MyCellS design
Version 1.0

September 8, 2017

Contents

1. Introduction **4**

2. Design principles **4**

3. Global presentation of the components of the simulator **6**
3.1. Components of the simulator 6
3.2. Reactant hierarchy 7
3.2.1. Reactant 8
3.2.2. Chemical 8
3.2.3. FreeChemical 8
3.2.4. BoundChemical 8
3.2.5. ChemicalSequence 9
3.2.6. DoubleStrand 10
3.2.7. BindingSiteFamily 11
3.3. Reaction hierarchy 11
3.3.1. Reaction 12
3.3.2. ChemicalReaction 12
3.3.3. SequenceBinding 13
3.3.4. Translocation 14
3.3.5. Loading 15
3.3.6. DoubleStrandRecruitment 17
3.3.7. Release 17
3.3.8. Degradation 19
3.4. Switches 20
3.5. Solver loop 20
3.6. Events 21
3.7. Input/Output handling 22
3.7.1. Simulator Input 22
3.7.2. Simulator Output 22

4. Detailed design **23**
4.1. Reactants 23
4.1.1. `FreeChemical` 23
4.1.2. `BoundChemical` 23
4.1.3. `ChemicalSequence` 24
4.1.4. `DoubleStrand` 25
4.1.5. `BindingSiteFamily` 25
4.2. Reactions 26
4.2.1. `ChemicalReaction` 26
4.2.2. `SequenceBinding` 26
4.2.3. `Translocation` 27
4.2.4. `Loading` 27

4.2.5. `Release` . . . 28
4.3. Solver loop . . . 28
4.3.1. `RateContainer` classes . . . 28
4.3.2. `RateManager` classes . . . 28
4.3.3. `Solver` classes . . . 29
4.4. Input/Output handling . . . 29
4.4.1. Parsing system . . . 29
4.4.2. Builder and interpreter . . . 30
4.4.3. Output . . . 30

5. Implementations of Gillespie algorithm **30**
5.1. Selecting reaction to perform . . . 31
5.1.1. Direct method . . . 32
5.1.2. Next reaction method . . . 34
5.1.3. Binary tree . . . 34
5.1.4. Hybrid method . . . 35
5.1.5. Summary . . . 38
5.2. Propensity updating . . . 41
5.2.1. Naive method: why not update everything? . . . 41
5.2.2. Using graphs to update propensities . . . 42
5.2.3. Pros and cons of graph algorithms . . . 43

A. Utility classes **47**
A.1. Exceptions . . . 47
A.2. Random handler . . . 47
A.3. Factories . . . 47
A.4. Vector-based containers . . . 48
A.5. Handler classes for memory handling . . . 49

B. Tests **49**
B.1. Testing philosophy . . . 49
B.2. Organizing and running tests . . . 51

C. Formats and Conventions **51**
C.1. Input format description . . . 51
C.2. UML . . . 52

1. Introduction

The aim of this document is to present the general design of MyCellS's implementation (the code is documented using Doxygen, technical details are therefore best found in the Doxygen-generated manual).

Before we started working on MyCellS, we set up a list of requirements that the simulator should fulfill. We wanted to be able to integrate MyCellS in a whole-cell framework applicable to several organisms. Such a simulator should meet the following requirements.

- Stochastic simulation of standard bacterial processes, in particular polymerization processes.
- Easy to extend and reuse (either by extending the core or coupling with external modules, in particular deterministic simulation modules).
- Efficient (not more than a couple of hours for one cell cycle).
- Generic formalism, applicable to any bacteria. Once a model has been built for some species, it should be easy to adapt it for another species by modifying input files only.
- Easy to use, inputs should be close to classical biochemical descriptions.
- Integrate various levels of description. The user should be able to focus on a process of interest with a very low level of description while keeping the remaining processes at a higher level.

We start by listing the central design choices of MyCellS. Then we present the base components of the simulator, organized around reactants and reactions. The rest of the document is intended for persons who are interested in implementation details. Section 4 describes all the components of the simulator again, but goes further into hypotheses and critical design elements. Section 5 provides detailed information on the drawing algorithms used in MyCellS. Finally, appendices are added to describe elements that have been important in the simulator development but did not fit naturally in the main document (testing strategies, utility classes, etc.).

2. Design principles

Stochastic simulation In order to meet the requirements listed above, we opted for a Gillespie-based simulator. The Gillespie algorithm has two important features in our context:

- It is a stochastic algorithm, so it naturally enables to simulate low-level stochastic processes.
- It offers a framework where an arbitrary number of reactions can be added.

Using the Gillespie algorithm, we can both simulate events at the molecular level and aggregated processes. The description level of a process simply depends on the number of reactions that the user has chosen to represent the process. MyCellS starts with an empty system of reactions. The user controls what reactions to add. Processes can easily be tuned to match a specific bacterial species.

Sequence-based reactions Standard Gillespie simulators only implement chemical reactions. This does not meet our requirement of simulating a wide variety of processes. For example, it is extremely tedious (nearly impossible) to simulate translation accurately using only chemical reactions. Take a simple molecule of species A translocating along a sequence S of length 100. Here are three possible models for these translocation events:

Model A	Model B	MyCellS
$S + A \rightarrow SA_1$	$S + A \rightarrow S + A_1^{bound}$	
$SA_1 \rightarrow SA_2$	$A_1^{bound} \rightarrow A_2^{bound}$	$S + A \rightarrow S + A^{bound}$ (Binding)
$SA_2 \rightarrow SA_3$	$A_2^{bound} \rightarrow A_3^{bound}$	$A_i^{bound} \rightarrow A_{i+1}^{bound}$ (Translocation)
$\dots$	$\dots$	
$SA_{99} \rightarrow SA_{100}$	$A_{99}^{bound} \rightarrow A_{100}^{bound}$	

Subscripts show the position of A along the sequence. Model A and B use only chemical reactions, both involve duplicating translocation events. In practice, this implies creating thousands of reactions and chemical species. In model A, the sequence S can only bind one chemical at a time, which is not realistic for biological systems (there is a high number of ribosomes per mRNA for example). Model B enables several molecules to "bind" the sequence at a given time. However, bound molecules do not interact with each other, *e.g.* it is impossible to handle sequestration of binding motifs. In MyCellS, we define new types of reactants and reactions for sequence-based events. A single reaction and a single chemical species are used to represent all translocation events.

We created a variant of the Gillespie algorithm where new types of reactants and reactions can be plugged in. We defined a minimal set of reactants and reactions that handles sequence-based reactions (*e.g.* binding, translation elongation). All reactions remain low-level, enabling flexible descriptions of complex processes using a limited number of reactions. A lot of information that is provided as an input for these reactions comes from standard sequence annotation. When switching from an organism to another, the key reactions that define the process remain the same. Only sequence information (DNA, position of genes, promoters, etc.) and rates need to be adapted.

Efficiency We evaluated performance by simulating protein production. Protein production (from gene to protein) is responsible for a large number of reactions in a bacterial cell (metabolism aside). Simulation is completed within hours even for detailed descriptions of all processes involved (stochastic base-by-base elongation with all cofactors).

This objective was reached by using the latest implementations of the (exact) Gillespie algorithm. We also tuned all new types of reactions to be nearly as efficient as chemical reactions.

Modularity We created clear modules in MyCellS's structure. This allows for core changes and facilitates communication with external modules. Typical core changes involve:

- Plugging in new solver variants (*e.g.* new implementation of the exact Gillespie algorithm, implementation of approximations such as τ-leaping).
- Plugging in new reactants and reactions.

Interfaces of the modules were designed for these operations to be pure plug-in operations (no need to change the code in existing modules). A similar design applies for external programs. Reactant concentrations can be modified during the course of a simulation. This enables to plug-in arbitrary external solvers. For example, we intend to plug-in a deterministic solver for metabolism on MyCellS.

3. Global presentation of the components of the simulator

3.1. Components of the simulator

The simulator can be decomposed into several large modules that handle specific tasks during simulation (Fig. 1). First of all, there is an **input/output** module that creates everything that is needed for the simulation from an input file. **Reactants** and **reactions** are user-specified and need to be created on demand, as well as **events** happening throughout the simulations and more technical aspects about which algorithm to use to perform the integration. Once everything is set up, the **solver** follows a simple loop that can be decomposed in three steps. Integration occurs reaction by reaction, at each loop, we go forward one reaction, update the simulation time, concentrations and reaction rates.

1. At the beginning of the loop, the **input/output** process checks whether **events** should occur at the current simulation time and whether it needs to write some concentrations to an output file.
2. It then hands control over to the **solver**, which is based on Gillespie's approach to integrate a network of chemical reactions. The Gillespie algorithm needs the current reaction rates of all **reactions** and draws a random reaction with a probability proportional to its rate. This task is delegated to a **rate manager**, which uses state-of-the-art methods to maintain the rate list updated and perform the drawing efficiently.

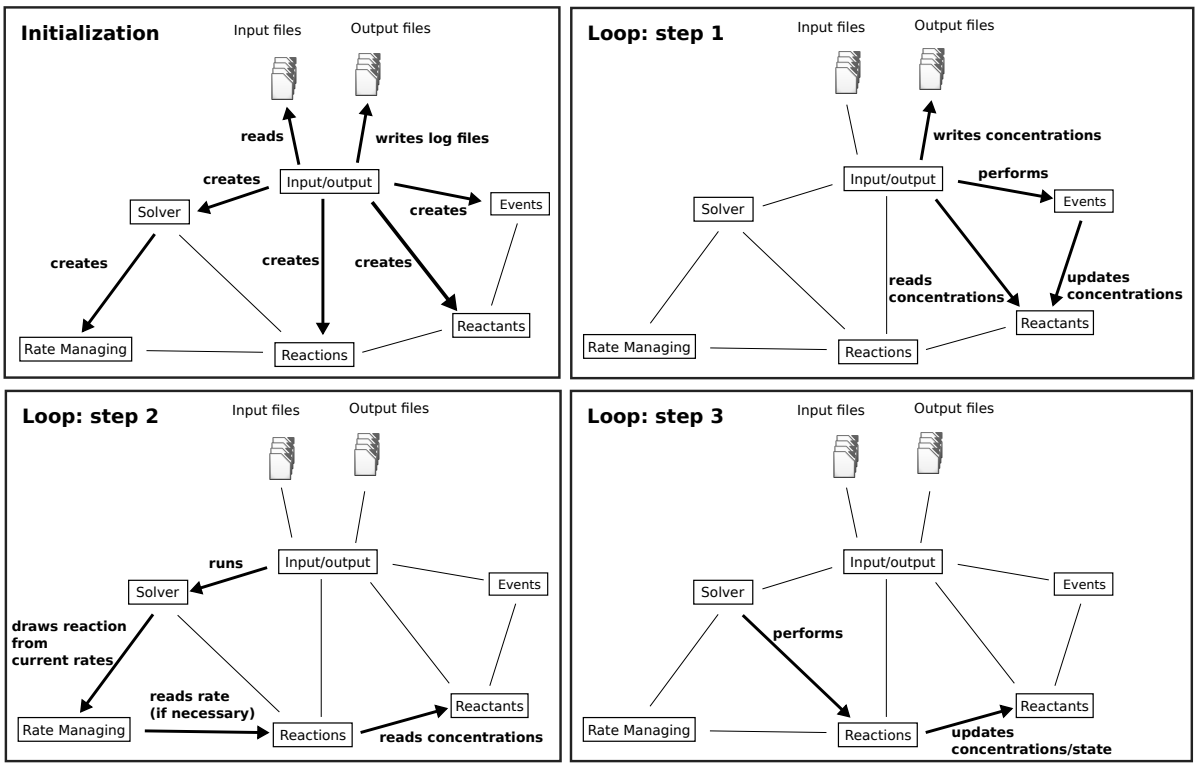


Figure 1: Schematical view of the simulator.

3. Once a **reaction** is drawn, it is performed. The concentrations (and the state, see below) of its **reactants** is modified.

3.2. Reactant hierarchy

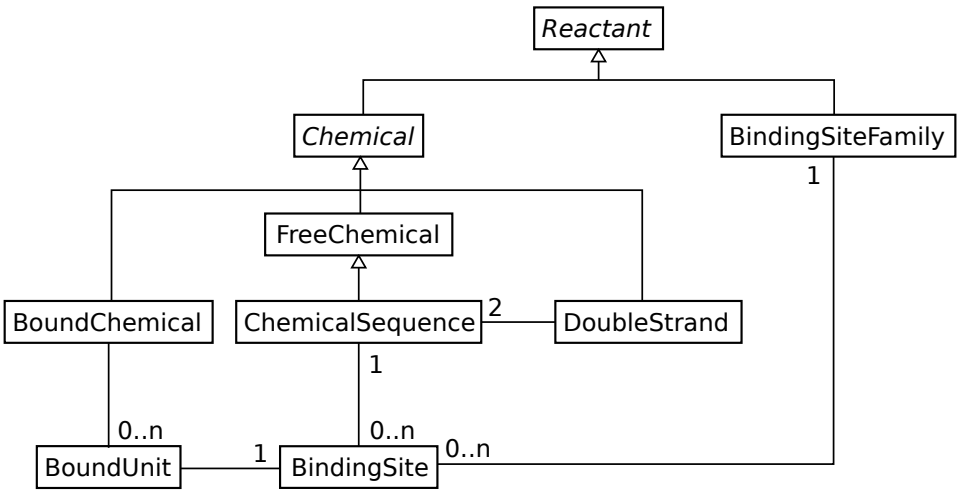


Figure 2: UML diagram of `Reactant` hierarchy

This section gives a quick overview of the contents of the `Reactant` hierarchy (Fig. 2). More details about how reactants are implemented can be found later.

3.2.1. Reactant

`Reactant` is a global abstract interface. All entities that can participate in a reaction *must* inherit from it.

3.2.2. Chemical

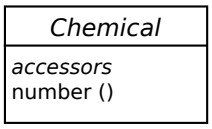


Figure 3: `Chemical` class

`Chemical` is an abstract class (Fig. 3). It defines all standard chemical entities. `Chemical` represents a *pool* of a given chemical species, meaning that one may access its current number at any time.

3.2.3. FreeChemical

Input format

```
FreeChemical <name> [<initial quantity>]
```

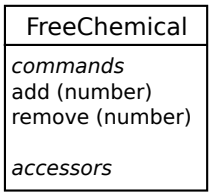


Figure 4: `FreeChemical` class

`FreeChemical` (Fig. 4) is a subclass of `Chemical` that represents free chemical (*e.g.* molecules diffusing in the cytosol or extracellular medium).

