The standard models of society

Daniel Tang

July 29, 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 Computational model

We build societies on top of the following computational model: A *calculation* is a function object that takes some parameters, performs some computation and returns a set of tasks to be executed at some time after the completion of the calculation, in any order. A task is a calculation that takes no parameters (e.g. a calculation with captured parameters). This implies that a calculation is a tree-shaped partial time-ordering of tasks. If a calculation depends on the completion of more than one parent task, we create a join. A join consists of a "payload" task and a set of "trigger" calculations. Each trigger is sent (as parameter) to a different calculation which executes the trigger, sending data from the calculation as parameter. When a trigger condition is met, the payload is executed with the parameters collected from the triggers. The result of a calculation can be retreived by sending trigger tasks as parameters to the calculation. If the call isn't the final task to complete, it registers the call with the trigger and returns the empty set of tasks, on the final call the object is triggered and calls the payload task, returning its result. In this way we can create any parial ordering without ever having to block. If a task is to return a single task, it may instead call it directly as its last statement. This can be implemented in C++ by defining a task as a lambda function that returns a tuple of lambda functions. In this way the lambdas can capture any local data or references to external state. As much as possible, this can be templated, but if necessary, the lambdas can be wrapped in a `std::function` to perform type removal.

An object oriented version of this consists of making an object a "calculation" which is a funtion object that takes some parameters and returns a set of tasks. A task can be a lambda that captures a calculation or a reference to a calculation along with its parameters. Objects provide a natural way of making data access thread safe. A calculation object is created by its parent task and deleted at

completion of execution. The task has exclusive write access to all its captured data for the duration of its life and should pass any results to the captures of its children tasks. Any non-const pointers should (explicitly or implicitly) be unique pointers. Trigger objects are the only exception, these collect data via some number of "trigger" methods which take a parameter. A trigger object captures a "payload" which is a single-threaded calculation which takes the data from all trigger methods. A trigger object should be created by a task and multiple references passed to child tasks. On triggering the object will delete itself. Data can be reduced by allowing a trigger object to have methods that create its own trigger tasks. The initiating task may take parameters and/or a result object that has some number of trigger methods which take some partial results. A static dataflow graph can be simulated if the initiating task generates an object for each node and each node has triggers for its inputs and trigger-calls for its outputs. However, we can also have dynamic dataflow graphs by allowing nodes to create their own children at runtime. The program can then create different graphs depending on its inputs.

An agent-based version of this: Every agent state maps to a point in a space-time which is a partially ordered set with an ordering operator, $\leq$, such that

- for all points $p \leq p$.
- There doesn't exist two distinct points such that $p \leq q$ and $q \leq p$. i.e. the $\leq$ relation describes a directed acyclic graph.
- if $p \leq q$ and $q \leq r$ then $q \leq r$.
- If an agent is in state p it may only transition to a state q such that $p \leq q$. That is, an agent's trajectory is a path through the partial order DAG.

Given a point p, we call the set of points $\{x : p \leq x\}$ the future light cone of p the points $\{x : x \leq p\}$ the past light-cone of p and the points $\{x :! (x \leq p) \wedge ! (p \leq x)\}$ space-like seaprated from p.

Agents interact via *space-time pointers*, which are pointers to other agents, through which an agent may call a method on another agent. However, a call to another agent is only executed on a target agent once the target is in the future of the caller. Any method may only change the state of the called agent and call methods via other space-time pointers and must return void [return values could be implemented by supplying a lambda to the original call to accept the result. The lambda would add a call to a return space-time pointer, which would then process the value once called].

At any point in the computation each agent is at a space-time position on its trajectory, agents need not be at the same time in any frame of reference so in this way we distinguish simulated space-time from execution-time. When one agent calls another, it creates a method call object (force carrier) which has a space-time position (it's origin) which is the location of the calling agent when the call is made. A call at point p is executed on the target agent's trajectory at the earliest point, q, such that $p \leq q$ (more generally, we can make this probabilistic).

In execution time, an agent at q may not move forward if there exists a potentially calling agent in its past as a call may be created there. So, it must block

until it has no agents in its past. It can then move forward along its trajectory, absorbing calls as it goes, until a potentially calling agent touches its past again. So, at any point in the execution, we can imagine a directed acyclic graph whose nodes are agents and whose edges signify that the parent is blocking the child. There is necessarily at least one agent that is not blocking on any agent (if two or more agents are in the same space-time position we treat them as a single, compound agent until the agents diverge), so the execution can always move forward and non-blocked nodes can be moved forward in any order. So, we can define a 'moveForward()' method that returns moveForward() calls to all agents that it has unblocked by moving forward.

