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

6 novembre 2017

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

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

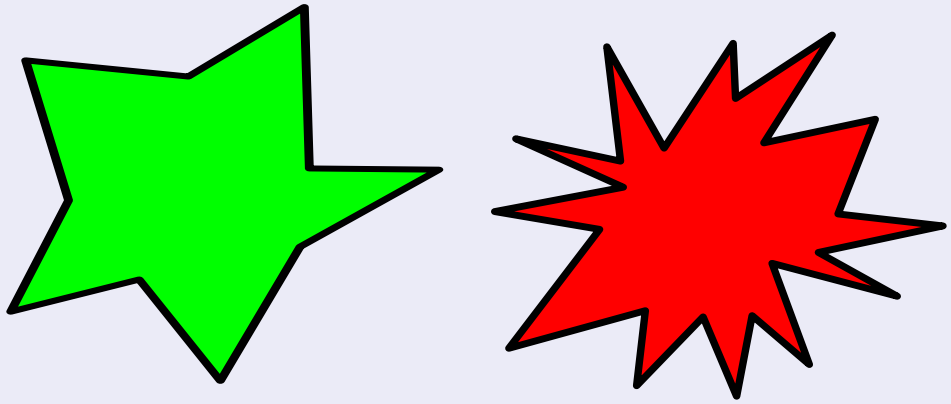
Trivia: the name is after the famous Casino of Monte Carlo, and refers to the exploration of the probabilities that gamblers can perform by repeatedly play and recording results.

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?

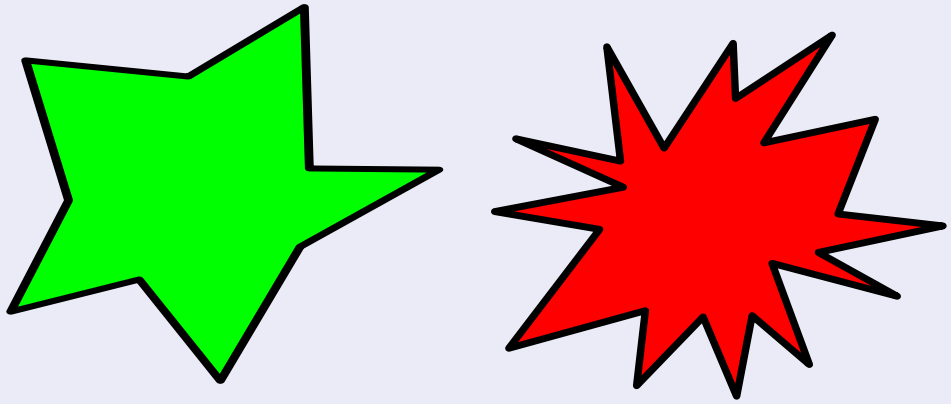


Monte Carlo method and simulation

The procedure can POSSIBLY (not compulsorily) be a simulation

Example

Which is the combined area A_F of these figures?



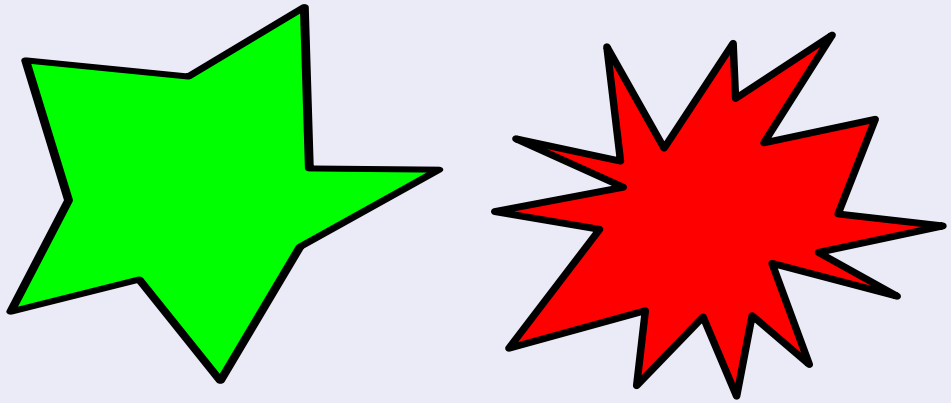
- Inscribe them within a rectangle of area A_R
- With a uniform distribution sample N points within that rectangle
- Count how many of them are also inside the figures, let this number be n
- The area of the figures is (approximately) $A_F = \frac{n}{N} A_R$

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 $\in \Re$
- 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

