08

Stochastic Simulation in Alchemist

Danilo Pianini
danilo.pianini@unibo.it

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

4 aprile 2016
Introduction

Goals

- Understand the need for fast simulators for complex systems
- Understand the limitations of classic approaches
- Learn a bit of Alchemist

Methodology

- From model checking to Monte Carlo
- Kinetic Monte Carlo (exemplified with chemistry)
- Speed up the Kinetic Monte Carlo
- Alchemist: Kinetic Monte Carlo for Pervasive computing
- Alchemist’s engine
- Alchemist’s model
- Quick tutorial on simulating collective behaviour
Outline

1 Simulation and Montecarlo

2 Exact stochastic simulation of chemical systems
   - The problem and a bit of the math behind
   - Speed up Gillespie

3 Alchemist
   - Motivation
   - Engine
   - Model
   - Architecture
   - Performance
   - Sapere incarnation
   - Simulation with the SAPERE incarnation: a mini-tutorial
### Model checking: a recap

#### Pros
- Complete exploration of the system
- Exact verification of property values

#### Cons
- In general extremely costly in terms of memory and time
- Complexity quickly grows with states
- normally only feasible with simple, small systems
Monte Carlo

Monte Carlo method

- When it’s impossible to explore the whole system
- Find a procedure that randomly explores a part of it
- Apply it repeatedly
- Aggregate the result

Trivia: the name is after the famous Casino of Monte Carlo, and refers to the exploration of the probabilities that gamblers can perform by repeatedly play and recording results.
Monte Carlo method and simulation

The procedure can POSSIBLY (not compulsorily) be a simulation

Example

Which is the combined area $A_F$ of these figures?

- Inscribe them within a rectangle of area $A_R$
- With a uniform distribution sample $N$ points within that rectangle
- Count how many of them are also inside the figures, let this number be $n$
- The area of the figures is (approximately) $A_F = \frac{n}{N} A_R$
- This is not a simulation
Simulation

General definition

Imitation of the operation of a real-world process or system over time [Banks et al., 2010].
- Not necessarily run on computers
  - e.g. putting a Formula 1 model into a wind tunnel is a sort of simulation

Model

The imitation of the real-world process is called **model**.
- A model is a simplified version of the reality
- Simplification is often a requirement, because the original process:
  - requires too much time
  - is not replicable in controlled environments
  - is too dangerous to replicate
  - is beyond our technical capacity
- Elements relevant to the experiment must be retained in the model
Types of computer simulation

**Time-driven**
- Time is simulated through discrete time slots (ticks)
- At every tick, the model is updated to reflect the new state
- All the changes occurring during the same tick are considered to be simultaneous

**Discrete events (DES)**
- Events are simulated one by one
- For every simulated event, the time is shifted forward
- Events are strictly ordered: in case two events are scheduled for the same time, one of the two is executed first (and its outcome may influence the remainder of the simulation)
Kinetic Monte Carlo and chemistry

Problem: we have a container with a precise number of molecules that may react with each other. We want to forecast the evolution of the system in future.

Relax to Continuous

- In classic chemistry, there are methods based on differential equations to understand the behaviour of such systems.
- They suppose the concentration of each reactant to be $\in \mathbb{R}$.
- It is an approximation: you cannot have a quarter of a molecule!
- These methods are accurate only for a high number of molecules.

Stochastic simulation

- What if our system counts few thousands molecules?
- Monte Carlo way: let’s start with the system in initial state, let it run and see how it behaves. Repeat.
- Very hard to do in a real setup: here it comes the simulation.
Outline

1 Simulation and Montecarlo

2 Exact stochastic simulation of chemical systems
   - The problem and a bit of the math behind
   - Speed up Gillespie

3 Alchemist
   - Motivation
   - Engine
   - Model
   - Architecture
   - Performance
   - Sapere incarnation
   - Simulation with the SAPERE incarnation: a mini-tutorial
Approach the problem

We need to find a procedure for simulating a chemical system. The system is composed of molecules and reactions. Reactions assume the form:

\[ A + B \xrightarrow{\mu} C \]

where \( A \) and \( B \) are reactants, \( \mu \) is an indication of the reaction speed and \( C \) is the product.