If space and time is continuous we can assume that the agent moves in an inertial reference frame (i.e. has a fixed velocity) until it interacts with another agent or with itself.

Agents become potential callers by opening channels to other agents and interact by creating a call on the channel. A channel can identify a specific interface on a single target agent, or can broadcast to all agents that implement an interface. Pointers can also have a decay rate so their probability of interaction decays with the spatial distance from the origin (in the frame of reference of the calling agent).

It can be shown that the order in which calls are initiated by an agent is also the order in which the calls will be executed on a target agent. So, let a comms-channel be a queue of undelivered calls. The calling agent has an output channel pointer that initiates a call by adding it to the back of the queue. Each item in the queue records the method, arguments and space-time origin, and represents a future light-cone whose origin is the agent's space-time locaton at call initiation. Each target agent has an input channel pointer which points to the front of the queue, which is the next call it will intersect [a broadcast channel can be templated on the broadcast type and can have a static collection of (initiatingAgent,channel) pairs and a non-static front-of-queue pointer]. An input channel has a position given by the origin of the call at the front of the queue, or if the queue is empty, the position of the sending agent, or if the calling agent no longer exists, then $\top$, the point after all points.

If space is discrete, it could be that many calls may share the same origin and target...these calls should commute. Also, in moving from one point to another, an agent disappears from one point in space and re-appears at another point at some later time, so its trajectory consists of jumps from one space-time position to another. This means an agent may block when a jump moves it into the future of an potentially interacting agent, but because of the temporary disappearance of the agent when it moves, the velocity between the call origin and its execution may be slightly slower than light speed, but we don't really care.

Agent dynamics in a discrete space induces a directed graph whose nodes are the spatial positions and an edge from A to B denotes that A can directly preceed B in an agent's trajectory. If each edge has a distance then we can define a spatial distance, $\Delta x(p, q)$, between any two points as the length of the shortest path between them. We can then define $p \leq q$ iff $\Delta x(p, q)^2 \leq c\Delta t(p, q)^2$ where $\Delta t(p, q)$ is the time difference between p and q. Note that the spatial distance is not frame invariant, but if space is discrete and time isn't then there is

Algorithm 1 moveForward() algorithm

```
inChannels ← collection of input channels
iBlockAgents ← collection of agents blocked on this
pos ← agent's current space-time position
loopChannel ← {NextSelfInteraction}
nextChannel ← loopChannel
v ← loopChannel.pos() − pos {4-velocity of this}
nextPos ← loopChannel.pos() {position at next call execution}
for all channel ∈ inChannels do
  if channel.pos() ⊀ pos and channel.pos() < nextPos then
    nextPos ← vt such that |pos + vt − channel.pos()|= 0
    nextChannel ← channel
  end if
end for
onwaredCalls ← ∅
for all blockedAgent ∈ iBlockAgents do
  if |nextPos − blockedAgent.pos()|≠ 0 then
    remove blockedAgent from iBlockAgents
    onwardCalls.add(blockedAgent.moveForwrad())
  end if
end for
simulate forward to nextPos
pos ← nextPos
if nextChannel.isEmpty() then
  nextChannel.sendingAgent().iBlockAgents.add(this)
else
  nextChannel.popAndExecute()
  onwardCalls.add(this.moveForward())
end if
return onwardCalls
```

necessarily a "natural" frame of reference where each discrete spatial point is stationary (assuming space itself isn't deforming).

If time is also discrete then velocities are also discrete, light cones consist of sets of points and trajectories are ordered lists of space-time points.

If space-time is heirarchical (i.e. has an `isIn` relation) we can define the spatial distance between objects as the number of edges in the shortest path between them. We can define the neighbourhood of a container by adding fixed "gateways" between nodes (i.e. an object can leave container A and appear in B at some later time). Gateways define transitions, but distances can be calculated using only the `isIn` tree. The trajectory of an object then becomes a sequence of `isIn(X,t)` and `isNotIn(X,t)` events. When an object moves, it "carries" its whole subtree of contained objects. In between leaving one location and entering another, no events can occur to any of the objects in the tree. This ensures that the velocity (rate of change of distance) of all objects remains bounded irrespective of whether it is actively moving or being carried.