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, μ 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:

$$P(next = \mu) = \int_0^\infty P(\mu, \tau) d\tau = \int_0^\infty a_\mu e^{-\tau \sum_j a_j} d\tau = a_\mu \int_0^\infty e^{-\tau \sum_j a_j} d\tau$$

$$= -\frac{a_\mu}{\sum_j a_j} [e^{-\tau \sum_j a_j}]_0^\infty = -\frac{a_\mu}{\sum_j a_j} (e^{-\infty} - e^0) = -\frac{a_\mu}{\sum_j a_j} (-1) = \frac{a_\mu}{\sum_j a_j}$$

Details

- $P(\mu, \tau) = a_\mu e^{-\tau \sum_j a_j}$: the probability that the reaction P occurs at time τ 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:

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

$$\sum_j a_j = \lambda \longrightarrow \lambda e^{-\lambda x}$$

$$F(x \leq t) = \int_{-\infty}^{t} \lambda e^{-\lambda x} dx = \int_0^t \lambda e^{-\lambda x} dx = \left[-e^{-\lambda t}\right]_0^t = 1 - e^{-\lambda t}$$

Now, given a uniformly distributed random ρ 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 μ 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 $T = T_{prev} - \frac{\ln(1-r)}{\lambda}$
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 $\left[0 : \sum_{(j \in R)} a_j\right]$, 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