3.2.4. BoundChemical

Input format

```
BoundChemical <name>
```

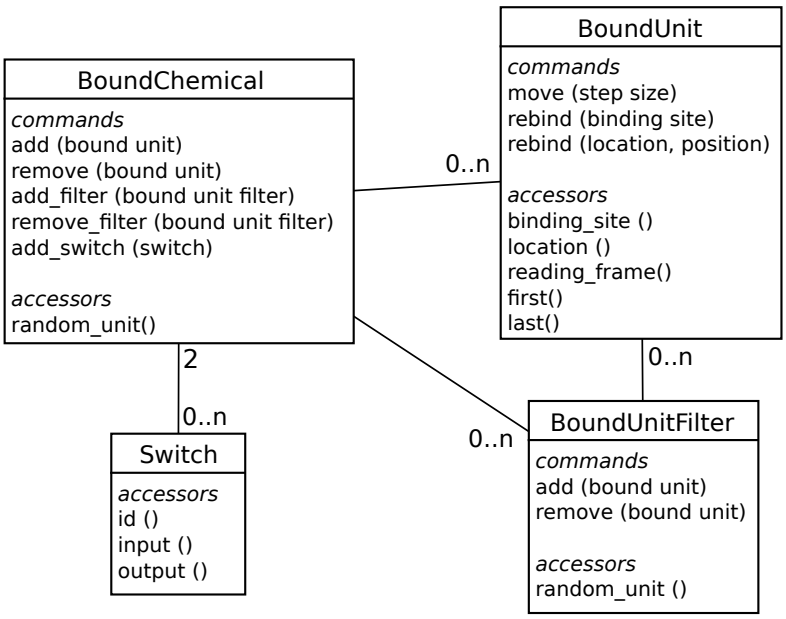


Figure 5: `BoundChemical` class

`BoundChemical` (Fig. 5) is a subclass of `Chemical` that represents chemicals that are bound to a sequence. Even though `BoundChemical` represents a pool of molecules, single elements are not interchangeable, they are defined by their position on a sequence. `BoundChemical` uses class `BoundUnit` to represent molecules individually. It uses `BoundUnitFilter` to organize bound units according to outside criteria needed for reactions (classify according to binding sites, motifs read, etc.). It also uses `Switch`es on specific switch sites that are sequence dependent (this will be explained in detail later).

For example, a RNA polymerase (RNAP) bound to DNA is a `BoundChemical`. A typical instance would be a RNAP bound to DNA on position 1000, another could be bound to position 2000. A `BoundUnitFilter` can be used to sort RNAPs according to the base they are trying to load (A, C, G or T). Finally, a `Switch` would be used to indicate termination sites.

3.2.5. ChemicalSequence

Input format

```
ChemicalSequence <name> sequence <sequence> [<initial quantity>]
TransformationTable <name> [<parent_letter> <product_letter>,]^{1..n}
ProductTable <name> <transformation table>
ChemicalSequence <name> product_of <parent sequence> \
  <starting position> <ending position> <product table> [<initial quantity>]
```

`ChemicalSequence` (Fig. 6) is a subclass of `FreeChemical`. It is defined by a sequence and the ability to bind elements. However, instances of a sequence are *not* treated individually, it is impossible to tell to which instance a given chemical bound. An

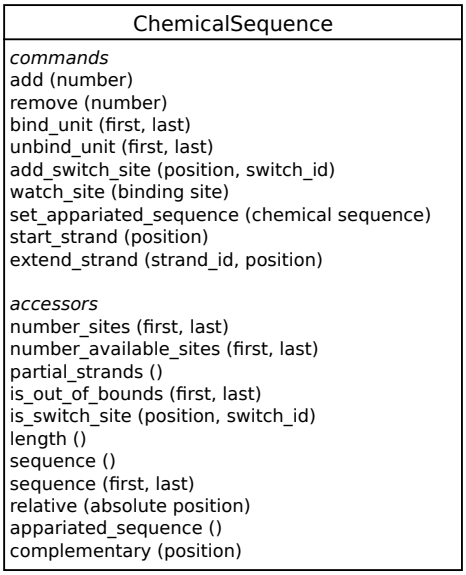


Figure 6: `ChemicalSequence` class

object called `SequenceOccupation` maintains occupation levels at sites of interest. For example, suppose the sequence is an mRNA carrying a ribosome binding site for the protein DnaA. The number of available sites is obtained by removing the number of bound chemicals occupying the site from the number of instances of the mRNA currently in the cell. A `ChemicalSequence` can be coupled to another `ChemicalSequence`. A `ChemicalSequence` can be created from a sequence or as a product of another sequence, in which case a `TransformationTable` is needed to generate the product's sequence from the parent's, and a `ProductTable` stores the parent/product relationship.

3.2.6. DoubleStrand

Input format

```
TransformationTable <name> [<letter> <complementary_letter>,]^{1..n}
DoubleStrandSequence <name> <name_sense_sequence> <sense_sequence> \
  <name_antisense_sequence> <transformation_table> [<initial quantity>]
```

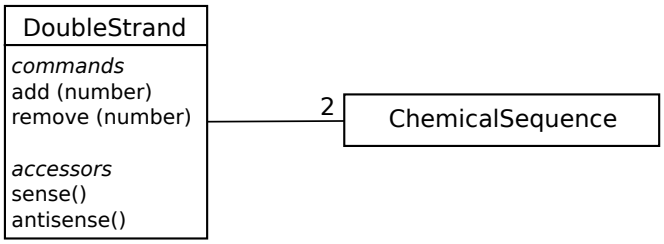


Figure 7: `DoubleStrand` class

`DoubleStrand` (Fig. 7) links two `ChemicalSequence` together that are biochemically linked (*e.g.* DNA), one sequence being complementary to the other. It enables segment extension on the complementary strand and free end binding (see interface of `ChemicalSequence`). A `DoubleStrand` is created from a sense sequence that is specified similarly to a `ChemicalSequence`. However, the complementary sequence is created from a `TransformationTable` that specifies how to transform the sense sequence into antisense sequence (*e.g.* for DNA, $A \to T$, $T \to A$, $C \to G$, $G \to C$).

3.2.7. BindingSiteFamily

Input format

```
BindingSite <binding site family name> <chemical sequence> \
  <start> <end> <k_on> <k_off> [<reading frame>]
```

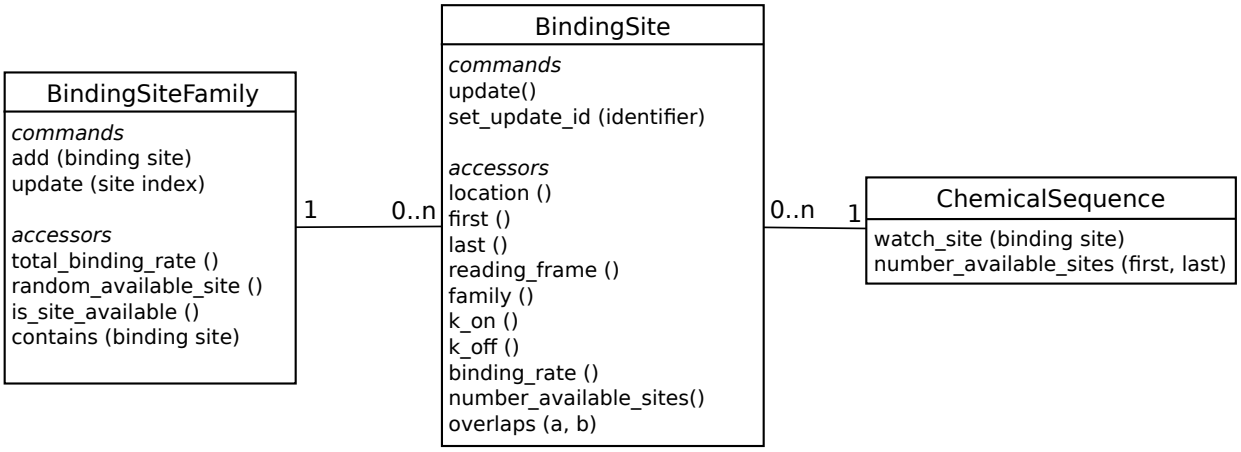


Figure 8: `BindingSiteFamily` class

`BindingSiteFamily` (Fig. 8) is a subclass of `Reactant`. Contrary to `Chemical`, it does not represent a countable pool of molecules. Each family contains a number of related instances of `BindingSite` (*e.g.* ribosome binding sites). `BindingSiteFamily`, `BindingSite` and `ChemicalSequence` use a notification pattern (via `update` methods) to dynamically maintain the number of available sites for each binding site as well as binding rates up to date. If a binding site is used to load polymerases, a reading frame should be provided to specify where a polymerase will start reading the sequence after binding.

3.3. Reaction hierarchy

This section gives a quick overview of the reaction hierarchy (Fig. 9). More details about how reactions are implemented can be found later.

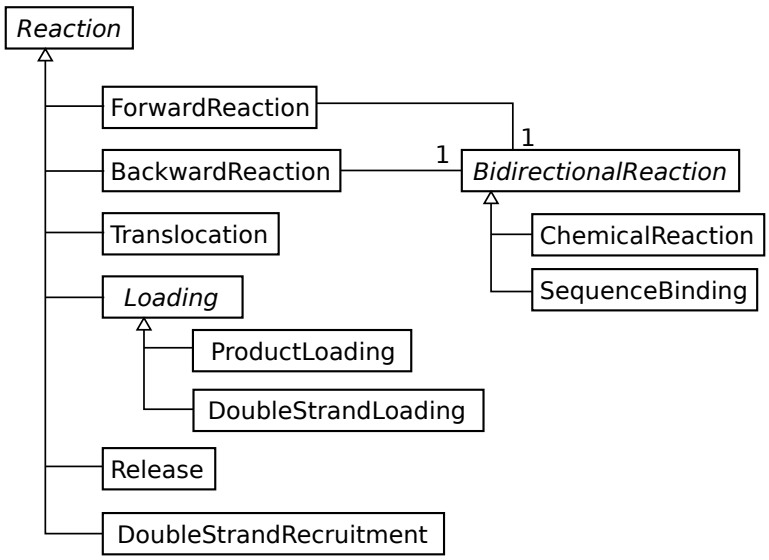


Figure 9: UML diagram of `Reaction` hierarchy.

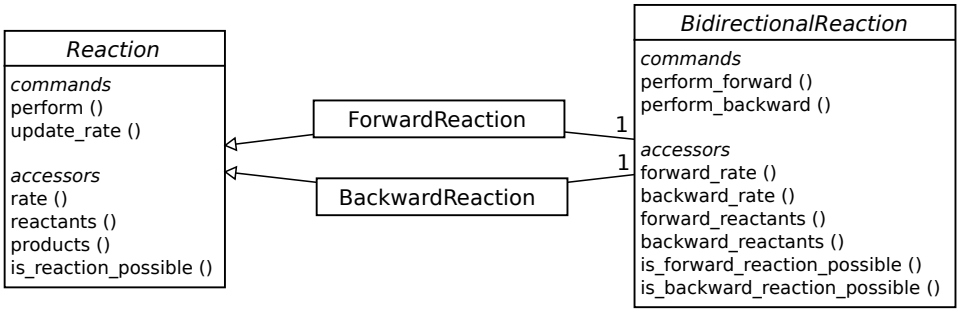


Figure 10: `Reaction` and `BidirectionalReaction` classes.

3.3.1. Reaction

There are two abstract classes used to define reactions: `Reaction` for one-way reactions and `BidirectionalReaction` for reversible reactions. Two adapter classes `ForwardReaction` and `BackwardReaction` split reversible reactions in two one-way reactions(Fig. 10). In the end, the solver only handles one-way reactions. A reaction can necessarily be performed, its rate updated and accessed and is composed of reactants and products.

3.3.2. ChemicalReaction

Input format

```
ChemicalReaction [<chemical> <stoichiometry>]^{1..n} rates <k_1> <k_-1>
```

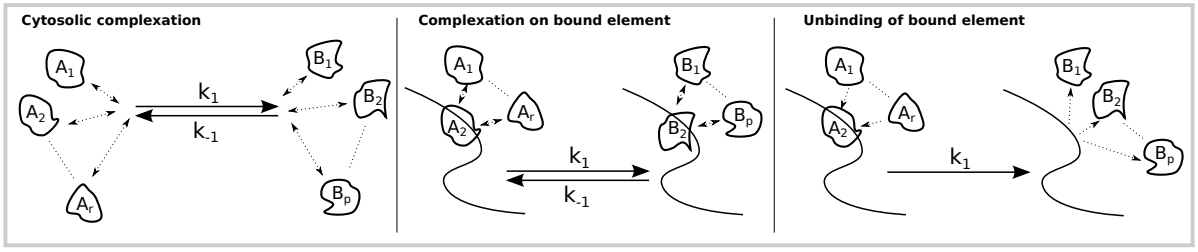


Figure 11: Schematic view of a `ChemicalReaction`.

Formula A `ChemicalReaction` represents association/dissociation of an arbitrary number of elements (Fig. 11). It is defined by

$$a_1 A_1 + a_2 A_2 + \cdots + a_r A_r \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} b_1 B_1 + \cdots + b_p B_p$$

where

- A_i and B_i are of type `FreeChemical`. They can be of type `BoundChemical` in two cases: (i) a reaction containing a `BoundChemical` on each side, (ii) an *irreversible* reaction where a *reactant* is a `BoundChemical` and where there are no bound product. In both cases, the associated stoichiometric coefficient must be 1.
- a_i and b_i are stoichiometric coefficients.
- k_1 and k_{-1} are rate constants.

Action When the reaction is performed, the number of chemicals involved is changed according to their stoichiometric coefficient. If `BoundChemical` are involved on each side, the simulator will assume that the bound chemical that is consumed is replaced by the bound chemical on the other side of the equation (*i.e.* it will be bound at the location previously occupied by the precursor). If there is a `BoundChemical` on the reactant side of an irreversible reaction, the simulator will assume that the reaction describes the unbinding of this bound unit into the cytosol.

Rate The rates are given by

$$\lambda_{forward} = k_1 \prod_{i=1}^{r} [A_i]^{a_i}$$

$$\lambda_{backward} = k_{-1} \prod_{i=1}^{p} [B_i]^{b_i}$$

3.3.3. SequenceBinding

Input format

```
SequenceBinding <chemical> <bound form> <binding site family>
```

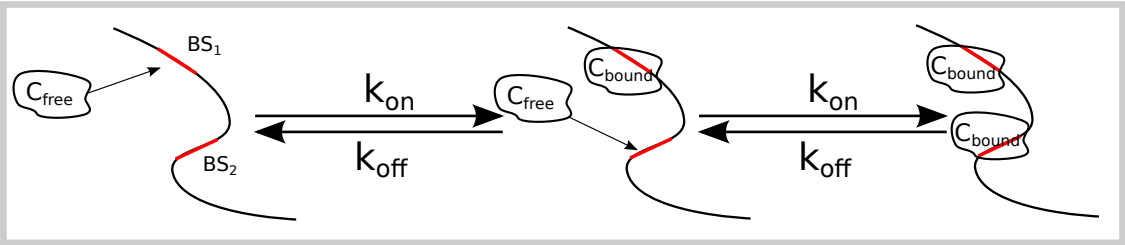


Figure 12: Schematic view of a `SequenceBinding`.

Formula A `SequenceBinding` represents binding of a free element on a binding site of a sequence (Fig. 12). It is defined by

$$C_{free} + BSF \rightleftharpoons C_{bound}$$

where

- C_{free} is of type `FreeChemical`.
- BSF is of type `BindingSiteFamily`.
- C_{bound} is of type `BoundChemical`.

Action When the forward reaction is performed, a random available binding site is drawn from the binding site family (drawing is weighted by affinity). A C_{free} molecule is removed from the pool and a C_{bound} added to the `ChemicalSequence` bearing the binding site. When the backward reaction is performed, a random molecule of C_{bound} is removed from the pool (and from its sequence) and a C_{free} molecule is added.

Rate The rates are given by

$$\lambda_{forward} = \frac{[C_{free}]}{V_c} \sum_{\text{sites } s \in BSF} (k_{on})_s \times \text{Number of sites } s \text{ available}$$

$$\lambda_{backward} = \frac{1}{V_c} \sum_{\text{molecules } m \in C_{bound}} (k_{off})_{\text{site on which } m \text{ is bound}}$$

- $(k_{on})_s$ is the association constant of C_{free} with binding site s.
- $(k_{off})_s$ is the dissociation constant of C_{bound} with binding site s.
- V_c is the volume of the cell.