A solution has been first proposed in [Gillespie, 1977] (Gillespie algorithm or Kinetic Monte Carlo):

1. Select next reaction using markovian rates: it supposes that a chemical system has no memory, and computes the speed of a reaction \( r \) as: \( a_r = [A][B]\mu \)
2. Execute it, changing the concentrations
3. Update the markovian rates which may have changed
Do the math: next reaction choice

If we assume every reaction is a Poisson process, the probability for it to be the next one is:

\[
P(\text{next} = \mu) = \int_0^\infty P(\mu, \tau) d\tau = \int_0^\infty a\mu e^{-\tau} \sum_j a_j d\tau = \frac{a\mu}{\sum_j a_j}
\]

Details

- \(P(\mu, \tau) = a\mu e^{-\tau} \sum_j a_j\): the probability that the reaction \(P\) occurs at time \(\tau\) is its speed times the probability distribution. Being this a Poisson process, the probability distribution is a negative exponential function, whose exponent is the sum of the speeds of all the reactions in the system.
Do the math: next reaction time

We can also compute the next time of occurrence:

\[ P(\tau) d\tau = \sum_j P(\mu = j, \tau) d\tau = \left( \sum_j a_j \right) e^{-\tau \sum_j a_j} d\tau \]

\[ \sum_j a_j = \lambda \rightarrow \lambda e^{-\lambda x} \]

\[ F(x \leq t) = \int_{-\infty}^{t} \lambda e^{-\lambda x} \, dx = \int_{0}^{t} \lambda e^{-\lambda x} \, dx = \left[ -e^{-\lambda t} \right]_{0}^{t} = 1 - e^{-\lambda t} \]

Now, given a uniformly distributed random \( \rho \) in \([0, 1]\), it’s possible to compute it’s equivalent for the desired distribution:

\[ 1 - e^{-\lambda t} = \rho \Rightarrow t = \frac{-\ln (1 - \rho)}{\lambda} \equiv \frac{-\ln (\rho)}{\lambda} \]
Solve the problem: base algorithm

**Algorithm**

1. Set the simulation time \( T = 0 \)
2. For each reaction \( r \) in the whole set of reactions \( R \), compute \( a_r \).
3. Select the next reaction \( \mu \) to execute. The probability for \( r \) to be executed will be \( P(r = \mu) = \frac{a_r}{\sum_{(j \in R)} a_j} \).
4. Execute the reaction, changing the concentrations.
5. Set the simulation time to \( T = T_{prev} - \frac{\ln(1-\gamma)}{\lambda} \).
6. GOTO 2

**Data structures**

- Choosing the next reaction to execute can be done by storing in a list like structure reactions and propensities, throwing a random number in \( 0 : \sum_{(j \in R)} a_j \), and selecting the first reaction whose propensity summed to all the previous is equal or higher than the random (linear complexity in time).
Speed it up: dependency graph

Algorithm

1. The propensities must be recomputed at each step, because they depend on concentration of reactants, which may have changed.

2. However, not every reaction affects the speed of every other: for instance, if $A + B \xrightarrow{\mu_1} C$ executes, the propensity of $D + E \xrightarrow{\mu_2} A$ will not be affected.

3. We can improve consistently the performance of the algorithm by keeping in memory which reactions influence which other, and updating only those required.

Data structures

- A map that connects each reaction to a set of reactions that must be upgraded represents a good dependency graph.
Speed it up: dependency graph

A + B → C
B + C → D
E + G → A
D + E → E + F
F → D + G
**Algorithm**

1. Instead of choosing the next reaction probabilistically by propensity, generate a putative time for each reaction.
2. Sort the reactions by putative time, and take the first.
3. At each step, for each reaction whose putative time has changed, re-sort the element.
4. The previous optimization (dependency graph) can be reused.

**Data structures**

- We only need that the first element is the next to be executed.
- The best solution is a binary heap*, which can be accessed in $O(1)$ and sorted in $log(n)$, but with a much smaller average complexity.