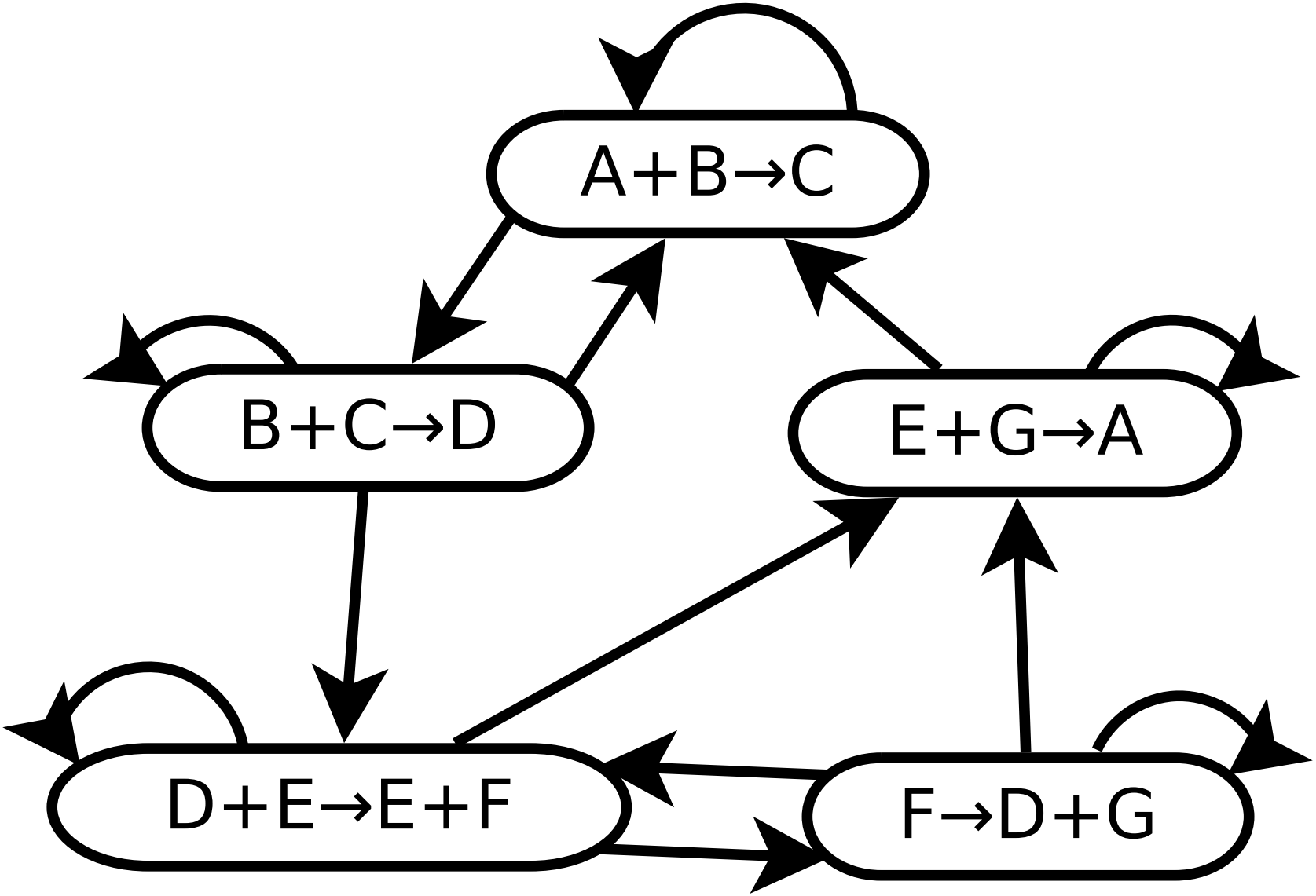
Algorithm

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



Next reaction

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

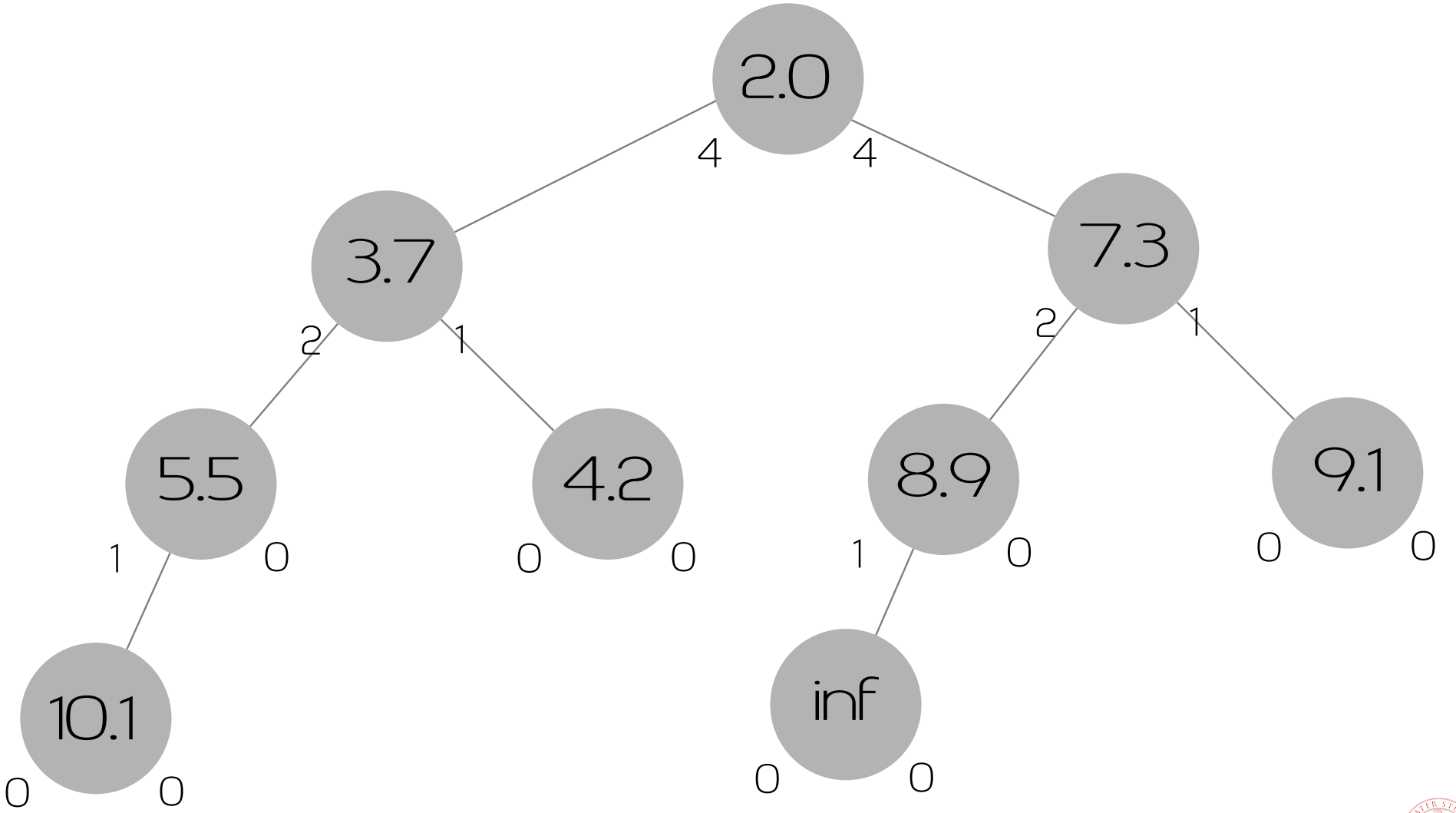
Random generation

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, τ_c be the new putative time, a_c the new propensity, τ_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