3.3.4. Translocation

Input format

```
TerminationSite <family name> <chemical sequence> <start> <end>
Translocation <bound chemical> <form after step> <stalled form> \
  <step size> <rate>
```

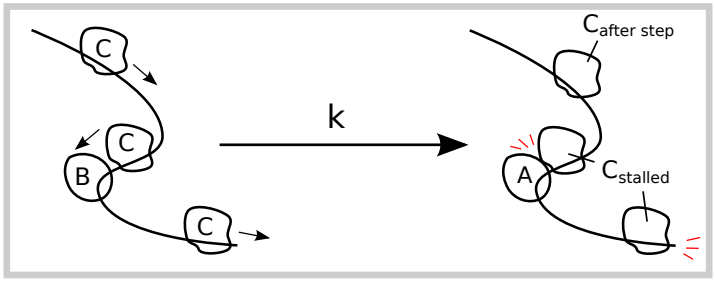


Figure 13: Schematic view of a `Translocation`.

Formula A `Translocation` represents movement of a bound element along a sequence (Fig. 13). It is defined by

$$C \xrightarrow{k} C_{\text{after step}}$$

or

$$C \xrightarrow{k} C_{\text{stalled form}}$$

where

- C is of type `BoundChemical`.
- $C_{\text{after step}}$ is of type `BoundChemical`.
- $C_{\text{stalled form}}$ is of type `BoundChemical`.
- k is a rate constant.

Action When the reaction is performed, a random C is chosen. Generally, it is replaced by a $C_{\text{after step}}$, moved by a step of a given size along the sequence the original C is bound to. If the chemical cannot move because it reached the end of the sequence, it is replaced by $C_{\text{stalled form}}$.

Rate The rate is given by

$$\lambda = k[C]$$

3.3.5. Loading

Input format

```
LoadingTable <name> \
  [<template> <element_to_load> <occupied_polymerase> <rate>,]^{1..n}
ProductLoading <bound chemical> <loading table>
DoubleStrandLoading <bound chemical> <loading table> <stalled form>
```

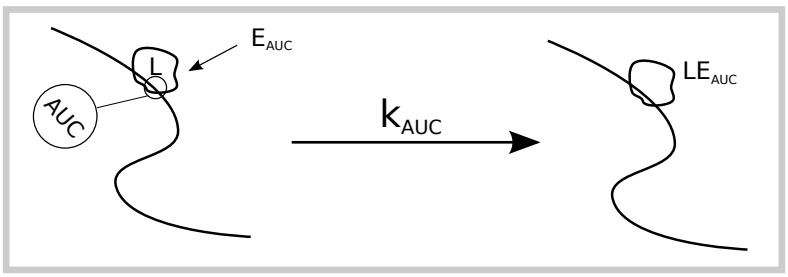


Figure 14: Schematic view of a `Loading`.

Formula A `Loading` typically represents loading of elements by a polymerase onto a template sequence (Fig. 14). It is defined by

$$L + E \longrightarrow LE$$

where

- L is of type `BoundChemical`.
- E is an element to load, of type `FreeChemical`. It is defined in a `LoadingTable` associated with the reaction.
- LE is the occupied form of the loader, of type `BoundChemical`. It is defined in a `LoadingTable` associated with the reaction.

Action Each instance of L reads a specific template. Using its `LoadingTable`, we know which E it tries to load, which LE is yielded if loading occurs and the loading rate associated with the template. When the reaction is performed, a random L is chosen according to loading rates. An element to load E is removed from the pool and L is replaced with LE. A `ProductLoading` assembles loaded elements into a product that will eventually be release in the cytosol (*e.g.* RNA synthesis), while `DoubleStrandLoading` extends segments along a `DoubleStrand` (*e.g.* DNA replication). In `DoubleStrandLoading`, loading may fail because the loader met a previously synthesized segment. In the latter case, it is replaced by a `BoundChemical` representing its stalled form.

Rate The rate is given by

$$\lambda = \sum_{t \in templates} k_t [L_t][E_t]$$

where

- k_t is the loading rate associated with template t.
- L_t corresponds to loaders L reading template t.
- E_t is the chemical to load onto template t.

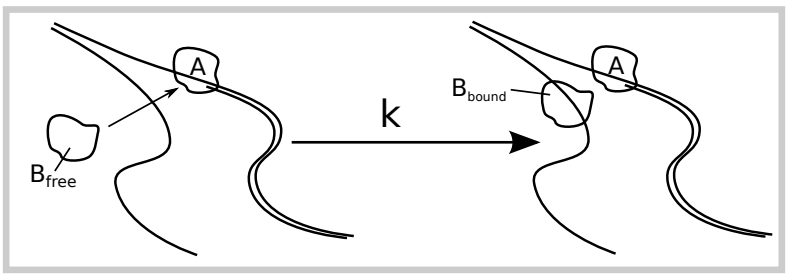


Figure 15: Schematic view of a `DoubleStrandRecruitment`.

3.3.6. DoubleStrandRecruitment

Input format

```
DoubleStrandRecruitment <BoundChemical> <FreeChemical> <bound form> <rate>
```

Formula A `DoubleStrandRecruitment` typically represents recruitment of a DNA polymerase by the replication fork on the opposite strand (Fig. 15). It is defined by

$$A + B_{free} \xrightarrow{k} A + B_{bound}$$

where

- A is of type `BoundChemical`, bound to a `DoubleStrand`.
- B_{free} is of type `FreeChemical`.
- B_{bound} is a `BoundChemical` representing the bound form of B_{free}.
- k is a rate constant.

Action When the reaction is performed, a random A is chosen. If A is not bound to a `DoubleStrand`, the reaction is ignored. If the position opposite to a on the `DoubleStrand` is already occupied, the reaction is ignored. Else, a B_{free} is bound on the complementary `ChemicalSequence`, opposite to A as a B_{bound}.

Rate The rate is given by

$$\lambda = k[A][B_{free}]$$

3.3.7. Release

Input format

```
TransformationTable <name> [<parent_letter> <product_letter>,]^{1..n}
ProductTable <name> <transformation table>
Release <polymerase> <empty_polymerase> <fail_polymerase> \
  <product table> <rate>
```

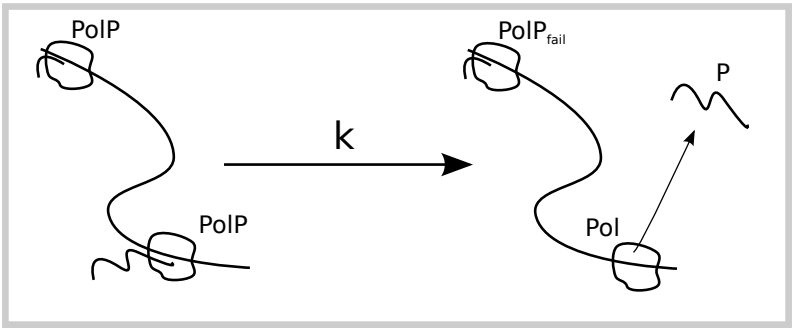


Figure 16: Schematic view of a `Release`.

Formula A `Release` represents release of a product from a polymerase (Fig. 16).

$$PolP \xrightarrow{k} Pol + P$$

or

$$PolP \xrightarrow{k} PolP_{fail}$$

where

- $PolP$ is a `BoundChemical` representing a polymerase-product complex.
- P is of type `ChemicalSequence`. It is a product that is released by $PolP$ defined in a `ProductTable` associated with reaction.
- Pol is a `BoundChemical` representing an empty polymerase.
- $PolP_{fail}$ is a `BoundChemical` representing the polymerase-product complex in case release failed because P was not a valid product defined in the `ProductTable` associated with reaction.
- k is a rate constant.

Action When the reaction is performed, a random $PolP$ is chosen. A `ProductTable` uses its binding and current position to determine what product P it has synthesized. If P is defined in the product table, it is released in the cytosol and $PolP$ is replaced by an empty version of the polymerase Pol. If there is no P corresponding to current $PolP$ position, the simulator assumes that $PolP$ has not reached its actual terminator and it is replaced by $PolP_{fail}$ to enable other treatments (*e.g.* abnormal termination or continuing synthesis).

Rate The rate is given by

$$\lambda = k[PolP]$$

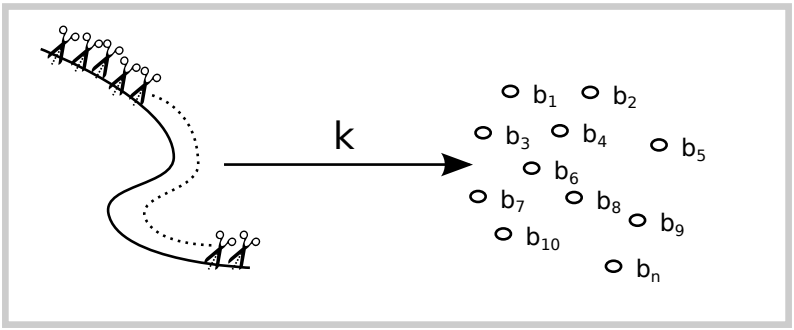


Figure 17: Schematic view of degradation reaction.

3.3.8. Degradation

Input format

```
CompositionTable <name> [<letter> [<chemical composing letter>]^{1..m}]^{1..n}
Degradation <chemical sequence> <composition table> <rate>
```

Formula A `Degradation` represents decomposition of a sequence into base components (Fig. 17). It is defined by

$$CS \xrightarrow{k} b_1 + b_2 + \cdots + b_N$$

where

- CS is of type `ChemicalSequence`.
- b_i are of type `FreeChemical`. They are found in a `CompositionTable` specified in the reaction.
- k is the degradation constant.

Action When the reaction is performed, a CS is removed from the pool. A `CompositionTable` is specified along the reaction. It allows base-by-base conversion of the sequence of CS into components yielded by degradation. The pools of base components is updated accordingly. In the simulator, a degradation reaction is effectively implemented as a `ChemicalReaction`.

Rate The rate is given by

$$\lambda = k[CS]$$

3.4. Switches

Input format

```
Switch <name> <input_bound_chemical> <output_bound_chemical>
SwitchSite <chemical_sequence> <position> <switch_name>
```

Switches are intrinsically linked to BoundChemicals but apply to specific BoundUnits through SwitchSites located on ChemicalSequences. Every time an instance of input_bound_chemica steps on a switch site, it *immediately* becomes an output_bound_chemical.

For example, during transcription, an RNA polymerase (RNAP) goes through an initiation state, then loops through several elongation states (loading of a nucleotide and translocation). Once it reaches a termination site represented by a SwitchSite, the RNAP leaves its current elongation state and enters termination state. It stops performing polymerization reactions and typically releases the polymerization product and unbinds from DNA.

A Switch is not considered a reaction because there is no rate associated with it (switches are performed automatically before the solver chooses the next reaction). We dedicate a section to these elements because they play a central role in the simulator's philosophy. The user can use generic reactions that apply in general (*e.g.* transcription of any gene based on its sequence) and use switches every time something more specific is needed. As seen before, termination sites for transcription are expected to be *SwitchSite*s. Similarly, important regulation sites can be implemented using *SwitchSite*s.

3.5. Solver loop

Once Reactions and Reactants are defined, they must be integrated properly. We use variants of the Gillespie algorithm to provide a framework where reactions are performed according to their current reaction rate. Roughly speaking, the main hypothesis of this framework is that reaction timings are distributed according to exponential distributions. This allows for many mathematical simplifications and harmonious integration of an arbitrary number of reactions. The central point of the algorithm is that the probability that a reaction will be the next reaction in the system is proportional to its rate (mathematically speaking, the reaction is obtained by multinomial drawing according to rates).

The solving loop is depicted in Figure 18. The Gillespie algorithm has many variants. We decided to implement it using three *abstract* classes. By using inheritance, variants can be combined for each step of the algorithm (how to update reactions, how to select a reaction). The three central classes are:

- Solver: Children of this class decide how and when rates should be updated, *e.g.* update rates after every reaction, only after a given time step, etc. Note that they do not perform any of these computations, they just organize how the algorithm should work.

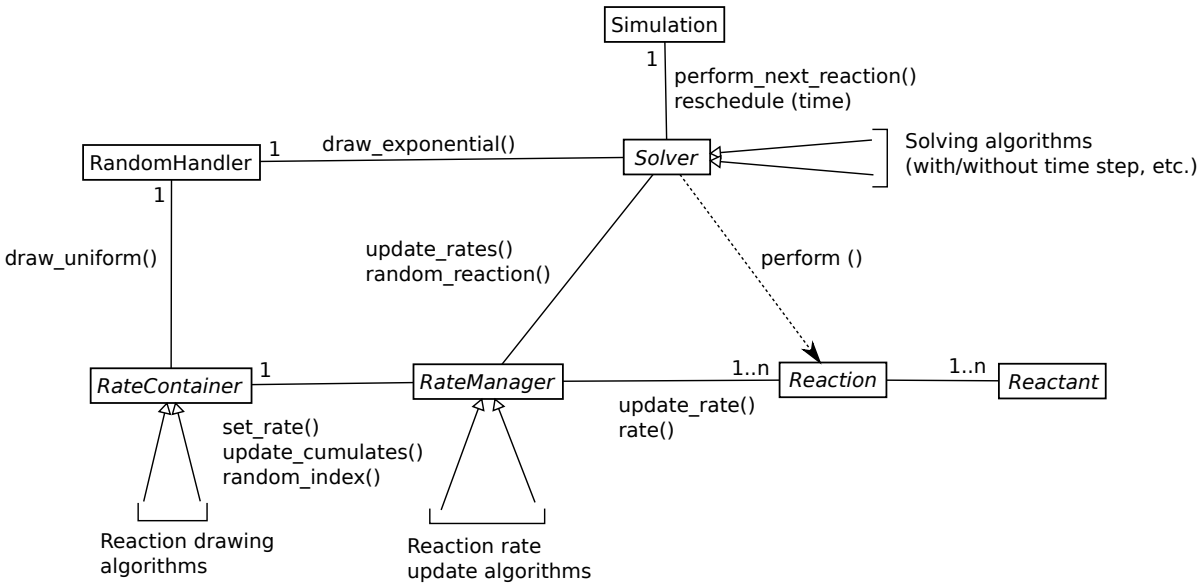


Figure 18: Solver loop. The loop is driven by the `Solver` class that defines how and when rates should be updated. The update task is performed by a `RateManager`. Once rates are known, multinomial drawing is delegated to a `RateContainer`. A central `RandomHandler` is used so that the solver only uses one seed, enabling simulation reproducibility.