* In the original work [Gibson and Bruck, 2000], the data structure is called “Indexed priority queue”.
Next reaction: random reuse

Random generation

1. Generating random number is costly
2. In a purely chemical simulator, is the most heavy task
   [Gibson and Bruck, 2000]
3. Reducing the number of generated random numbers is key

Random reuse

- Next reaction allows for random reuse
- In case the reaction which is being updated is not the one executed but one of its dependencies, then:
  - let $T$ be the current simulation time, $\tau_c$ be the new putative time, $a_c$ the new propensity, $\tau_p$ the previous putative time, $a_p$ the previous propensity.
  - $\tau_c = \frac{a_c(\tau_p - T)}{a_p} + T$
  - This is possible due to the exponential distribution being memory less
  - Note: $\forall \tau_p, T : \tau_p \geq T$
Binary heap
## Sleppy’s Algorithm

### Idea
- Divide the reactions in groups depending on their propensity
- Define the groups in such a way that throwing a limited number of randoms the engine can select the next to execute in constant time
- Updating the reactions can be done in constant time since the groups have a well defined propensity interval
- If the number of groups does not depend on the number of reactions, then the algorithm is $O(1)$.

### Drawbacks
- The algorithm assumes that the number of groups does not depend on the number of reactions, namely, it supposes the propensities to change only a little during the simulation
- This assumption is mostly true in real purely chemical systems, but does not hold in general
Slepoy’s Algorithm

From [Slepoy et al., 2008]
Outline

1. Simulation and Montecarlo
   - Exact stochastic simulation of chemical systems
   - The problem and a bit of the math behind
   - Speed up Gillespie

2. Alchemist
   - Motivation
   - Engine
   - Model
   - Architecture
   - Performance
   - Sapere incarnation
   - Simulation with the SAPERE incarnation: a mini-tutorial
Background considerations

- Pervasive computing scenarios are normally simulated by means of “agent based simulators” (ABS) [Wooldridge and Jennings, 1995]
- ABS are extremely flexible, but they lack performance: it’s the price to pay for being able to simulate a very wide spectrum of situations
- Many pervasive computing scenarios can be modelled as mobile multi-compartmented chemical systems, where molecules are pieces of data (equivalent to a network of Petri Nets)
- A whole literature exists on how to make very fast kinetic Monte Carlo algorithms

Idea

Instead of use classic ABSs and optimize at the simulation level, can we take a kinetic Monte Carlo and extend it until it supports all the abstractions we need?
Which scenarios

We want a tool that supports:

- Self-organising systems
- Pervasive computing systems
- Crowds of people
- Large scale situated systems
- Smart Mobility
- Crowd detection and steering
- Sensor networks
- Computational biology
- Aggregate programming
From chemistry to pervasive computing

Requirements

- Multiple compartments (from now on: nodes)
- Molecules can be different data types
- Nodes mobility
- Non Markovian events
- More flexible concept of reaction
- High performance

Idea

Instead of using a classic ABS and optimize at the simulation level, can we take a kinetic Monte Carlo and extend it until it supports all the abstractions we need?
Multiple compartments

Extension
- Up to now we just used a single container with molecules
- What if we had multiple intercommunicating containers?

Changes
- Concept of “neighborhood”, namely the compartments that can communicate with each compartment
- Concept of moving molecules from a compartment to another
- Possibly different set of reactions for each compartment

Challenges
- Who does decide if two compartments are communicating?
- How to model a molecule moving towards a new node?
- How does the dependency graph change?
Spatial dependency graph

Challenge

- There are more reactions: each node has its “copy”
- A reaction may affect the propensities locally, in the neighborhood, or globally
- The fewer are the bindings between reactions, the higher is efficiency of a dependency graph
- We want to detect the context of the reactions and filter the dependencies accordingly
Spatial dependency graph

Possible solution

- Define three contextual levels: *local*, *neighborhood*, *global*
- Assign to each reaction an “input context”, namely which parts of the environment a reaction should read to compute its propensity.
- Assign to each reaction an “output context”, namely which part of the environment will be modified by this reaction.
- A reaction $r_1$ may influence a reaction $r_2$ if one of the following is true:
  - $r_1$ and $r_2$ belong to the same compartment.
  - $r_1$’s output context is *global*.
  - $r_2$’s input context is *global*.
  - $r_1$’s output context is *neighborhood* and $r_2$ belong to a compartment of the neighborhood.
  - $r_2$’s input context is *neighborhood* and $r_1$ belong to a compartment of the neighborhood.
  - Both $r_1$’s output context and $r_2$’s input context are *neighborhood*, and there is a compartment shared by the two neighborhoods.
Non-markovian events

Example

Every second, an external device injects some quantity of molecules within a compartment.