Slepoy's Algorithm

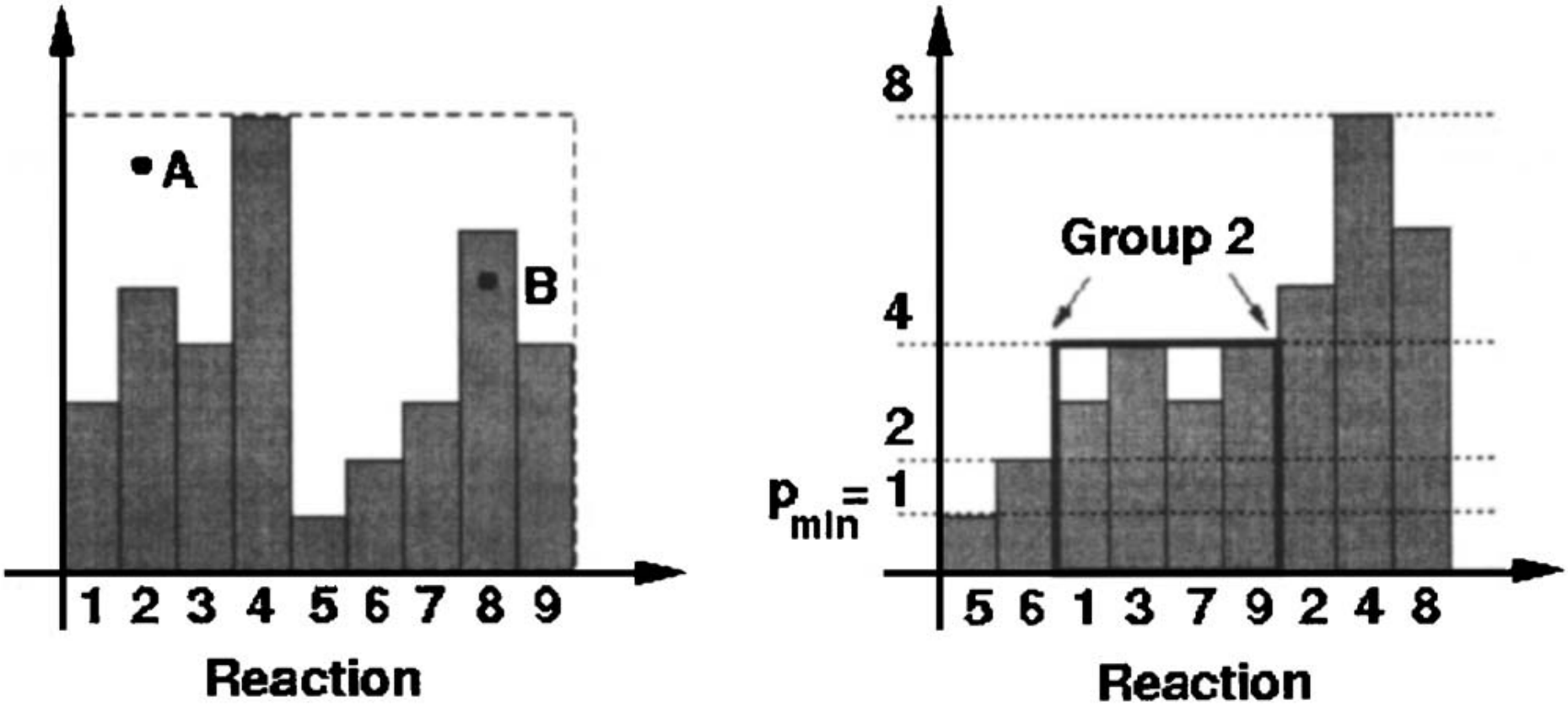
Idea

- 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

From chemistry to pervasive computing

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 is propensity
- Assign to each reaction an "output context", namely which part of the environment will be modified by this reaction
- A reaction `r1` may influence a reaction `r2` if one of the following is true:
  - `r1` and `r2` belong to the same compartment
  - `r1`'s output context is *global*
  - `r2`'s input context is *global*
  - `r1`'s output context is *neighborhood* and `r2` belongs to the neighborhood
  - `r2`'s input context is *neighborhood* and `r1` belongs the neighborhood
  - both `r1`'s output context and `r2`'s input context are *neighborhood*, and there is a compartment shared by the two neighborhoods