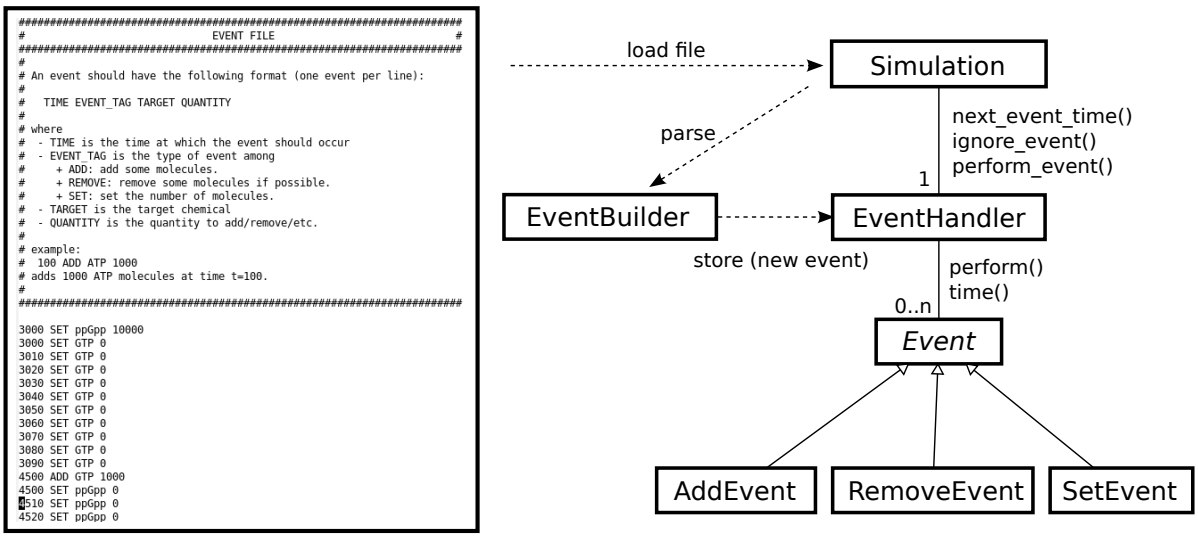
- `RateManager`: Children of this class are responsible for updating reaction rates when prompted to by a `Solver` class. Recomputing all rates is generally inefficient, so various implementations of this task can be used to improve the global loop speed.

- `RateContainer`: Children of this class are responsible for storing reaction rates in a specific structure *adapted* to multinomial drawing. Again many implementations exist, their efficiency depends on the system that is integrated.

The implementations of these three classes will be described later in the document.

3.6. Events

`Event`s enable users to change molecule numbers outside of the solver loop at specific times (Fig. 19). A `Simulation` instance handles both a `Solver` instance and an `EventHandler` instance. Every time an event timing is reached, the solver loop is stopped, the event(s) is (are) performed, the solver is reinitialized and the simulation resumes. Different `Event` implementations are offered to modify molecule numbers in a convenient way.




```
#################################################################
#                          EVENT FILE                           #
#################################################################
#
# An event should have the following format (one event per line):
#
#   TIME EVENT_TAG TARGET QUANTITY
#
# where
#  - TIME is the time at which the event should occur
#  - EVENT_TAG is the type of event among
#      + ADD: add some molecules.
#      + REMOVE: remove some molecules if possible.
#      + SET: set the number of molecules.
#  - TARGET is the target chemical
#  - QUANTITY is the quantity to add/remove/etc.
#
# example:
#  100 ADD ATP 1000
# adds 1000 ATP molecules at time t=100.
#
#################################################################

3000 SET ppGpp 10000
3000 SET GTP 0
3010 SET GTP 0
3020 SET GTP 0
3030 SET GTP 0
3040 SET GTP 0
3050 SET GTP 0
3060 SET GTP 0
3070 SET GTP 0
3080 SET GTP 0
3090 SET GTP 0
4500 ADD GTP 1000
4500 SET ppGpp 0
4510 SET ppGpp 0
4520 SET ppGpp 0
```

Figure 19: `Events`: another way to modify chemical concentrations aside from reactions, *e.g.* to simulate the injection of a chemical inside a cell.

3.7. Input/Output handling

3.7.1. Simulator Input

The simulator needs the following to work:

- A general input file defining simulation parameters. A sample file is provided were all options are described (*e.g.* length of simulation, what to output, algorithm variants). One important parameter is the location of the files the simulator should open to read reactants, reactions and events.
- An arbitrary number of files where reactants, reactions and events are declared. The simulator solves dependencies across files, it is not necessary to declare reactants in the same file as or before reactions using them.

Caution:

- All reactants must be declared in some file with their appropriate type (*e.g.* `FreeChemical` or `BoundChemical`).
- Multiple declarations are forbidden, a name cannot be reused.

3.7.2. Simulator Output

Outputs provided by the simulator are:

- A general output file logging parameters used for simulation (input files used, random seed, algorithms used, etc.).

- A concentration file with the number of molecules over time (for the chemicals and at a time step defined in the parameter file).
- If a `DoubleStrand` was added in the chemicals to output, a replication file describing replication advancement of that `DoubleStrand`.

4. Detailed design

This section is intended for people who are interested in implementation details of the simulator. It shows how elements presented in Section 3 were implemented.

4.1. Reactants

4.1.1. `FreeChemical`

`FreeChemical` simply represents a pool of interchangeable molecules distributed uniformly in the cell. Computationally, only the number of molecules in the pool is relevant.

4.1.2. `BoundChemical`

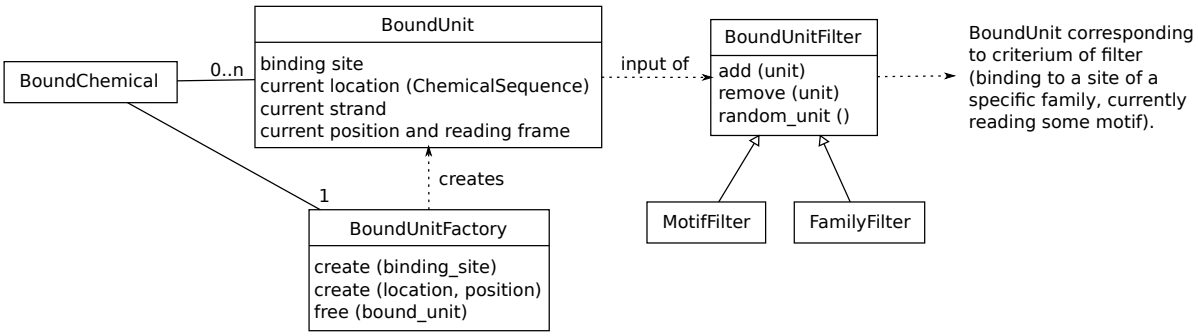


Figure 20: `BoundChemical` are in fact a pool of individual `BoundUnit` created using a `BoundUnitFactory`. A `BoundUnit` is characterized by the `ChemicalSequence` it bound to and its current position. Reaction then use `BoundUnitFilter` to sort `BoundUnit` according to some criterium of reference (*e.g.* `Loading` reactions sort `BoundUnit` according to the motif they read).

`BoundChemical` represents molecules of the same chemical species, but there are specificities for each unit of a `BoundChemical`, as all units are bound at different locations of different `ChemicalSequence` (Fig. 20). A `BoundUnitFactory` is used to recycle `BoundUnit`s, avoiding memory reallocation throughout simulation. `BoundUnitFilters` are used to sort `BoundUnit`s according to criteria useful for reactions (Fig. 20).

`BoundUnit`s are passed from one `BoundChemical` species to another through reactions, their attributes are updated if needed. They are only destroyed once they are unbound from their `ChemicalSequence`.

4.1.3. `ChemicalSequence`

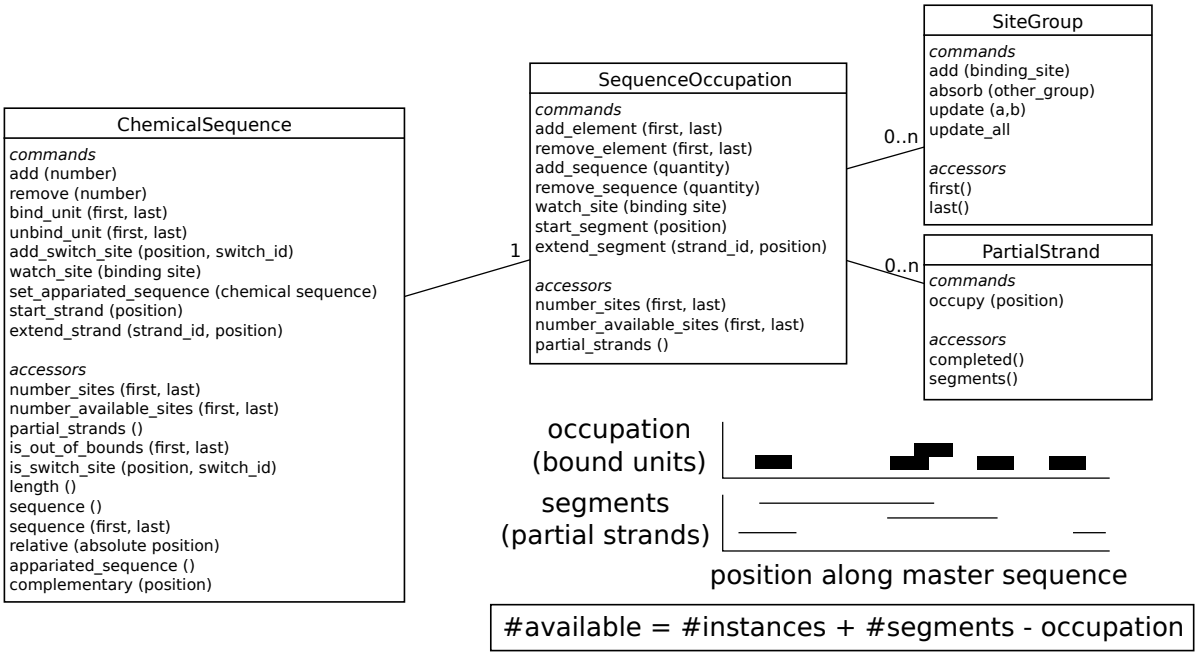


Figure 21: `ChemicalSequence` represents a pool of polymers that can be elongated and on which `BoundUnit`s bind through `BindingSite`s. For binding to occur, availability of `BindingSite` is assessed using a utility class `SequenceOccupation` that records the number of instances of the polymer, the position of `BoundUnit`s and elongation of `PartialStrand`s. `SiteGroup` is used to notify sites of availability changes more efficiently.

`ChemicalSequence` handles a pool of polymers. A pool is defined by a *master sequence* describing what a typical polymer looks like (*e.g.* the sequence of DnaA protein) and the number of *instances* of the master sequence in the pool. For efficiency reason, we do the following assumptions.

Simplifying assumptions

- No deviation from master sequence, all instances are identical.
- `BoundUnit`s are not assigned to a specific instance of the sequence, they are positioned on the master sequence.

Consequences

- No direct inference of collisions is possible.
- A chemical can bind on a partial strand, yet move along the whole sequence freely.
- Degradation of an instance does not cause unbinding.

Site availability Despite our simplifying assumptions it is still possible to provide an accurate description of site availability. Availability depends of the number of sequences, number and position of bound elements, number and position of newly polymerized sequence segments (Fig. 21).

4.1.4. `DoubleStrand`

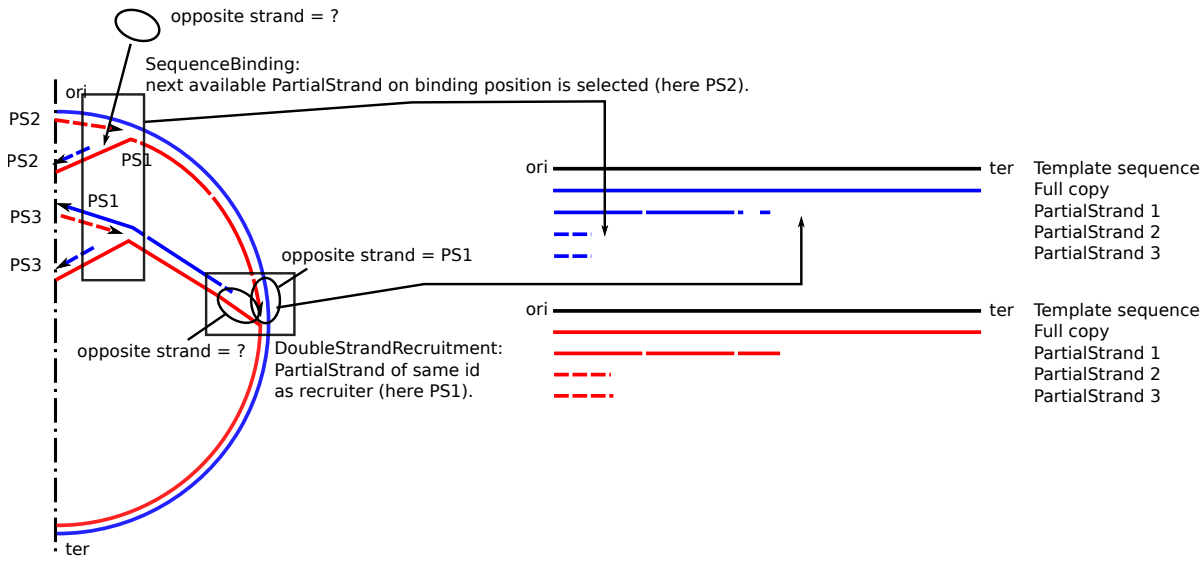


Figure 22: Strands of a `DoubleStrand` are identified according to creation order. Every time a new segment is polymerized, it is necessary to determine which `PartialStrand` is elongated. If a polymerase has been recruited on the complementary strand by `DoubleStrandRecruitment`, it is automatically assigned the same partial strand as the recruiter.

Strand identification Because `DoubleStrand` typically represents DNA, we expect that the `DoubleStrand` will contain a lot of `PartialStrand`s. For replication, it is important to know exactly which strands are opposite to one another for `DoubleStrandRecruitment` to work properly. We use strand identification as shown in Figure 22.

4.1.5. `BindingSiteFamily`

The task of a `BindingSiteFamily` is to regroup all the binding sites that can participate in a same `SequenceBinding` reaction. To simplify the reaction, it stores the subrate associated with each binding site. In order to update the rate properly when availability of sites changes, an *observer pattern* is used (Fig. 23).

Every `BindingSite` is viewed as an *observer* by the `ChemicalSequence` it belongs to. Every time a change occurs on the site, the `BindingSite` is notified. The latter binding site notifies its `BindingSiteFamily` using a specific identifier, letting the family know

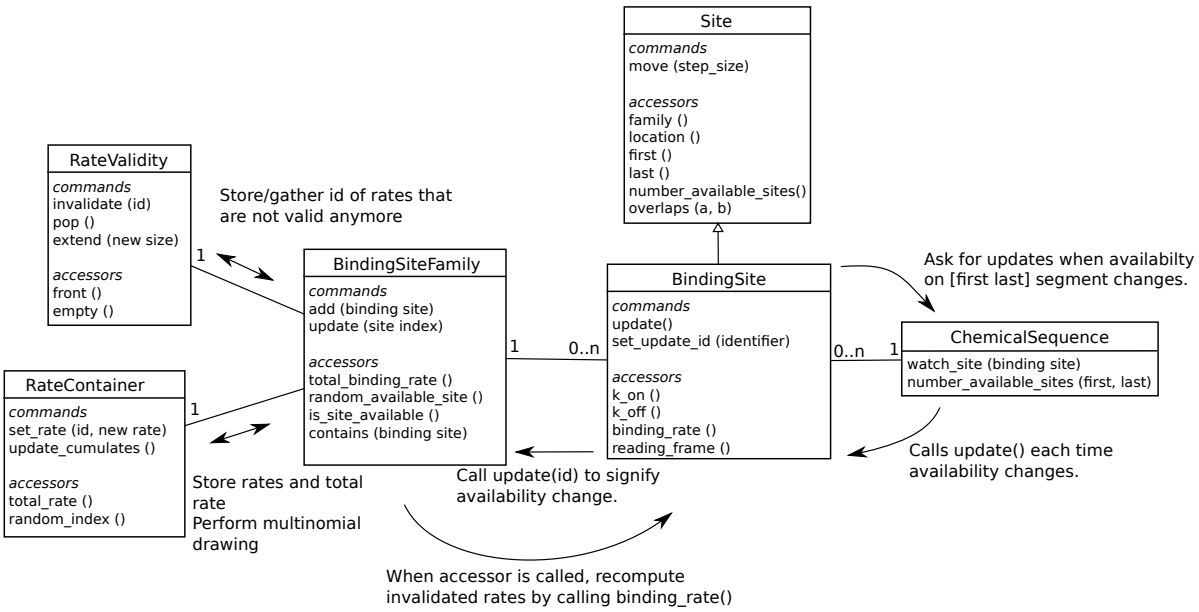


Figure 23: Schematical view of the Observer pattern used to keep availability of binding sites up to date for `SequenceBinding` reactions.

which binding rate is out of date. This information is stored in a `RateValidity` class. It is only when it is really needed (*i.e.* when a `SequenceBinding` wants to access total rate or a random site) that rates are recomputed. This avoids useless computations *e.g.* in the case of a translocation, where a bound unit is first unbound from its `ChemicalSequence` then rebound. If the bound unit does not move away from the site, two updates will be sent, but the rate will only be recomputed once at the end.