In a discrete space, we can also assume that agents only interact when at the same spatial position, so that a call . In continuous time the call can have a lifetime (or perhaps a half life) at that position, however, the space-time positions that are in an agent's future must be all space-time points it can call by travelling there [So, we should distinguish between the "force-carrying" field which difines the possible execution points given an emmision point, a "can-move-to" partial ordering that bounds the points an agent can appear given its start point, and a "can-interact-with" field that is the convolution of the two and bounds the points a call can be absorbed given an initial point of an agent].

There exists a heterogeneous collection of objects, each of which implements an `interactionState()` method which returns an object with a binary `==` operator. This defines a partition of the space of all objects. There also exists a `step(P)` method which takes a set of objects in a partition and returns a set of objects (possibly stochastically and not necessarily in the same partition). This induces a directed graph whose nodes are partition steps and whose edges indicate that execution on the source partition may return an object in the destination partition, so calculation of the desitination step depends on the source step, we have a computational graph. The partition and execution function should be chosen such that this graph is acyclic. At any point in the computation, let the *active graph* be the (acyclic) subgraph of nodes with non-zero occupation number. In an acyclic graph, there must be at least one node that has no incoming edges. These are the *complete nodes* which can be immediately executed. At any point we have the set of complete nodes.

[In full generality, there is a directed graph in computational time whhose edges mean the source could call the destination. If space-time is itself a set of objects then this can be the graph of pointers that each object has (in computational time). In this view, we reify the non-empty partitions and constrain the movement of objects to neighbouring partitions. This means that partition objects can have a dependency on their neighbours in either direction...effectively, each partition has a watch on the simulated time of each of its neighbours, and a trigger...but this is an observation and the observation can also contain a trigger...so one event schedules other events...easy. This makes a clean separation

of computational and simulated time. It is then left up to the higher levels to ensure no race conditions: there is no guarantee on the order of execution of submitted-but-not-completed events. If events return a set of events, then there is a guarantee only that the returned events occur after the causing event has completed: this gives us the partial ordering. This gives us a tree ordering. To specify a join we create a trigger object which is called by each of its parents on completion, on the final call the object is triggered and calls, or schedules, its body. In this way we can create any parial ordering without ever having to block.]

[]

This model of computation works if both time and space are discrete and we have a synchronous timestep in simulated time. This allows us to step spatial positions forward without necessarily waiting for all other spatial positions to catch up.

If time is continuous and space is discrete, and we include a travel time between spatial positions, then a partition step advances the time of all objects in the position until it reaches the light cone of another object/position. At this point a new object may enter the partition so there is a possible dependency.

If space and time are continuous, then instead of discrete state changes, we have a rate of change of state (i.e. a direction in space-time, within its light cone) and an object follows a continuous trajectory within its light cone. We then ask the probability of an interaction within a volume of space-time [every object has a distribution on a manifold in space-time]

[When space-time is continuous, we have the problem that as we go to the limit of infinite partitions, the probability of objects being in the same state tends to zero. So, we need to define a fixed-scale interaction-kernel in space-time, and define the prob of interaction to be the integral over the joint times the A==B line convolved with the kernel. If the prob changes slowly compared to the width of the kernel, then this will be well approximated by some constant times the line integral along A==B where the probabilities are expressed as probability density per unit volume, this leads to a non-zero limit. This implies a generalisation where the `==` operator is probabilistic and objects can interact between partitions. However, if we define the rate of interaction as a rate per space-time distance (or per second from the frame of the actor[?]) and the state of an object as a probability density per space-time volume, then the line integral of rate times space-time position has a proper limit.]

We build societies on top of the following computational model: There exists a heterogeneous collection of objects, each of which implement some set of interfaces and an overloaded *interaction method*, `episode(A)`, which takes an object that implements a given interface. An agent may have an episode with itself (this can be implemented as `A_1.episode()` if desired). Episodes can only modify the two agents involved, so episodes that involve non-intersecting agents commute. There also exists a partition, S, of the space of all objects such that only agents within the same partition can interact. We call an object's partition its *space-time coordinate*, note that this is an abstraction of its full state. The partitions induce a directed graph whose nodes are partitions and whose edges indicate that there exists an interaction between members of the source parti-

tion that results in an agent in the destination partition. The partition should be chosen such that this graph is acyclic.