Non-markovian events

Example

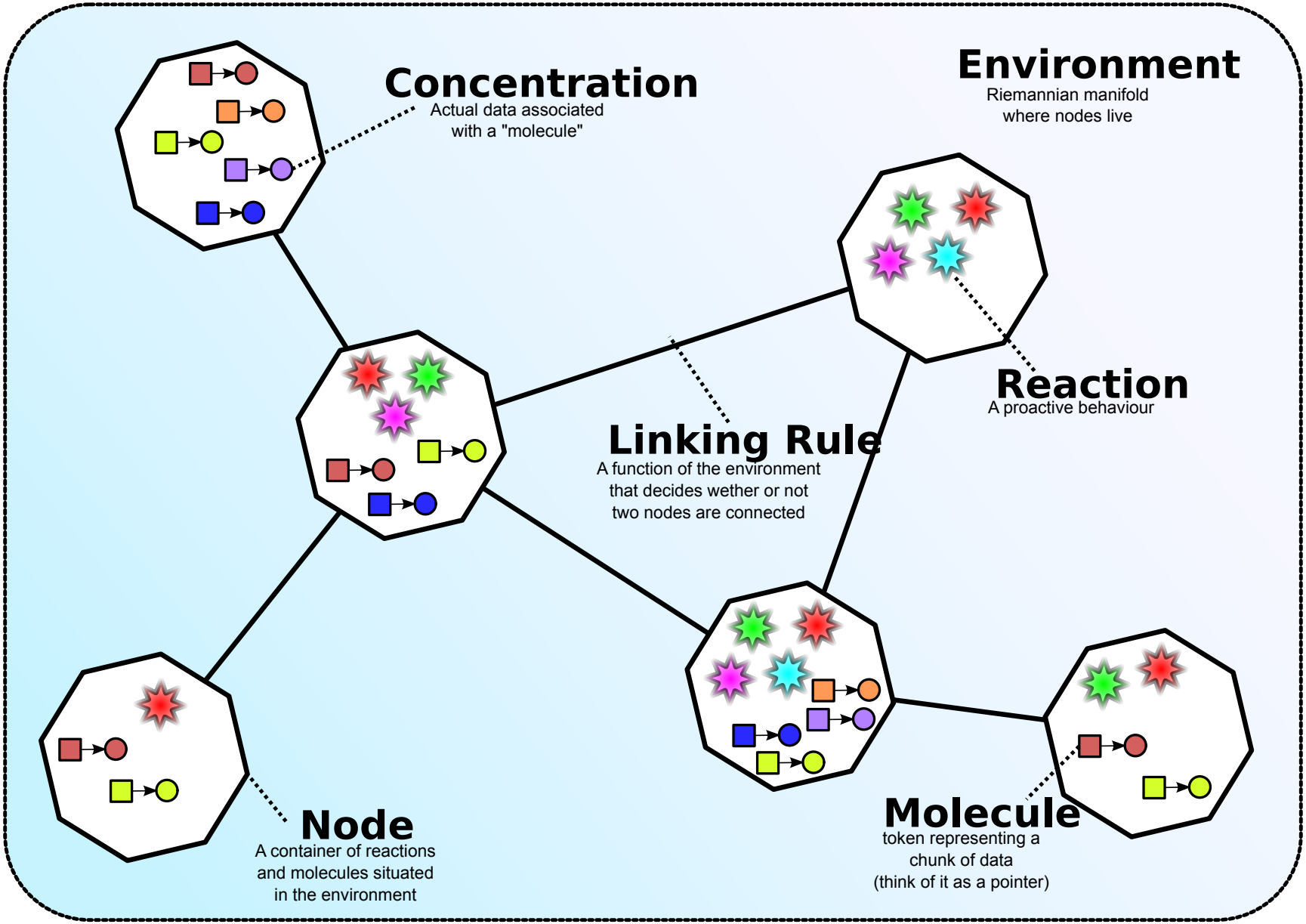
Every second, an external device injects some quantity of molecules within a compartment.