- this event happens precisely every second: it is not a Poisson process!
- Its probability distribution is a $\delta$-Dirac Comb

Algorithms

- The basic Gillespie algorithm is hard to modify to support such events. The main reason is that the choice is not made depending on time, but on propensity, which is an entity strictly bound to the markovian model.
- The next reaction algorithm, instead, uses putative times: this makes it able to simulate events independently from their distribution, since we just need to correctly estimate the next time of occurrence.

  ▶ NOTE: the random reuse is NOT allowed for non-exponential events
Abstract model

Concentration
Actual data associated with a "molecule"

Environment
Riemannian manifold where nodes live

Reaction
A proactive behavior

Linking Rule
A function of the environment that decides whether or not two nodes are connected

Node
A container of reactions and molecules situated in the environment

Molecule
token representing a chunk of data (think of it as a pointer)

Danilo Pianini (Università di Bologna)
Reactions

Reaction

Conditions

- Number of neighbors < 3
- Node contains something
- Any other condition about this environment

Probability distribution

Rate equation: how conditions influence the execution speed

Actions

- Move a node towards...
- Change concentration of something
- Any other action on this environment

Danilo Pianini (Università di Bologna)
ISAC08 – StSimulation 4 aprile 2016 31 / 60
Architecture

YAML interpreter → Create and populate model
CLI Interpreter → Create model

Model
- Environment
- Incarnation

Incarnation-specific entities
- Nodes
- Reactions
- Molecules
- Conditions
- Actions
- Time Distributions
Against Repast

Performance comparison with Repast

 Execution time [s]  

<table>
<thead>
<tr>
<th>Number of agents</th>
<th>Repast</th>
<th>Alchemist</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td></td>
<td></td>
</tr>
<tr>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>350</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Danilo Pianini (Università di Bologna)
SAPERE incarnation

Motivation

- Alchemist was initially developed within the SAPERE Project (http://www.sapere-project.eu/)
- At the model level, captures the required abstractions of a SAPERE system

Details on the incarnation

- In this incarnation, the concentration is defined as “list of tuples matching a tuple template”
- Basically, in this configuration Alchemist does not simulate a simple collection of intercommunicating compartments, but a network of (possibly mobile) programmable tuple spaces
- This one and the incarnation supporting Protelis based aggregate programming are the only two completely implemented
  - there is a sketched biochemical implementation
Simulating in Alchemist

**Alchemist XML**

Alchemist 1.+ provides support for writing simulations using an XML file
- Inspired by CellML, very verbose: a file can get well over 10MB
- Not human friendly
- Each incarnation normally provides also a DSL that translates a human friendly language to the XML
- Considered legacy, deprecated

**Alchemist YAML**

Alchemist 2.+ adds support for writing YAML instead
- Human readable and small in size
- Demands creation of actual model objects to the incarnation
- Works for any (correctly implemented) incarnation
- It is still possible to write DSLs if the need arises
Simulate using SAPERE

Using the Alchemist-SAPERE DSL
- the DSL exposes the SAPERE Incarnation concepts directly, allowing for short specifications
- a compiler automatically produces the Alchemist XML
- developed with the Xtext framework

Using YAML
- No XML involved
- The same syntax can be used for any incarnation
- Support for running batches
- No intermediate compilation
- No large files involved

→ We’ll use the new method
Minimal specification

YAML version:

```yaml
incarnation: sapere
```

DSL version:

```sql
default environment
linking nodes in range 0
```

XML translation:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<environment name="environment" type="Continuous2DEnvironment">
  <linkingrule type="EuclideanDistance" p0="0"></linkingrule>
  <concentration type="LsaConcentration"></concentration>
  <position type="Continuous2DEuclidean"></position>
  <random type="MersenneTwister" seed="RANDOM"></random>
</environment>
```
Single nodes

YAML

incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [5]

displacements:
  - in:
      type: Point
      parameters: [0, 0]
  - in:
      type: Point
      parameters: [0, 1]
Single nodes (legacy approach)

Default environment
Linking nodes in range 5
Place single node at point (0,0)
Place single node at point (0,1)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<environment name="environment" type="Continuous2DEnvironment">
  <linkingrule type="EuclideanDistance" p0="5"></linkingrule>
  <concentration type="LsaConcentration"></concentration>
  <position type="Continuous2DEuclidean"></position>
  <random type="MersenneTwister" seed="RANDOM"></random>
  <node name="group_0_node_0" type="LsaNode" position="0.0,0.0">
    <content></content>
  </node>
  <node name="group_1_node_0" type="LsaNode" position="0.0,1.0">
    <content></content>
  </node>
</environment>
```
Multiple nodes

incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [0.5]

displacements:
  - in:
    type: Circle
    parameters: [10000, 0, 0, 10]

default environment
linking nodes in range 0.5
place 10000 nodes within circle (0,0,10)