2 Root model

A social simulation consists of multiple interacting objects, at least two of which are considered *agents* in that they have *minds* and *bodies*. Each object implements some number of standard *interfaces* each of which consists of a number of *methods*. An *event* is an atomic interaction which is defined as a function $e(x, A, y)$ that signifies that object x has performed action e on object y with arguments A. We also define *autonomous events* that involve only one agent $e(x, A)$ acting on itself.

An event can be defined as a program that takes interface references for x and y and can call the methods on those interfaces, can delete the underlying objects or create new objects. Functionally, this can be thought of as a function that takes object states and arguments and returns a set of object states that contains the post-event states of the passed objects (if they're not deleted) and the states of any new agents created.

A simulation is a directed graph whose nodes are events and whose edges are objects. Each node has associated with it an event type and a set of arguments, its incoming edges (marked as subject and object in binary interactions) are the object references and each of its outgoing edges is an object resulting from the event. Exactly one event is the initial event which takes no agents and exactly one event is the final event which takes any number of agents and deletes them all.

The dynamics of a model is a function that builds a directed graph from the set of objects from the initial event.

2.1 Time-stepping

Events occur at the start of each timestep. Onward events occur at the start of the next timestep. [what if two non-commuting methods are called on an agent in one timestep: how do we decide which order these are called in? We can define an order (e.g. observations first, timestep last), choose a random ordering or require that all methods commute. In general, we define n substeps within the time-step and each method belongs to a substep. All methods in a substep commute. - so, in-fact, non-commutive methods exist in the interfaces but there is a run-time guarantee that in a (sub-)step only commutative methods will be called.]. We can define an order over calls by defining an order over agents: each agent has a call queue, for each agent in agent-order we call the methods on its method queue in order, and this adds calls to other agent's call queues for the next timestep.

2.2 Rate of call

Each method on an agent has a rate of activation which is a function of the states of other agents in the environment. So, the probability that a method will be called in time dt is ρdt. For each method, we define the self-call rate ρ_σ in state σ, and the interaction rate $\rho_{\sigma\sigma'}$. The total rate is given by

$$\rho = \rho_\sigma + \sum_{\sigma'} \rho_{\sigma\sigma'}$$

[If a method call doesn't change the state of the caller, then we can sum over the caller (so we never know who was the caller). If we allow a call to change the state of the caller, then we can't do the sum as it results in different states depending on the caller, so either each method has a set of potential callers, or each caller has a set of potential calls. Or, an event is a complementary pair of calls. Or an event is a method $\sigma_1.m(A, \sigma_2)$ which can call methods on σ_2 and agents cannot store pointers/references to other agents (so a method call can only be a binary interaction). This is the most general.]

2.3 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.

2.4 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].

2.5 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 $\sigma_1, and \sigma_2$.

2.6 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.

2.7 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.

2.8 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. $C(\pi'|\sigma, \pi)$ which gives the rate-coefficient that an agent created in state σ at point π will enter its next interaction at space-time point π'. Or, if we assume homogeneity of space-time then $C(\Delta\pi|\sigma)$ where $\Delta\pi = \pi' - \pi$ is the space-time vector between interactions. [This is a coefficient (amplitude), not a probability. Or is it a probability density in space]

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

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

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, given homogeneity of space-time, is given by

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

and the rate of spontaneous transition is given by

$$\rho(R, \pi_2|\sigma_1, \pi_1) = \rho(R|\sigma_1)C(\pi_2 - \pi_1|\sigma_1)$$

So, an object created at a given space-time point π can be thought of as a rate-coefficient function centred on π.

[This should come out of a sum over paths over time dt.

...or movement is just death and rebirth at another point (with positive space-time distance) at some rate, so space-time position becomes nothing special beyond a variable that afects interaction rate....we can then choose to sum over sponeaneous transitions which gives a probability distribution over states given that no interactions occur. In space-time a rate is a probability per unit volume. If the total probability of interaction is small, then we can consider trajectories with n interactions as n^{th} order perturbations.

Suppose a 1D space with move-right rate a and spontaneous transition rate b so in time dt there's a probability of moving and transitioning

$$\int_0^T at.bdt = ab \int_0^T tdt = ab.T^2/2$$

]

2.8.1 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.