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

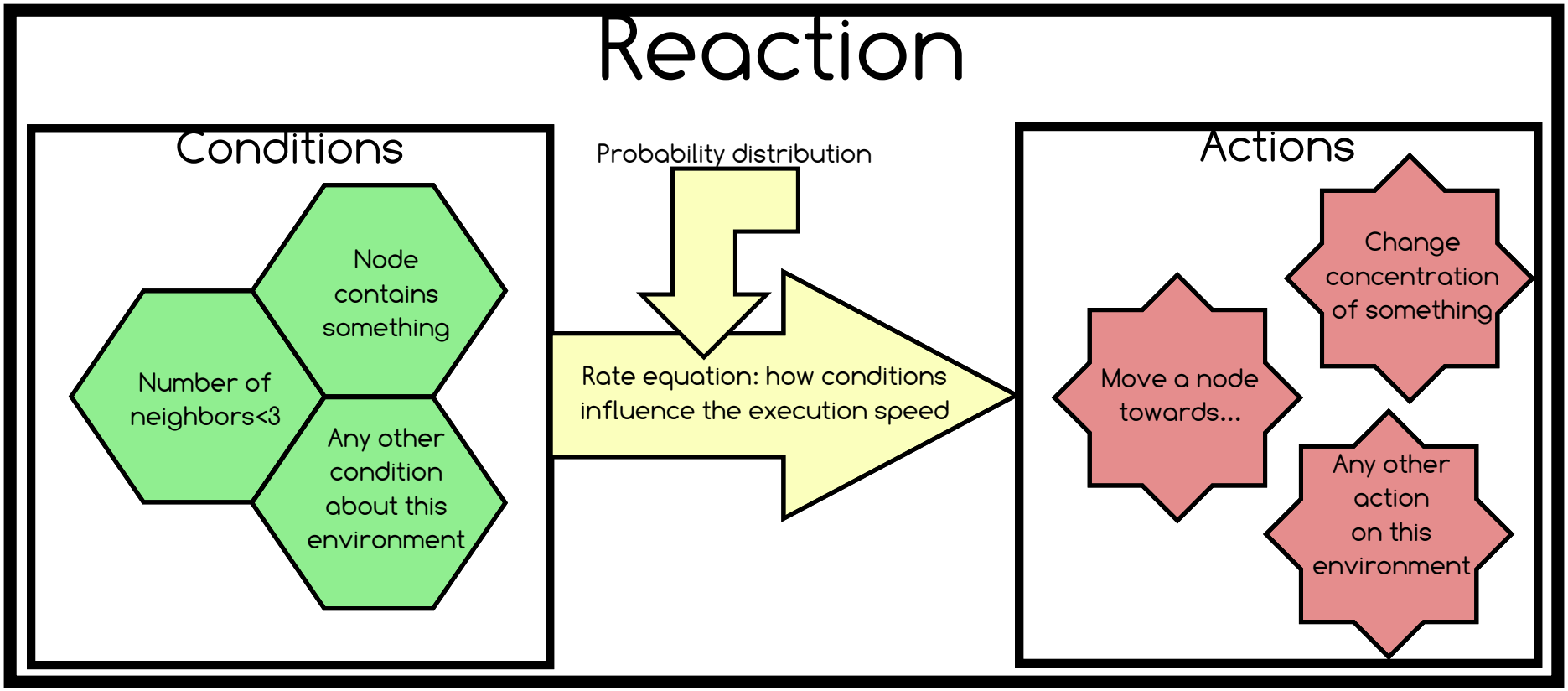
Algorithms

- 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

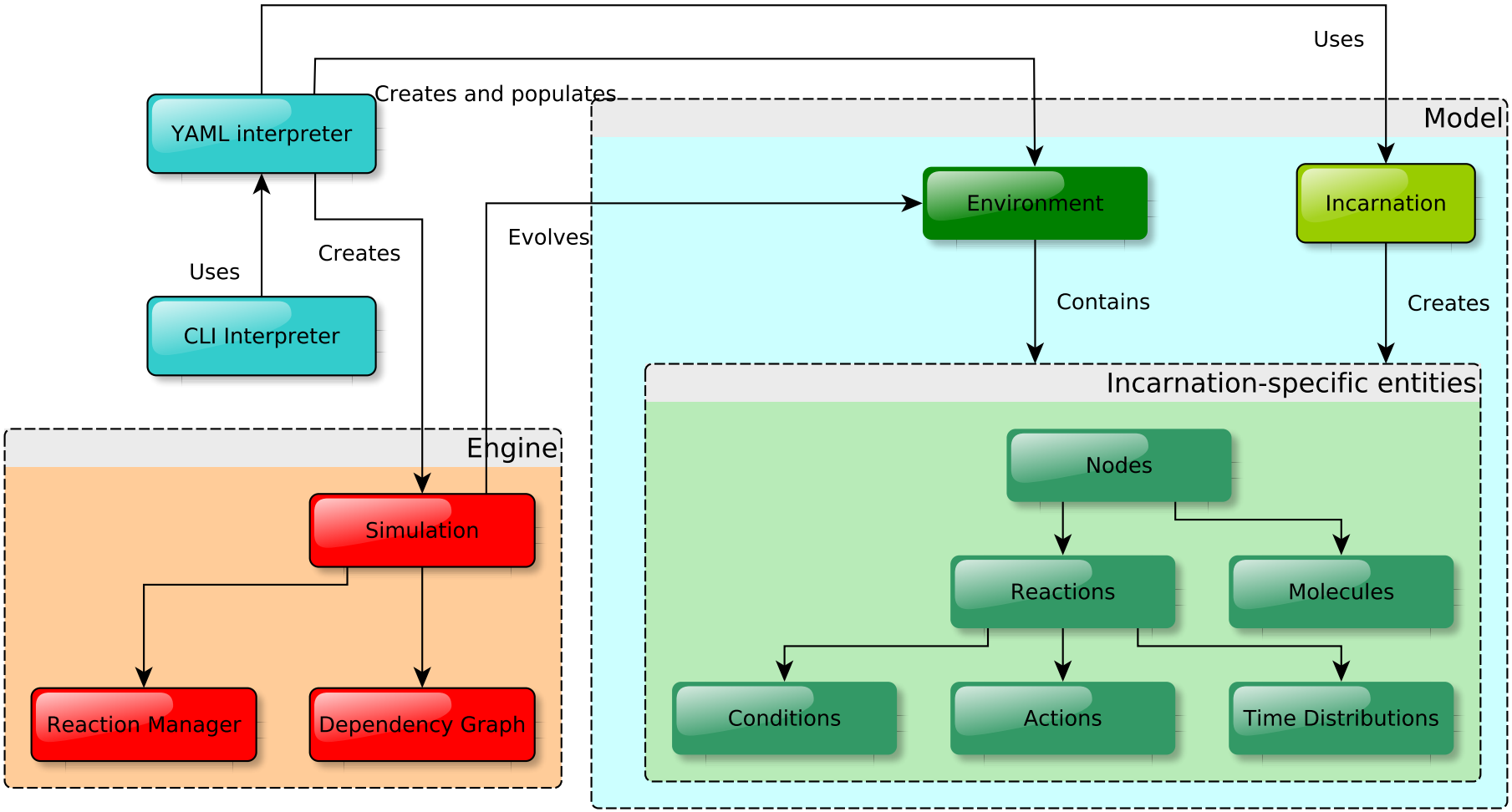