4.2. Reactions

4.2.1. `ChemicalReaction`

Nothing particular.

4.2.2. `SequenceBinding`

Binding Because of the way `BindingSiteFamily` is implemented, the reaction can easily and efficiently access the binding rate at all times, no matter what reactions have occured previously and how site availability changed in the meantime.

Unbinding `SequenceBinding` uses a `FamilyFilter` (see detailed description of `BoundChemical`) to filter out all `BoundUnits` that are bound to a binding site of the `BindingSiteFamily` associated with the reaction. `BoundUnits` that have bound to sites of a different family or that have moved away from the binding site through `Translocation` are *not* candidates for unbinding.

4.2.3. `Translocation`

Collisions For now, `Translocation` ignores collisions, making its implementation straightforward.

Stalled form Translocation enters stalled form if a `BoundUnit` reached the end of a sequence.

4.2.4. `Loading`

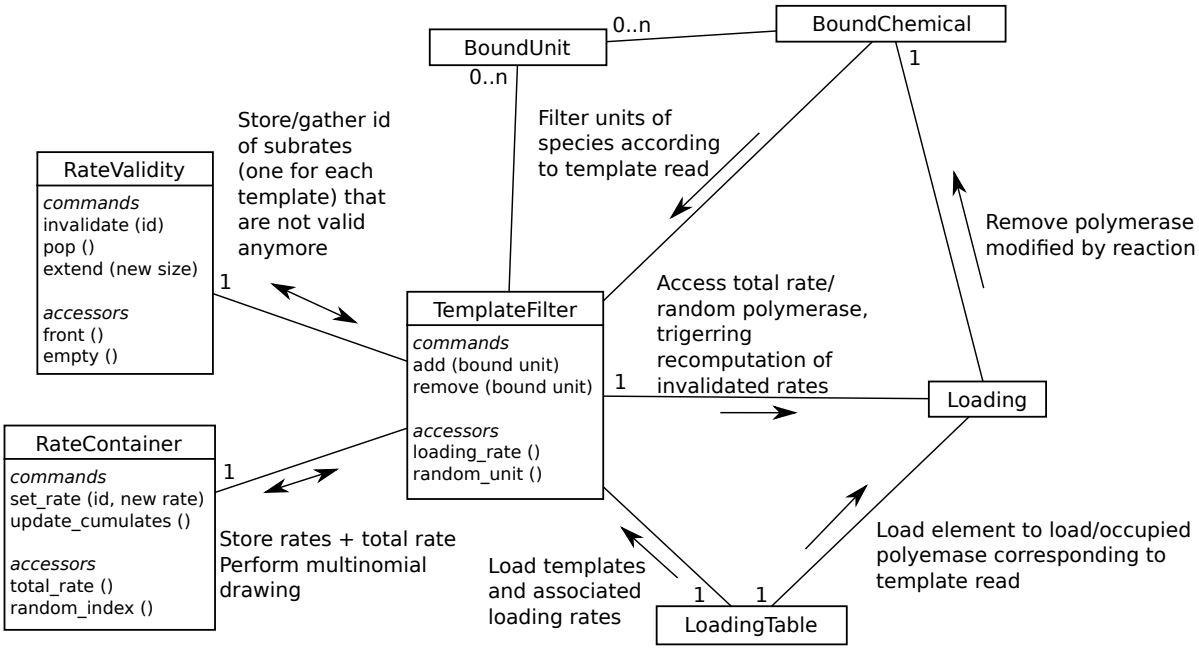


Figure 24: Schematical view of the pattern used to keep subrates associated with each template up to date in a `Loading` reaction.

Handling each polymerase individually The main challenge with `Loading` is to maintain the subrates associated with each motif up to date. It needs to maintain a list of all `BoundUnit`s reading a specific motif. To this end it uses a `TemplateFilter` (see detailed implementation of `BoundChemical`). Every time a `BoundUnit` becomes of the type of the `BoundChemical` associated with the reaction, the filter looks what motif defined in the `LoadingTable` it is currently reading. If the motif could not be found, an `UNKNOWN TEMPLATE` error message is displayed, the `BoundUnit` is not recorded in the filter and will not participate in the `Loading` reaction. The implementation is very similar to that used for `BindingSiteFamily` (Fig. 24).

`ProductLoading` **vs** `DoubleStrandLoading` There difference between the two processes is rather small. We just added a failure condition in the case of `DoubleStrandLoading`

for convenience. Depending on what reactions are used to synthesize a `DoubleStrand` it might be possible that a polymerase arrives upon a position that has already been synthesized. In this case, the `DoubleStrandLoading` fails and the polymerase is replaced by the polymerase in its stalled form.

4.2.5. `Release`

Fail polymerase (unknown product) When a release is triggered, a `BoundUnit` from the `BoundChemical` associated with the `Release` reaction is randomly chosen. Because the `BoundUnit` knows its current position and its binding site, it will assume that product it has synthesized starts the *reading frame of the binding site* and ends *at the position directly preceding its current reading frame* (we assume that the polymerase translocates onto a terminating sequence which does not contribute to product synthesis). If the product is found in the `ProductTable`, everything works normally.

If the product is not found, we display a `Unknown Product` error message but keep the simulation alive. The fail polymerase in the reaction enables the user to define a rescue pathway. If the release competes with some other reaction for the original polymerase, the fail polymerase can be the original polymerase itself. If products overlap and the polymerase was stalled due to a termination site of another product, fail polymerase can be a polymerase in a synthesizing step (*e.g.* `ProductLoading`) so synthesis will resume until the next termination site is reached.

4.3. Solver loop

Here we quickly describe the implementations provided for each step of the algorithm. Most of the details are explained in Section 5.

4.3.1. `RateContainer` classes

We start with the lowest level classes, which perform one of the central tasks of the Gillespie algorithm: drawing a reaction from reaction rates. For efficiency reasons, we propose several implementation of the algorithm (Fig. 25). Comparison and description of theses classes are given in Section 5.

Note that multinomial drawing occurs within the solver loop, but also within some reactions such as `Loading` or `SequenceBinding`, so these classes are used quite extensively throughout the simulation.

4.3.2. `RateManager` classes

The second layer of the solver loop ensures that the rates are updated when needed to. Two implementations are proposed for this task (Fig. 26). The `NaiveRateManager` updates every rate. While it is inefficient, it can be used as a reference to test other managers. The `DependencyRateManager` uses an observer pattern to update only reactions for which a reactant concentration has changed (see Section 5 for further details).

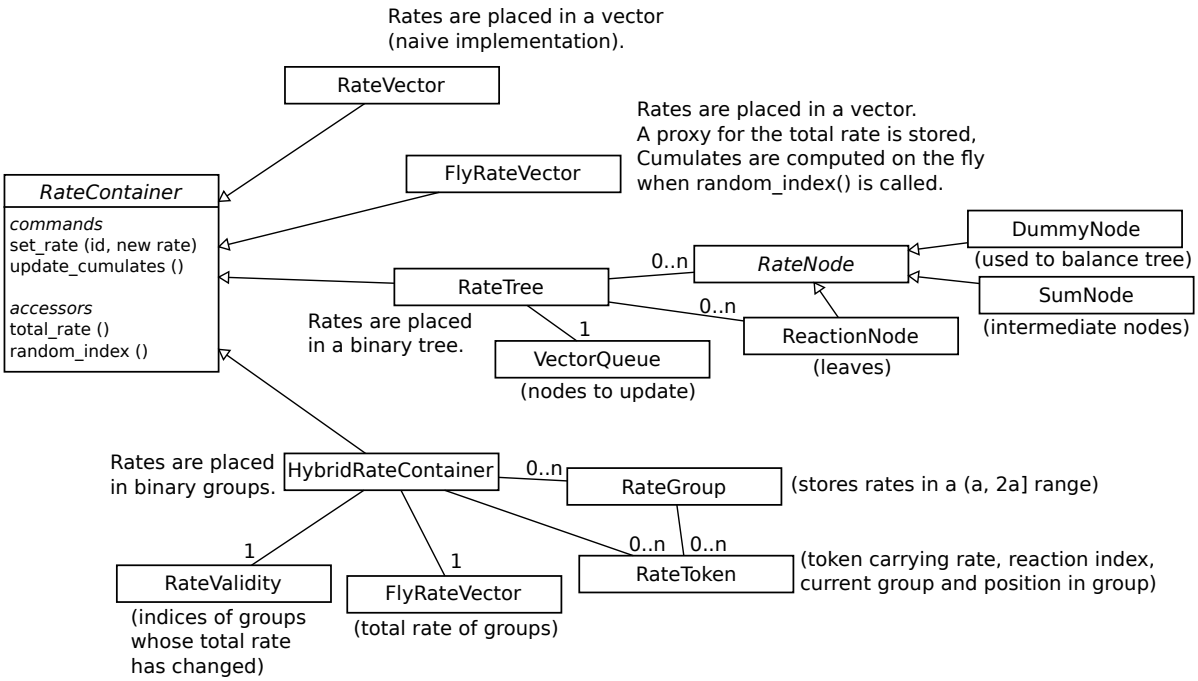


Figure 25: Implementations provided to store rates and perform a multinomial drawing. Implicitly, all theses classes use `RandomHandler` to perform their random drawings.

4.3.3. `Solver` classes

For the moment, only one solver class is fully available to the user, `NaiveSolver`, which implements the exact Gillespie algorithm. Another variant called `ManualDispatchSolver` is implemented, where the user can assign a time step to each reaction at which its rate will be updated (Fig. 27). However, when the rate of a reaction is a constant, there is a risk that its reactants will run out and the reaction will be impossible to realize or reactant number will become negative. In the simulator, the latter case is forbidden, so `ManualDispatchSolver` ignores reactions impossible to perform due to reactant unavailability.

4.4. Input/Output handling

4.4.1. Parsing system

The parsing system used by the simulator is pretty simple (Fig. 28). A `SimulationParams` class is used to store simulation parameters (which will be used to create and drive the `Solver`), `CellState` stores reactions to integrate and `EventHandler` stores events. At the moment, an *ad hoc* input format is used.

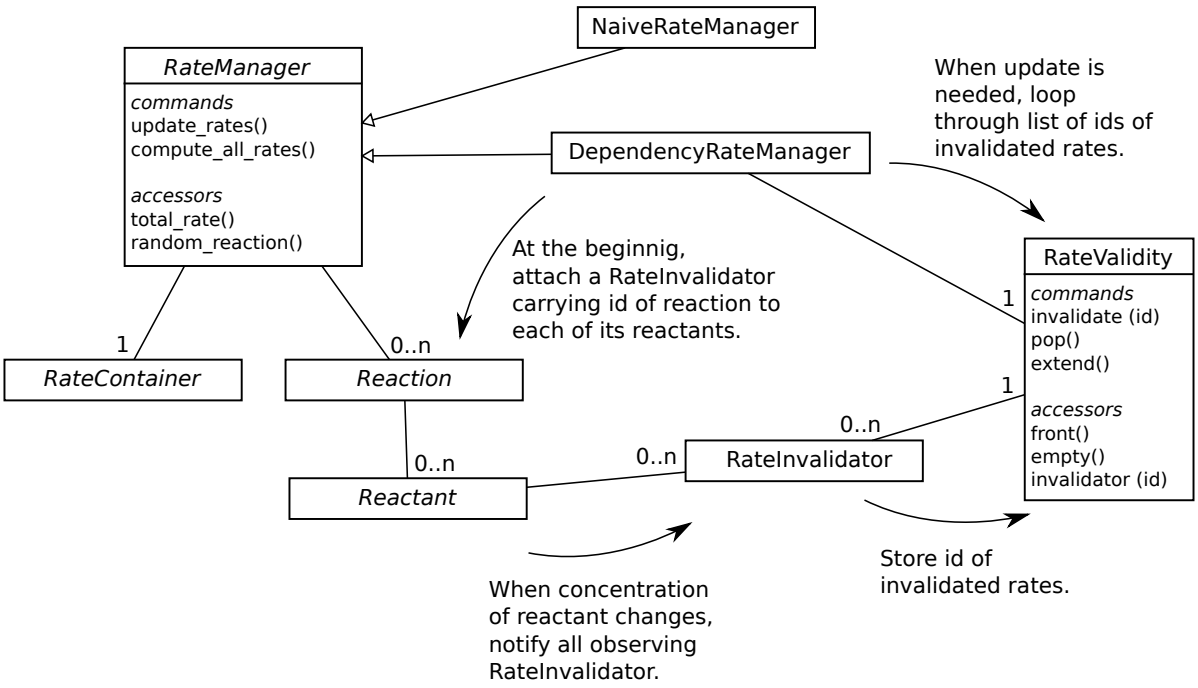


Figure 26: Implementations provided to update reaction rates. Note that the drawing part of the algorithm is always delegated to a `RateContainer`. `DependencyRateManager` uses an Observer pattern to monitor which rates have changed.

4.4.2. Builder and interpreter

The parsing system is designed to cut each line into words, then the words are interpreted by `Builder`s that try to create instances of each of the class of the simulator. The `Parser` loops through the `Builder`s until an instance was successfully created. Line format is checked token-wise by an `Interpreter` (Fig. 29). If no `Builder` is able to match the line, a `FormatException` is raised. If some dependency could not be solved, a `DependencyException` is raised. In the latter case, the `Parser` will postpone the line until dependency can be successfully solved.

4.4.3. Output

Two classes are used to produce output. `ChemicalLogger` logs chemical numbers through time. `DoubleStrandLogger` logs partial strands of a `DoubleStrand`.

5. Implementations of Gillespie algorithm

In this section, we show the principles of the algorithms we implemented. Gillespie et al. (2013) propose a good review of the original Gillespie algorithm and some of its variants.