The corresponding XML is 30007 lines of code: it has a separate node tag for each node in the scenario, making it impossible to write by hand, and unconvenient for data exchange.
incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [0.5]

displacements:
  - in:
    type: Grid
    parameters: [-5, -5, 5, 5, 0.25, 0.25, 0, 0]
Irregular grid of nodes

incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [0.5]

displacements:
  - in:
    type: Grid
    parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]
incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [0.5]

displacements:
  - in:
    type: Grid
    parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]

contents:
  - molecule: hello
  - in:
    type: Rectangle
    parameters: [-1, -1, 2, 2]
  molecule: token
incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [0.5]

displacements:
  - in:
    type: Grid
    parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]

contents:
  - in:
    type: Rectangle
    parameters: [-0.5, -0.5, 1, 1]

molecule: token

programs:
  -
    - time-distribution: 1
      program: >
      {token} --> {firing}
    - program: "{firing} --> +{token}"
Code reuse in YAML

incarnation: sapere

network-model:
  - type: EuclideanDistance
  - parameters: [0.5]

send: &send
  - time-distribution: 1
    program: >
      {token} --> {firing}
  - program: "{firing} --> +{token}"

displacements:
  - in:
    type: Grid
    - parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]

contents:
  - in:
    type: Rectangle
    - parameters: [-0.5, -0.5, 1, 1]

molecule: token

programs:
  - *send
incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [0.5]

send: &send
  - time-distribution: 1
    program: >
      {token} --> {token} *{token}
  - program: >
    {token}{token} --> {token}

displacements:
  - in:
    type: Grid
    parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]

contents:
  - in:
    type: Rectangle
    parameters: [-0.5, -0.5, 1, 1]
    molecule: token

programs:
  - *send
Mathematical operations

incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [0.35]

send: \&grad
  - time-distribution: 0.1
    program: \"{source} \rightarrow \{source\} \{gradient, 0\}\"
  - time-distribution: 1
    program: \"{gradient, N} \rightarrow \{gradient, N\} \{gradient, N+\#D\}\"
    program: \{gradient, N\}{gradient, def: N2>=N} \rightarrow \{gradient, N\}
  - time-distribution: 0.1
    program: \{gradient, N\} \rightarrow \{gradient, N + 1\}
    program: \{gradient, def: N > 30\} \rightarrow 

displacements:
  - in:
    type: Grid
    parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]

contents:
  - in:
    type: Rectangle
    parameters: [-0.5, -0.5, 1, 1]
    molecule: source

programs:
  - *grad
Synthetic variables

- #ID – unique id for each LSA
- #NODE – this node id
- #O – the “orientation”, namely the node id of the local node when an operation involving the neighborhood is performed
- #D – distance with the neighbor
- #T – current time
- #RANDOM – a random number
- #NEIGHBORHOOD – list of all neighbors ids
- #SELECTEDNEIGH – neighbor selected when performing a “+” operation
- #ROUTE – distance using routes, only works with maps
incarnation: sapere

network-model:
  type: EuclideanDistance
  parameters: [0.5]

send: &grad
  - time-distribution:
    type: DiracComb
    parameters: [0.5]
    program: "\{token, N, L\} \rightarrow \{token, N, L\} \{\text{token, N+D, L add [#NODE;]}}"
  - program: >
    \{token, N, L\}{\text{token, def: N2>=N, L2}} \rightarrow \{token, N, L\}

displacements:
  - in:
    type: Grid
    parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]
    contents:
      - in:
        type: Rectangle
        parameters: [-0.5, -0.5, 1, 1]
        molecule: token, 0, []
    programs:
      - *grad
Personalised time distribution

Considerations:

- The syntax is a shortcut for the desired Java class’ constructor
- You can implement your own classes implementing TimeDistribution and model arbitrary distributions
- Alchemist is a discrete-event simulator: events are forced to be ordered, even if they happen at the same time.
- The same syntax (a YAML map with type and parameters key) can be used to load arbitrary implementations of any simulation element
- The Alchemist loader automatically assigns values to arguments of type Environment, Node, Reaction, and TimeDistribution depending on the context, letting the user specifying only the parts strictly required.
The variables section

incarnation: sapere

variables:
  rate: &rate
    type: GeometricVariable
    parameters: [2, 0.1, 10, 9]
  size: &size
    min: 1
    max: 10
    step: 1
    default: 5
  mSize: &mSize
    formula: -$size
  sourceStart: &sourceStart
    formula: $mSize / 10
  sourceSize: &sourceSize
    formula: $size / 5
The variables section

network-model:
  type: EuclideanDistance
  parameters: [0.5]

send: &grad
  - time-distribution: *rate
    program: "{token, N, L} --> {token, N, L} *{token, N+#D, L add [#NODE;]"
  - program: >
    {token, N, L}{token, def: N2>=N, L2} --> {token, N, L}

displacements:
  - in:
    type: Grid
    parameters: [*mSize, *mSize, *size, *size, 0.25, 0.25, 0.1, 0.1]

contents:
  - in:
    type: Rectangle
    molecule: token, 0, []

programs:
  - *grad
Variables in Alchemist

- Variables can be defined in a variable section
- They are implementations of the Variable interface
- Very useful for running batches
- They can be specified as dependent variables by indicating a formula (that will then be interpreted by an internal Javascript engine)
package it.unibo.alchemist.model.implementations.actions;

import...

public class BrownianMove<T> extends AbstractMoveNode<T> {

    private final double r;
    private final RandomEngine rng;

    public BrownianMove(final IEnvironment<T> environment, final INode<T> node,
            final RandomEngine rand, final double range) {
        super(environment, node);
        r = range;
        rng = rand;
    }

    @Override
    public IPosition getNextPosition() {
        return new Continuous2DEuclidean(genRandom() * r, genRandom() * r);
    }

    private double genRandom() {
        return rng.nextFloat() - 0.5f;
    }

    @Override
    public IAction<T> cloneOnNewNode(final INode<T> n, final IReaction<T> reaction) {
        return new BrownianMove<>(getEnvironment(), n, rng, r);
    }

}
incarnation: sapere

variables:
  rate: &rate
    type: GeometricVariable
    parameters: [1, 0.1, 10, 9]
  size: &size
    min: 1
    max: 10
    step: 1
    default: 5
  mSize: &mSize
    formula: -$size
  sourceStart: &sourceStart
    formula: $mSize / 10
  sourceSize: &sourceSize
    formula: $size / 5

network-model:
  type: EuclideanDistance
  parameters: [0.5]
send: \&grad
  - time-distribution: \*rate
    program: "\{token, N, L\} \rightarrow \{token, N, L\} \*\{token, N+\#D, L add [\#NODE;]\}"
  - program: >
    \{token, N, L\}\{token, def: N2>=N, L2\} \rightarrow \{token, N, L\}

# Age information
- time-distribution:
  type: DiracComb
  parameters: [0.1]
  program: >
    \{token, def: N>0, L\} \rightarrow \{token, def: N + 1, L\}

# Cleanup old information
  program: >
    \{token, def: N>30, L\} \rightarrow

move: \&move
- time-distribution: 0.5
  type: Event
  actions:
  - type: BrownianMove
    parameters: [0.1]
displacements:
  - in:
    type: Grid
    parameters: [*mSize, *mSize, *size, *size, 0.25, 0.25, 0.1, 0.1]
contents:
  - in:
    type: Rectangle
molecule: token, 0, []
programs:
  - *grad
  - *move
Non exhaustive Alchemist UI keyboard shortcuts

P  Pause/play
L  enables and disables link painting
R  Enables and disables realtime mode: tries to sync the simulation with the real time, always ensuring at least 25fps.
→  Makes the simulation faster (less update calls to the UI)
←  Makes the simulation slower (more update calls to the UI)
M  Turns on and off the graphical marker for the node closest to the mouse pointer
S  Enters select mode (nodes can be selected)
O  When in select mode, enables manual move mode for selected nodes