Abstract model



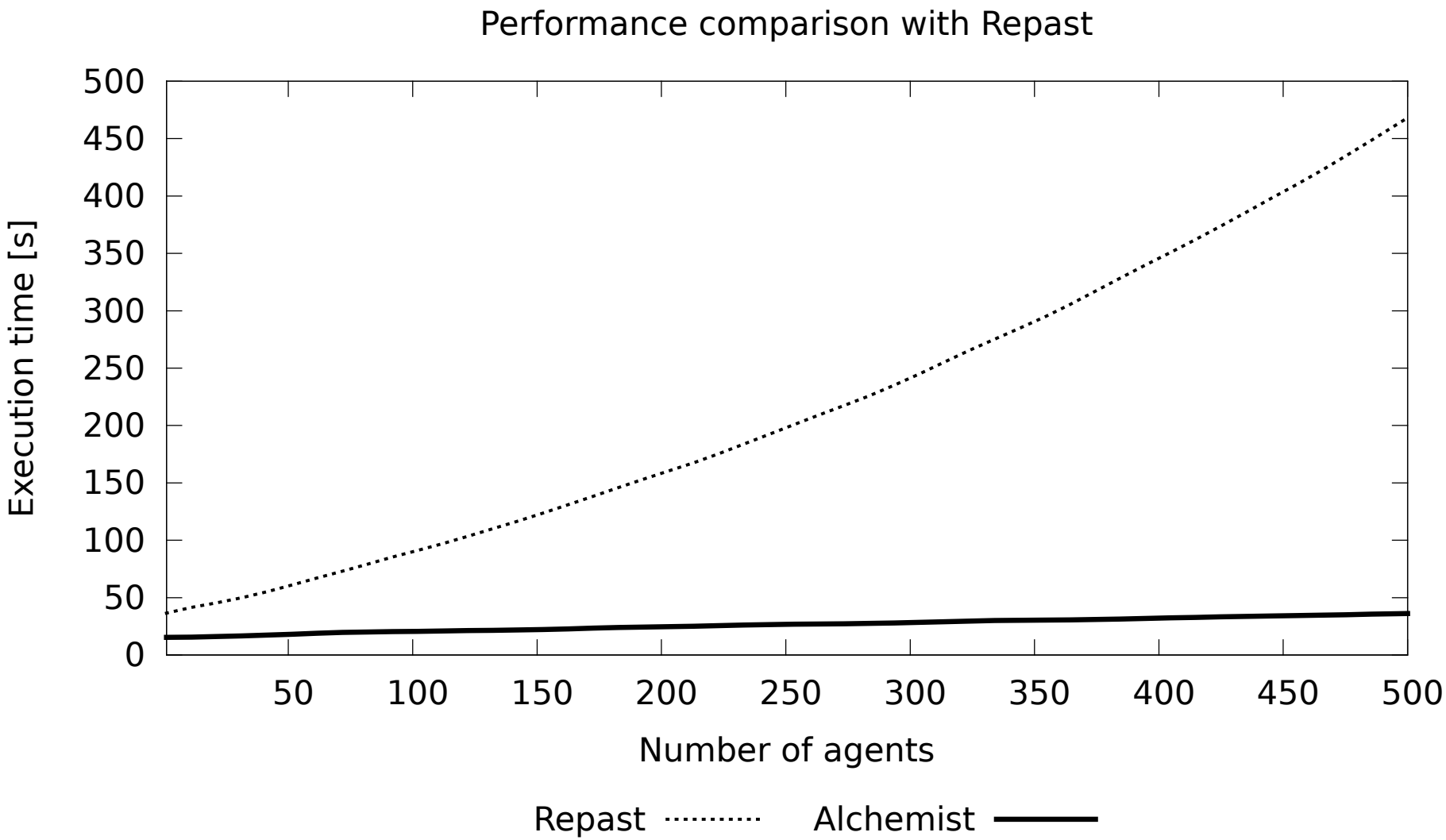
Reactions


Reaction
Conditions
Node contains something
Number of neighbors<3
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