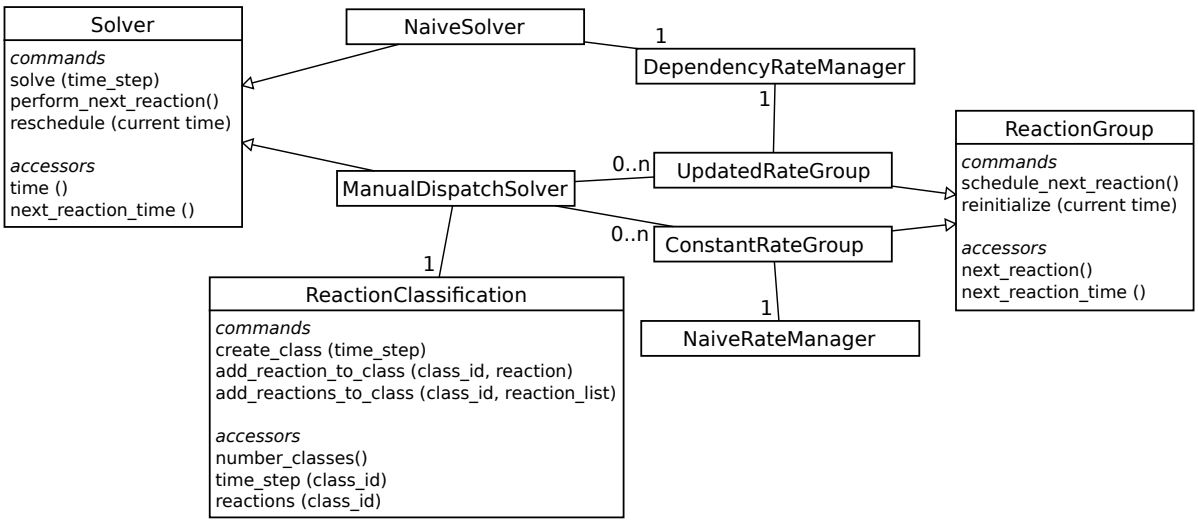


Figure 27: Two implementations of the `Solver` class organizing rate updating. `NaiveSolver` forces recomputation of rates at each time step. `ManualDispatchSolver` puts reactions into groups: reactions in `UpdatedRateGroup` are updated after every reaction while those in `ConstantRateGroup` only at user-defined steps defined in `ReactionClassification`. Note that all `Solver` classes use at least a variant of `RateManager` at some point to delegate storing and updating of rates.

The original Stochastic Simulation Algorithm (original name of the Gillespie algorithm) can be summarized as follows:

- STEP 1: Update rate of reactions.
- STEP 2: Select reaction to perform and next reaction time, perform reaction.

Most of the effort is spent on studying how to optimize the second step, but we will see that the two steps are actually interrelated. In a sense, the SSA and its variants are said to be (statistically) exact.

Our aim was to try various methods to optimize the second step of the SSA for a large biological system. We reviewed the literature and implemented the main alternatives. We provide some perspectives that could be worked on to improve the algorithms or the way they interact with the updating step.

5.1. Selecting reaction to perform

We start with the second step of the SSA, selecting a reaction given some propensities. Most algorithms presented here are reviewed in Gillespie et al. (2013).

Formally, the problem is quite simple: we are given a set of N real values $\{r_1, r_2, \ldots, r_N\}$ and we draw index i with probability $p_i = r_i / \sum r_i$. Mathematically, this is a multinomial distribution and thus could be drawn as such by any standard random number library.

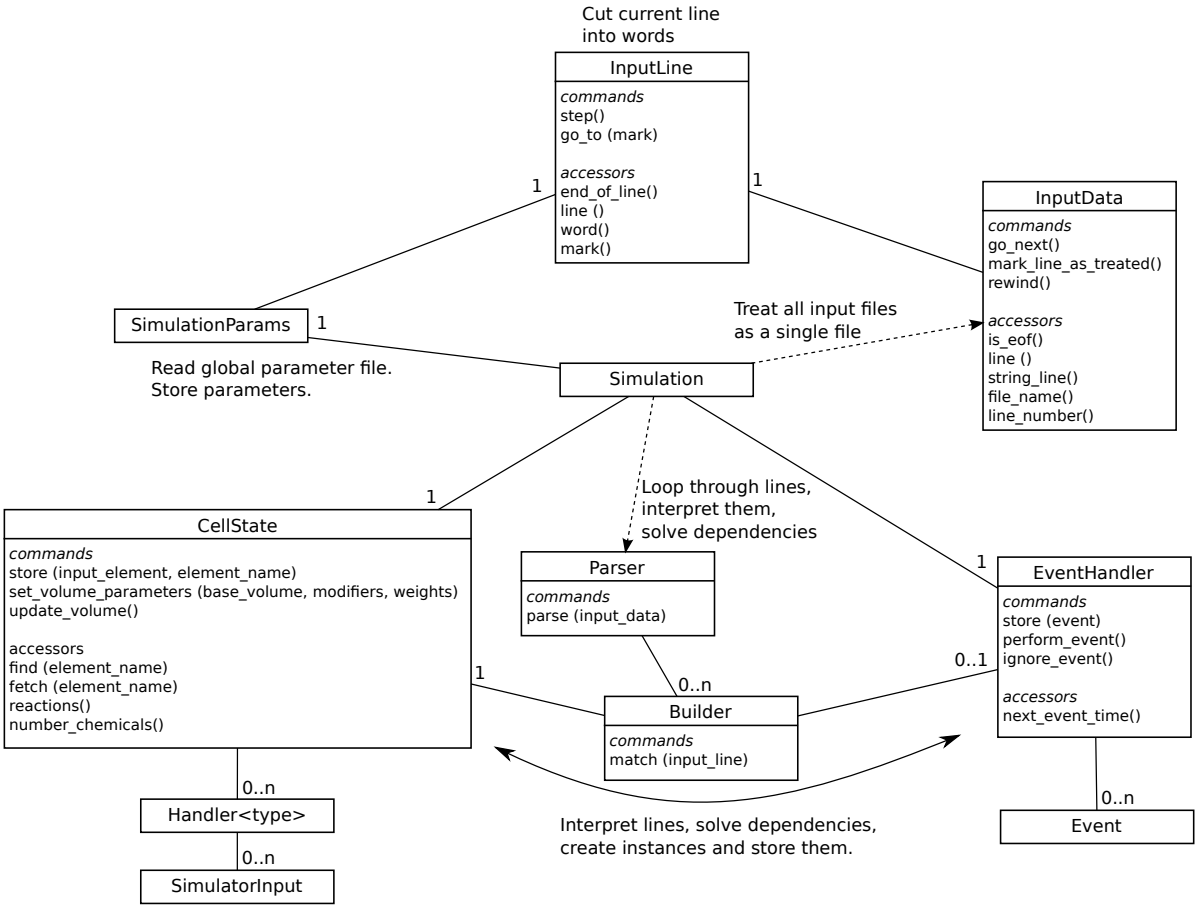


Figure 28: Architecture of the parsing system. `SimulationParams` reads the global parameter file and stores parameter values. `InputData` is used to provide a simplified interface to the input files and mark treated lines. A `Parser` and `Builders` are used to make sense of individual lines. Everything that is created is stored in `EventHandler` (for events) and `CellState` (for the rest).

We will start by stating a standard technique to perform multinomial drawings. We will then introduce methods that increase the speed of the drawing by using *a priori* knowledge.

5.1.1. Direct method

Principle The first method that was used historically is straightforward and sometimes referred to as *biased wheel*. Schematically speaking, you could imagine a wheel similar to "wheel of fortune", except the size allowed to each index on the wheel is proportional to its propensity value, so that large value have a larger probability to be drawn when the wheel is spinned (Fig. 30).

Generally the wheel is seen as a segment subdivided into N subsegments of length r_1, $\ldots$, r_N. A value u is drawn on this $[0, \sum r_i)$ segment. We proceed iteratively to find to

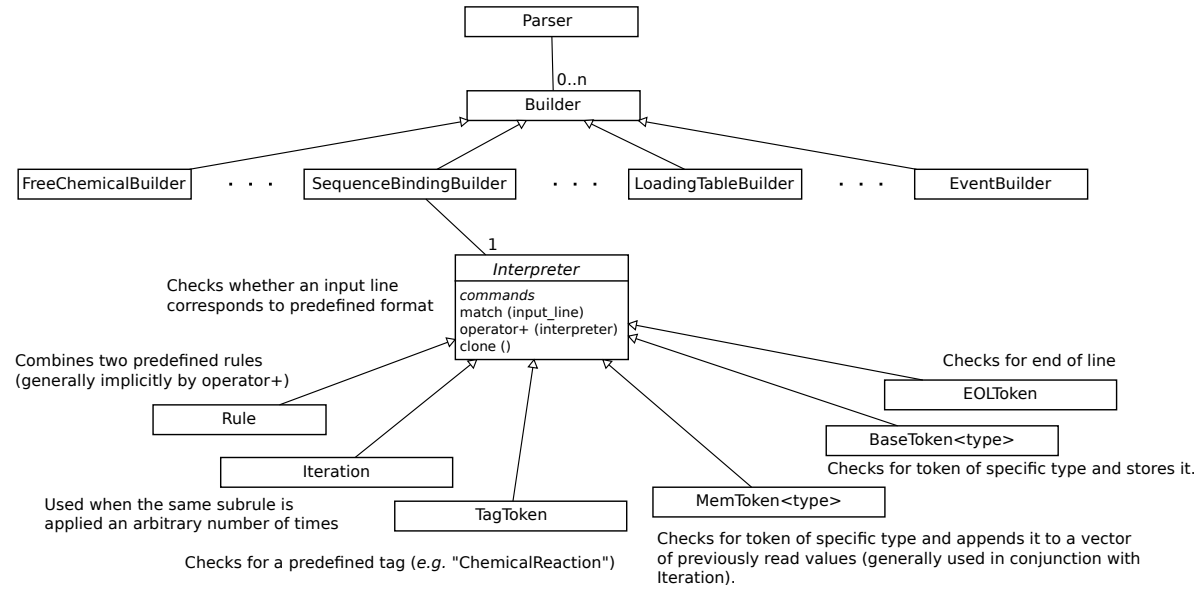


Figure 29: Interpreter system used. For each class of the simulator, a `Builder` is responsible for interpreting current line, solving dependencies, checking validity of parameters. If format is invalid or dependencies could not be resolved, exceptions are raised to warn the user.

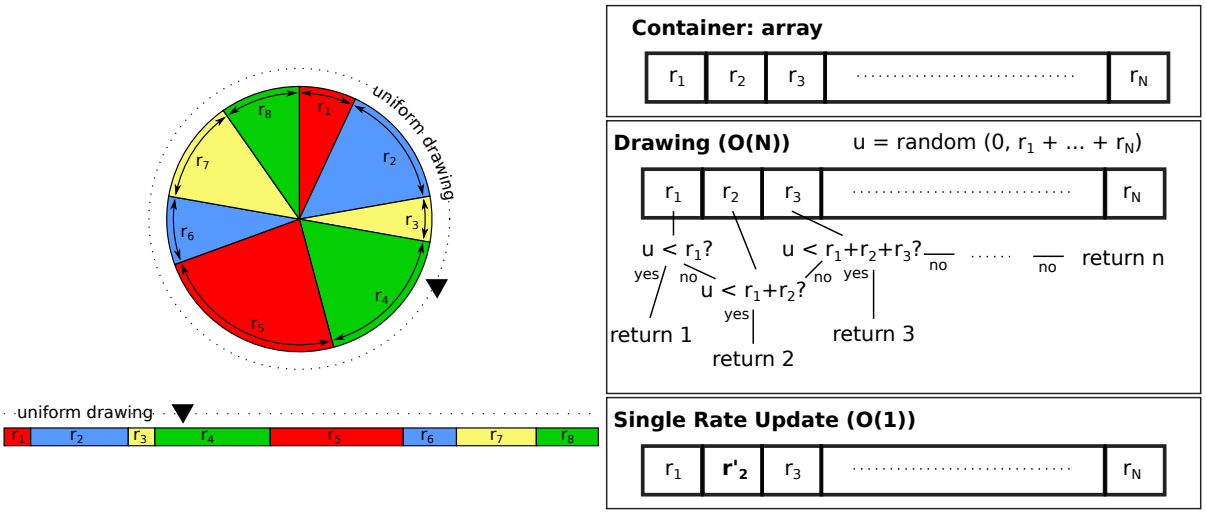
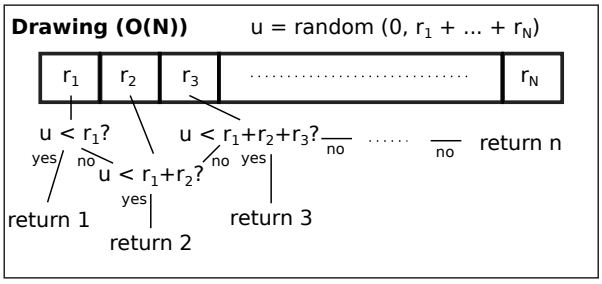


Figure 30: (Left) Illustration of drawing along a biased wheel and its equivalent representation as a segment. (Right) Container and algorithms used to maintain the structure.

which subsegment u belongs. If $u < r_1$, it belongs to subsegment 1. If $r_1 \leq u < r_1 + r_2$, it belongs to subsegment 2, etc.



Data: Array r of size N, R = sum (r).
Result: Index drawn according to multinomial drawing.

```
u = uniform ([0, R));
index = 1;
cum_sum = r[1];
while u ≥ cum_sum do
    index = index + 1;
    cum_sum = cum_sum + r [index];
end
return index
```

Figure 31: Direct drawing method

Sketch of algorithm and complexity Worst case of the drawing (Fig. 31) occurs when u is in the last subsegment, so the loop has to be iterated N times, yielding $O(N)$ complexity.

5.1.2. Next reaction method

The next reaction method is based on the direct method. It is used in systems where it is known that propensities are not uniform. The point is to accelerate the direct method by sorting (at least roughly) propensities within the vector. This does not change drawing statistics but accelerates finding where a drawing is located on the wheel. For example, say a propensity takes up 50% of total propensity and is located at the beginning of the vector. Then each random drawing $u < 0.5 \sum r_i$ will be instantly found to fall into the first sector of the biased wheel. In a non-sorted vector, we would have to loop through several small sectors before finding the big one.

Because the next reaction method is only a twist of the direct method, we do not go into further details now. We will also omit it in complexity analysis, as it has the same worst-case complexity as the direct method. It was not implemented in MyCellS.

5.1.3. Binary tree

Principle In this approach, we organize propensities inside a tree. Propensities are placed in the leaves of the tree. Nodes are then assembled iteratively 2 by 2 to compute the sum of all propensities (Fig. 32).

The idea is that with the structure *in place*, finding the index that has been drawn is quicker. Similarly to the standard drawing, a value u is drawn on the $[0, R = \sum r_i)$ segment. We start from the root node and need to find on which side of the tree u lies. The two children nodes summarize how much weight there is on each side of the tree, say w_{left} and w_{right} respectively. If $u < w_{\text{left}}$, we descend to the left child node and proceed

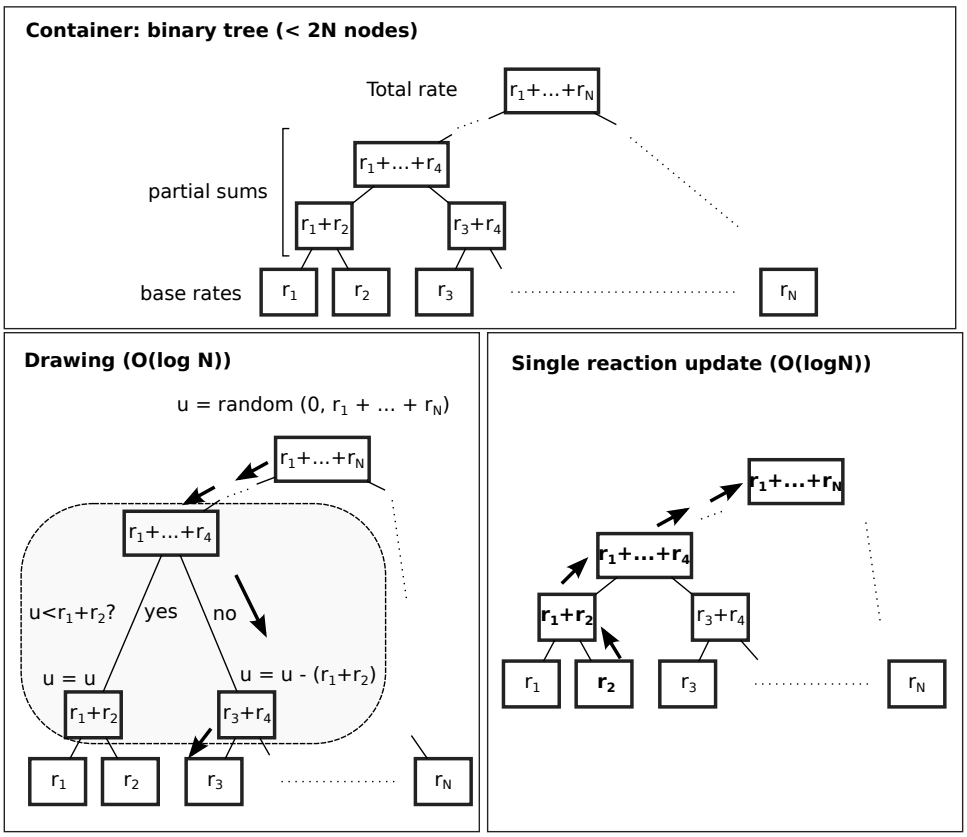


Figure 32: Binary tree containing propensities. Propensity values are found at the leaves of the tree. Intermediate nodes represent partial sums of nodes below, root holds the sum of all propensities.

the same way until we reach a leaf. If $u \geq w_{\text{left}}$, we descend to the right child and we proceed iteratively with $u = u - w_{\text{left}}$.

