σ_1 at π_1 and σ_2 at π_2, then the rate of interaction at π_3 is given by

$$\rho(R, \pi_3|\sigma_1, \pi_1, \sigma_2, \pi_2) = \rho(R|\sigma_1, \sigma_2, \pi_3)P(\pi_3|\sigma_1, \pi_1)P(\pi_3|\sigma_2, \pi_2)$$

[Need to decide whether state encodes point of creation or not. Probably best if it does...]

1.6.1 Poisson time

One way of defining a probability distribution over execution orderings is to assume that agents call methods according to a Poisson process and every agent always has a rate vector over all possible calls. An agent's call rate can only change when one of its methods are called or one of its calls are executed. This, along with a space-time metric, induces a distribution over location orderings which in turn induces a distribution over execution orderings.