Architecture



Against Repast



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

Writing simulations with the SAPERE incarnation

- Alchemist uses YAML as language for writing simulations
  - JSON superset
  - Compact
  - Human-readable
  - Supports anchoring (referencing)
- The same syntax can be used for any incarnation
- Support for running batches
- No intermediate compilation
- No large files involved

Minimal specification

```
# We are using SAPERE as incarnation
incarnation: sapere
```

Node displacement and connection

```
incarnation: sapere

network-model: # Network type description
  type: EuclideanDistance # Search a class implementing LinkingRule with this
   ↪  name
  parameters: [5] # Use a constructor that can take a single integer number.
   ↪  Deduce other parameters (if any) from the context.

displacements: # Description of where the nodes should be
  - in:
      type: Point # Search a class implementing Displacement with this name
      parameters: [0, 0] # As above: use a constructor requiring only
       ↪  parameters deducible from context and two integers
  - in:
      type: Point
      parameters: [0, 1]
```

Multiple nodes

```
incarnation: sapere

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

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

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, 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]
```

Initial node content

```
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: # A description of what will be included in the node
      - molecule: hello # Everywhere
      - in: # Restrict the area to...
          type: Rectangle # ...a class implementing Shape with this name...
          parameters: [-1, -1, 2, 2] # ...which can get built with these
           ↪  parameters
        molecule: token # Molecule to inject
```

Programming 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]
    contents:
      - in:
          type: Rectangle
          parameters: [-0.5, -0.5, 1, 1]
        molecule: token
    programs: # A list of the sets of reactions programming the node
      -
        - time-distribution: 1 # Frequency. If the class is not specified, the implementation to use is
          ↪    chosen by the incarnation. The SAPERE incarnation automatically loads ExponentialTime,
          ↪    which takes a number representing the Markovian rate.
        # program lets the incarnation choose the class implementing Reaction, and passes down a string
          ↪    that, when parsed, produces the program
          program: > # ">" begins a multiline string (quote mode)
            {token} --> {firing}
        # If the time distribution is unspecified, the SAPERE incarnation assumes a "ASAP" behavior (rate
          ↪    = Infinity)
        - program: "{firing} --> +{token}"
```

Code reuse in YAML

```yaml
incarnation: sapere

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

send: &send # Create an anchor
  - 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 # Reference the anchor
```

Diffusion

```
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} --> {source} {gradient, 0}"
  - time-distribution: 1
    program: "{gradient, N} --> {gradient, N} *{gradient, N+#D}"
  - program: >
      {gradient, N}{gradient, def: N2>=N} --> {gradient, N}
  - time-distribution: 0.1
    program: >
      {gradient, N} --> {gradient, N + 1}
  - program: >
      {gradient, def: N > 30} -->

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

Personalised time distribution

```
incarnation: sapere

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

send: &grad
  - time-distribution:
      type: DiracComb
      parameters: [0.5]
    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: [-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`, `Incarnation`, `RandomGenerator`, `Node`, `Reaction`, `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
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, parameters: [*sourceStart, *sourceStart, *sourceSize, *sourceSize] }
        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)

Movement

(Complete version on the provided code)

```
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} --> {token, N, L} *{token, N+#D, L add [#NODE;]}"
  - program: >
      {token, N, L}{token, def: N2>=N, L2} --> {token, N, L}
# Age information
  - time-distribution:
      type: DiracComb
      parameters: [0.1]
```

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

Bibliography I


Banks, J., Carson, J., Nelson, B., and Nicol, D. (2010).
Discrete-event system simulation.
Prentice Hall, 5th ed. edition.


Gibson, M. A. and Bruck, J. (2000).
Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels.
The Journal of Physical Chemistry A, 104(9):1876–1889.


Gillespie, D. T. (1977).
Exact stochastic simulation of coupled chemical reactions.
Journal of Physical Chemistry, 81(25):2340–2361.


Slepoy, A., Thompson, A. P., and Plimpton, S. J. (2008).
A constant-time kinetic monte carlo algorithm for simulation of large biochemical reaction networks.
The Journal of Chemical Physics, 128(20):205101+.

Bibliography II


Wooldridge, M. J. and Jennings, N. R. (1995).
Intelligent agents: Theory and practice.
Knowledge Engineering Review, 10(2):115–152.