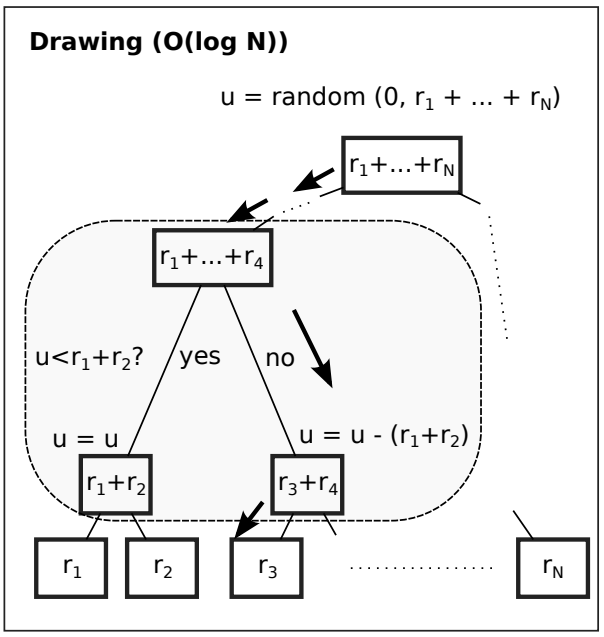
This procedure is actually very similar to the biased wheel method, except we perform some kind of progressive zooming in on the subsegments delimited by the propensity values (Fig. 32).

Sketch of algorithm and complexity Complexity for performing the drawing (Fig. 33) is given by the depth of the tree, $\lceil \log_2 N \rceil$, which is $O(\log N)$.

Complexity for updating the tree (Fig. 34) is also given by depth of the tree, $O(\log N)$. Note that we need not update every node in the tree, only the parents of the updated leaf up to the root node.

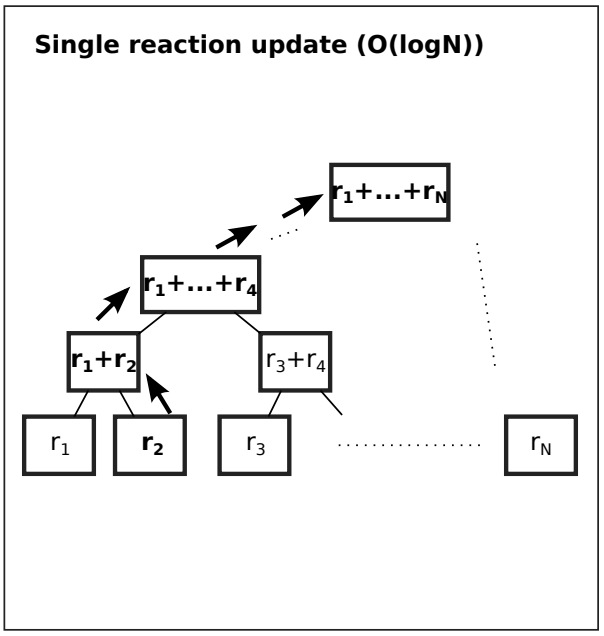
5.1.4. Hybrid method

General principle In this approach, we organize propensities into groups and use a different drawing method: *rejection-base drawing*. This approach has been presented in Slepoy et al. (2008).



Data: Binary tree with propensities at its leaves.
Result: Index drawn according to multinomial drawing.
u = uniform ([0, tree.root.value));
node = tree.root;
while *node is not a leaf* **do**
  if *u < node.left_child.value* **then**
    node = node.left_child;
  else
    node = node.right_child;
    u = u - node.left_child.value;
  end
end
return *node.index*

Figure 33: Binary tree: drawing method.



Data: Binary tree with propensities at its leaves. Index i_update of propensity to update, new propensity value p_update.
Result: Updated binary tree.
node = tree.leaf [i_update];
node.value = p_update;
while *node is not root* **do**
  node = node.parent;
  node.value = node.left_child.value
   + node.right_child.value;
end

Figure 34: Binary tree: update method.

Rejection-based drawing The aim is to perform a multinomial drawing, similar to what is done by a biased wheel or a binary tree. The problem with the latter methods is that we need to iterate through some structure before finding the right value. With rejection-based drawing, we attempt to *jump* to the right solution. Let $\{r_i\}$ be the propensities of the N reactions in the system and $R = \sum_{1 \leq i \leq N} r_i$.

1. We choose a value r_M such that $\forall i, r_M \geq r_i$.

2. Until a good candidate is found.

    a) We draw a random number i between 1 and N (with replacement).

    b) We draw a random number u on the $[0, r_M]$ segment. If $u > r_i$, we reject i, else we keep it.

A careful proof shows that the probability of drawing index i is equal to r_i/R (Serebrinsky, 2011). Note that the choice of r_M is critical for the efficiency of the method (Fig. 35A). Formally, the probability to accept a candidate is $\sum_i \mathbb{P}(\text{draw } i)\, \mathbb{P}(\text{accept } i) = \sum_i 1/N \times r_i/r_M = R/(Nr_M)$. If applied naively, the number of candidates to loop through is a geometric law with parameter $R/(Nr_M)$. The expected number of candidates is thus Nr_M/R. For a uniform distribution, this value can be 1, but in general, it yields bad results (Fig. 35B).

Group method The idea behind the algorithm is to improve the acceptation probability by placing propensities in *groups*:

1. We draw a group index by using a classical method (biased wheel or binary tree).

2. We draw a propensity inside the group by using the rejection-based method.

Slepoy et al. (2008) propose placing propensities into binary groups. They choose a base rate b. Groups are of the form $(0, b]$, $(b, 2b]$, $(2b, 4b]$, *etc.* $(0, b]$ contains all propensities between 0 and b, and so on. When applying the rejection method to any of these groups (except $(0, b]$), the acceptation probability is $\geq 1/2$ (Fig. 35C). Note that the number of groups K does not generally depend on N, it only depends on the highest propensity value. In general, the number of groups remains relatively small.

Suppose the structure is already in place, *i.e.* propensities are placed in the right group and the total propensity for each group is known. Step 1 is at most $O(K)$, which is independent of N. Step 2 requires less than 2 candidates on average, so it is $O(1)$. This results in $O(1)$ globally, making it significantly more efficient than the two previous methods. However, its implementation is also trickier in order to preserve this theoretical complexity.

Sketch of algorithm and complexity Because of the group structure, the loop in Figure 37 is $O(1)$ (see above). The first multinomial drawing is at most $O(K)$, so the complexity is globally $O(K)$. Because K, the number of groups, does not naturally scale with N, the number of reactions, the complexity is overall $O(1)$, as N is the real variable of interest here.

At first sight, updating the group structure is also $O(1)$ (Fig. 38). However, the parts about removing or inserting a reaction into a group must be carefully implemented in order to achieve that result.

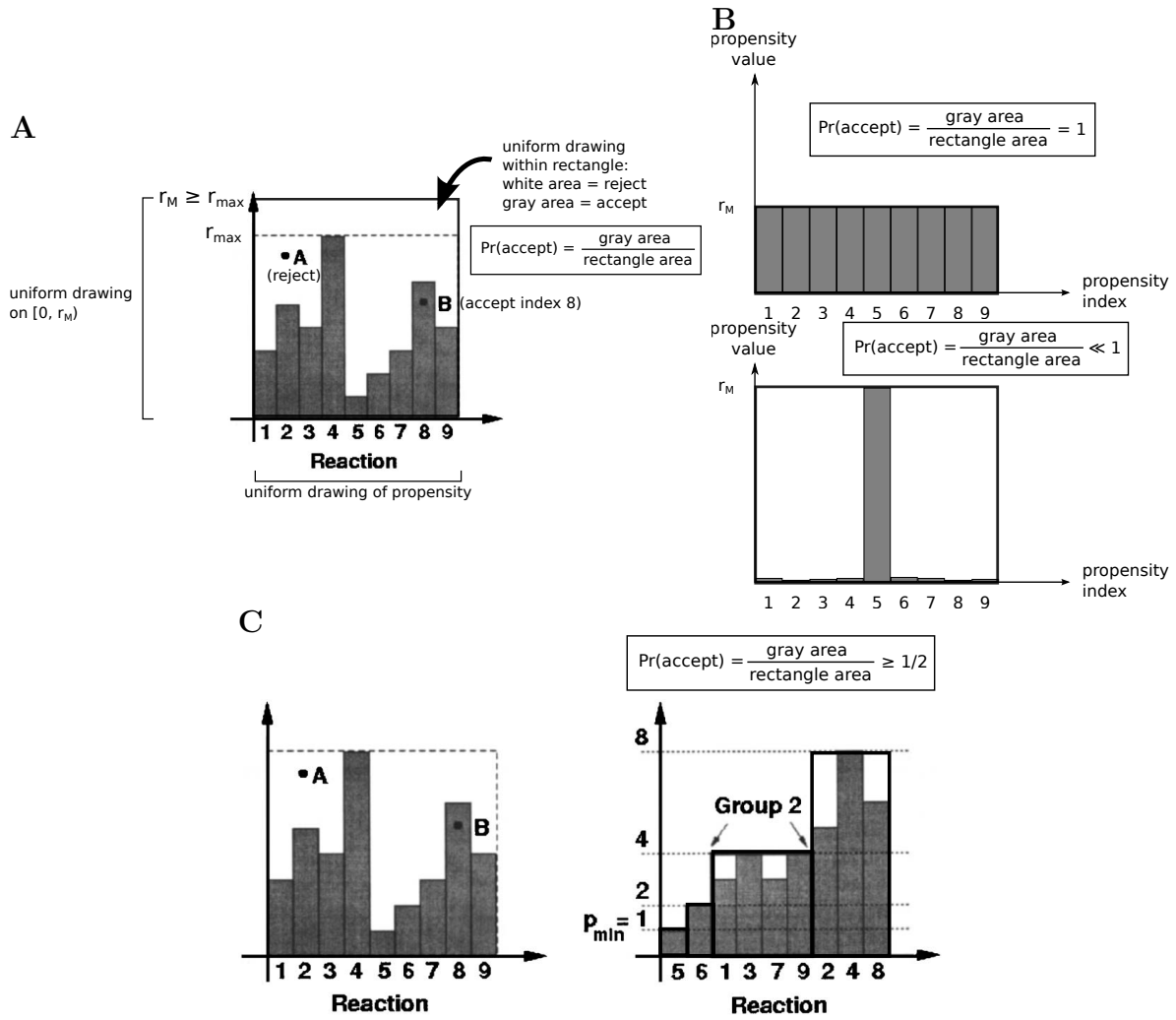


Figure 35: Rejection based drawing (adapted from Slepoy et al. (2008)). (A) Geometric illustration of rejection principle. Drawing occurs in a 2D space, with propensities aligned along the x axis and their value given by gray bars along the y axis. A drawing is accepted if it falls into the gray domain. Note that the probability to draw a propensity is proportional to its value, as an accepted drawing will be distributed uniformly across the gray domain. (B) Examples displaying efficiency of the technique (maximal for uniform propensities, minimal when some are very high and most are very low). (C) Sorting propensities into groups whose limits are powers of 2 ensures a minimal 1/2 acceptation probability *within a given group*.

5.1.5. Summary

Table 1 summarizes the worst case complexity of four methods presented. Note that the update complexity was derived in the case where only one propensity needed to be updated. To obtain the overall complexity, we need to take into account the number of

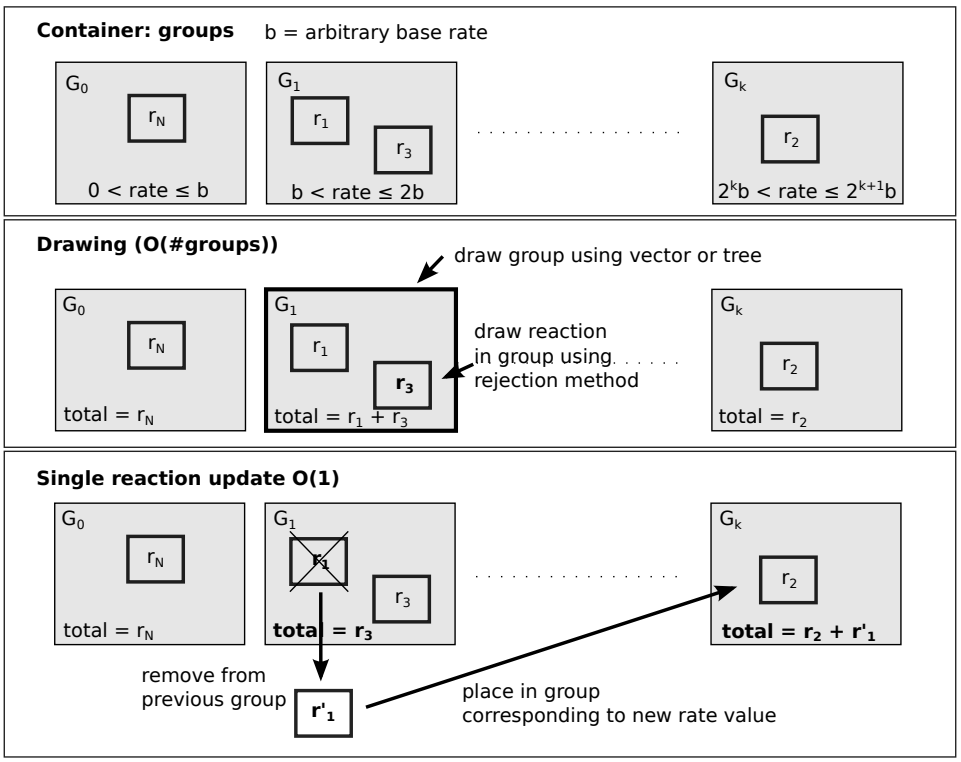
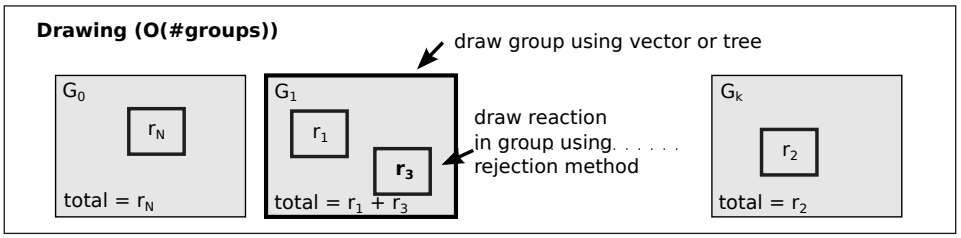


Figure 36: Hybrid method using group structure and rejection algorithm.



Data: $K+1$ groups, group k containing propensities whose value falls in the interval $(0, b]$ if k=0, $(2^{k-1}b, 2^k b]$ if k > 0. Propensities are stored as a couple containing their value and original index.

Result: Index drawn according to multinomial drawing.

`// drawing using a direct method like binary tree or biased wheel`

group = groups [multinomial (group[0].total_propensity, . . . ,
group[K].total_propensity)];

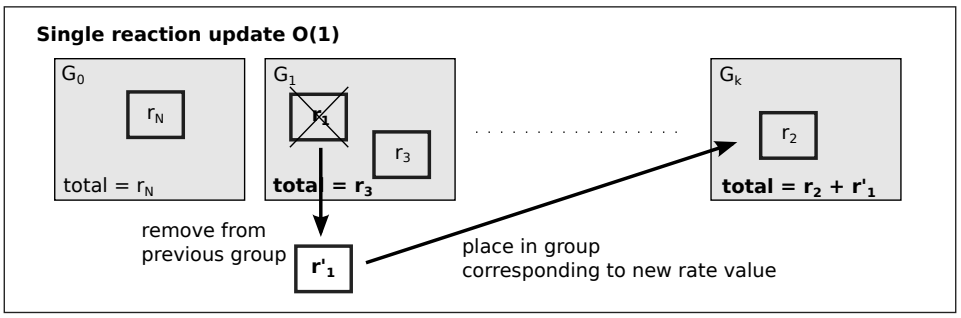
repeat

  candidate = group.propensities [uniform (1, group.number_propensities)];

until *candidate.value > uniform (0, group.max_propensity)*;

return *candidate.index*

Figure 37: Hybrid method: drawing method.



Data: $K+1$ groups, group k containing propensities whose value falls in the interval $(0, b]$ if k=0, $(2^{k-1}b, 2^k b]$ if k > 0. Propensities are stored as a couple containing their value and original index. Index i_update of propensity to update, new propensity value p_update.
Result: Updated group structure.
Function *group_index (propensity)*
  if *propensity* $\geq$ *b* **then**
    return $\lceil \log_2(reaction.propensity/b) \rceil$
  else
    return *0*
  end
propensity = propensity corresponding to index i_update;
previous_group = groups [group_index (propensity.value)];
Remove propensity from previous_group and update group's total propensity;
new_group = groups [group_index (p_update)];
propensity.value = p_update;
Add propensity to new_group and update group's total propensity;

Figure 38: Hybrid method: update method.

reactions U whose propensity needs to be updated. A naive analysis indicates that the binary tree could be less efficient than the direct method depending on U and N (Tab. 1). In the next section, we will analyze how U can influence the efficiency of each method.

Method	Drawing Complexity	Update Complexity (one propensity)	Total Complexity
Direct Method	$O(N)$	$O(1)$	$O(N)$?
Binary Tree	$O(\log N)$	$O(\log N)$	$O(U \log N)$?
Hybrid Method	$O(1)$	$O(1)$	$O(U)$?

Table 1: Comparison of worst-case complexities of methods presented here. N is the number of reactions in the system, $U \leq N$ the number of reactions whose propensity needs to be updated. The last column is a projection based on the first two columns, real total complexities are given in the next section.

5.2. Propensity updating

In Section 5.1, we have introduced methods to perform a multinomial drawing among N propensity values. We have seen that special structures can improve the drawing time but there is an additional cost to update those structures when a propensity value changes. Here we will study how this additional cost evolves when U propensities need to be changed from one drawing to the next drawing.

5.2.1. Naive method: why not update everything?

We begin with the simple case where $U = N$.

Direct Method In the direct method, we need to compute the total propensity of the system, so we need to loop through all propensities at least once. In this sense, the overall update complexity is $O(N)$ and the total complexity (including drawing) is equally $O(N)$.

Binary Tree We need to update all leaves of the tree. Naively, this would lead to a $O(N \log N)$ overall update complexity. However, this assumes that each time a leaf value is updated, we update all the intermediate nodes up to the root. In practice, it is possible to differ update of intermediate nodes and just mark them as outdated. We only update them once all their children have been treated, so that every node is updated at most once. In the case where all leaves are updated, every node in the tree is updated exactly once. The complexity is thus given by the number of nodes in the tree, approximately $2N$, yielding $O(N)$ overall update complexity. The total complexity is then $O(\log N + N) = O(N)$.

Hybrid Method Here the update algorithm is applied to every propensity, yielding a overall $O(N)$ update complexity. Total complexity is $O(1 + N) = O(N)$.

Summary As a matter of fact, when propensities are updated naively, the benefit of using sophisticated structures is (asymptotically) lost (Tab. 2).

Method	Drawing Complexity	Update Complexity	Total Complexity
Direct Method	$O(N)$	$O(N)$	$O(N)$
Binary Tree	$O(\log N)$	$O(N)$	$O(N)$
Hybrid Method	$O(1)$	$O(N)$	$O(N)$

Table 2: Worst-case complexities for $U = N$.

It is absolutely necessary to keep U as small as possible. Formally, we need to determine a subset of propensities that includes all propensities that might have changed. In that case, specific drawing structures start to be interesting (Tab. 3). Roughly speaking,

the binary tree is better than the direct method as soon as $U < N/\log N$ and the hybrid method outcompetes the two other methods for $U < N$.

This trend is even stronger in a system where the number of propensities to update does not depend on N (Tab. 4). In this case, the direct method is linear, the binary tree logarithmic and the hybrid method constant time, yielding huge improvements when switching from one method to the next one.

Method	Drawing Complexity	Update Complexity	Total Complexity
Direct Method	$O(N)$	$O(U)$	$O(N)$
Binary Tree	$O(\log N)$	$O(\min\{U \log N, N\})$	$O(\min\{U \log N, N\})$
Hybrid Method	$O(1)$	$O(U)$	$O(U)$

Table 3: Worst-case complexities for a general $U \leq N$.

Method	Drawing Complexity	Update Complexity	Total Complexity
Direct Method	$O(N)$	$O(1)$	$O(N)$
Binary Tree	$O(\log N)$	$O(\log N)$	$O(\log N)$
Hybrid Method	$O(1)$	$O(1)$	$O(1)$

Table 4: Worst-case complexities for the ideal case $U = O(1)$ (number of propensities to update does not scale with the number of reactions N in the system).

5.2.2. Using graphs to update propensities

The most standard way to study how reactions influence each other is to draw a graph that tracks the dependencies between reactions. A first version can be drawn using a *bipartite graph* with reactions on one side and reactants on the other one. We can draw two sets of vertices (Fig. 39).

- If the reaction modifies the concentration of a reactant, a directed vertex is drawn from the reaction to the reactant.
- If the concentration of a reactant is used to compute the propensity of a reaction, a directed vertex is drawn that goes from the reactant to the reaction.

Reactant to reaction graph The first algorithm we propose only takes advantage of the set of vertices that goes from reactants to reactions. Here is the sketch of the algorithm:

- We set up an *observer pattern* to track changes in concentrations. Schematically, every time the concentration of a reactant changes, the reactant sends an update that will *invalidate* the propensities in which it is involved (as pointed by the vertices on the graph).

$$R_1 : A \longrightarrow C + D$$
$$R_2 : B + C \rightleftharpoons E$$
$$R_3 : B + E \longrightarrow 2A$$
$$R_4 : D + A \longrightarrow B + A$$

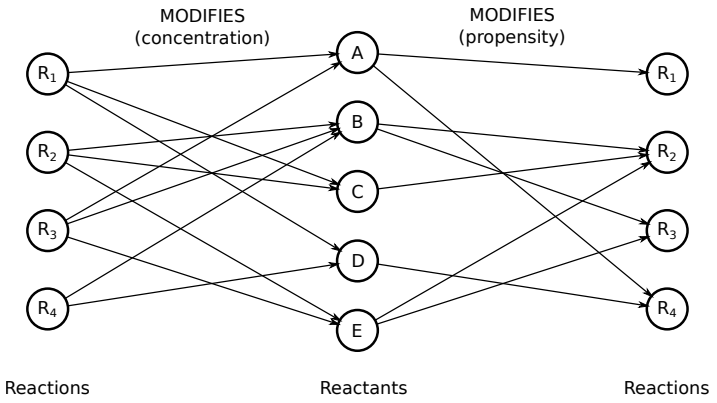


Figure 39: (Left) Sample reaction system. (Right) Corresponding bipartite graph displaying interactions between reactions and reactants. The reaction nodes were duplicated for readability.

- We initialize the system by computing all propensities.
- Until some target is reached:
  - A random reaction is performed. Concentration of some reactant changes. Updates are sent and some propensities are invalidated.
  - We recompute the propensities that were invalidated.

Reaction to reaction graph In the second algorithm, we collapse the bipartite graph shown above into a graph that only contains reactions. The collapsing process is rather intuitive if we use the directed vertices defined above. Take two reactions A and B. There is a vertex $A \to B$ if there is some reactant R such that $A \to R$ and $R \to B$ in the bipartite graph. Put into words: when reaction A occurs, it changes the concentration of a reactant R used to compute the propensity of B (Fig. 40).

Here is a sketch of the algorithm:

- We create the reaction graph: for each reaction we store the list of propensities to update when it occurs.
- We initialize the system by computing all propensities.
- Until some target is reached:
  - A random reaction is performed.
  - We recompute the propensities according to the reaction graph.

5.2.3. Pros and cons of graph algorithms

Simple system The main difference between the algorithms is that the reactant to reaction algorithm uses an observer pattern. This pattern it slightly more complicated to implement than the reaction graph. Moreover, in closed systems of chemical reactions,

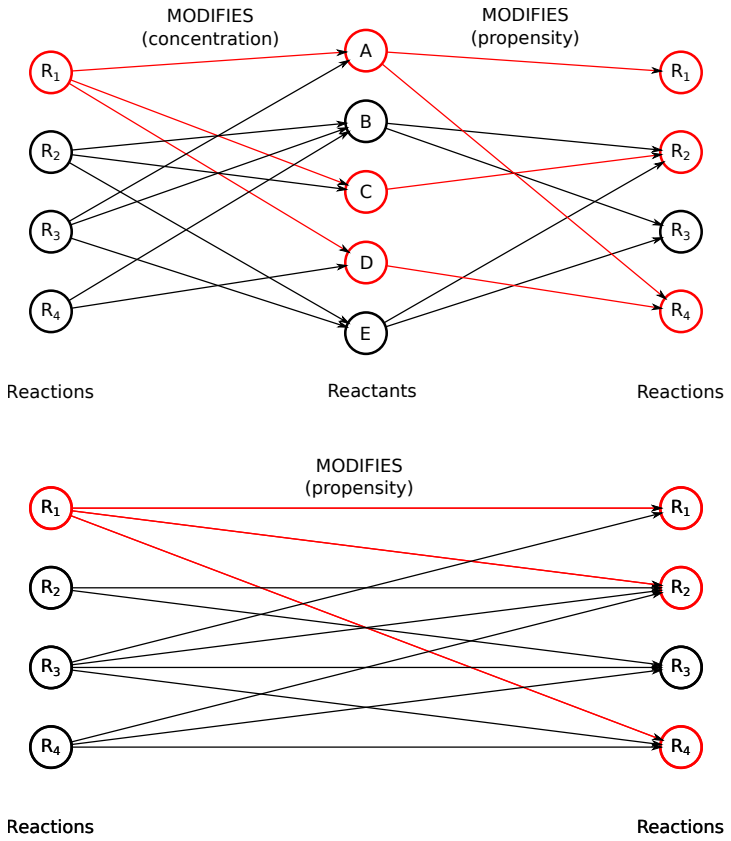


Figure 40: Collapsing bipartite graph into a pure reaction graph. (Top) Example of collapsing for reaction R_1. All paths leading to a potential propensity change in other reactions are displayed in red. (Bottom) Reaction to reaction graph. In red are the vertices that correspond to the paths displayed in red in the bipartite graph. Reaction nodes were duplicated for readability, the graph could also be represented with only one node for each reaction.

the reaction graph approach is more efficient. We can perform a very simple cost analysis for the update occurring after one reaction has been performed.

- Rectant to reaction graph: for every reactant involved in the reaction, an update is sent that invalidates a subset of propensities. The subsets invalidated by the reactants might overlap, resulting in invalidating the same propensity several times. Note that invalidating does not mean recomputing the propensity, it is only recomputed after the whole invalidation process if it has been tagged by at least one reactant.

- Reaction to reaction graph: when the reaction occurs, we have direct access to the subset of propensities to recompute.

We see that there are additional costs in the reactant to reaction method because of the update sending and possible overlaps that are eliminated when the reactant to

reaction graph is collapsed into the simpler reaction graph. Depending on the system, these additional costs may not be negligible, even though they should remain fairly low.

System with external constraints If something else than a reaction might change the concentration of a reactant, propensities will be updated correctly only when using the reactant to reaction algorithm. When using the reaction graph, such interactions are not visible and it may be necessary to update all propensities every time such an event occurs.

This flexibility problem can be overcome without an observer pattern by storing both the reaction graph *and* the bipartite graph to use the reactant to reaction graph when needed. However, the solver still needs to be notified every time the concentration of a reactant changes due to external reasons. There is a danger that this notification might be omitted by the programmer, while an observer pattern guarantees that the solver will be aware of such changes.

System containing aggregated reactions In more complex systems that do not only include chemical reactions, it is possible to include reactions that have multiple outcomes. We developed a simulator that includes a `Loading` reaction. A typical example is a RNA polymerase building a RNA by matching either an ATP, a CTP, a GTP or a UTP depending on the DNA base it is reading. The base is read dynamically during simulation, so we do no know in advance which NTP pool is going to be impacted by the reaction.

In this case, when collapsing the reactant to reaction graph, we would need to consider all possible outcomes. We would end up updating all propensities involving ATP, CTP, GTP and UTP, even though only one NTP was really affected. The reactant to reaction graph is more efficient in this case, as updates are only sent for the NTP effectively used, leading to fewer propensity computations.

System containing complex dependencies between reactions An even trickier issue are hidden dependencies between reaction. Let us quote an example from our simulator again. It contains a `SequenceBinding` reaction, where a molecule can bind on a binding site, *e.g.* a transcription factor binding on DNA at a specific site. The propensity depends on the availability of the transcription factor but also the availability of the binding site. Suppose a DNA polymerase happens to hide the binding site, how can we make sure the binding propensity reflects the unavailability of the binding site?

With a reaction to reaction graph, it is difficult to account for that situation in a comfortable way. We might be tempted to introduce special interactions between instances of reactions (translocation of a polymerase *may* change binding propensity). In our opinion, this would lead to bad software compared with the reactant to reaction approach. We can declare a binding site to be a reactant that gets notified every time its availability changed. This is enough to ensure that all propensities will be updated properly. Implementation is simple and transparent. Everything is based on a single abstraction (reactant concentrations determine propensities): no painful special cases.

Conclusion In simple systems, the reaction graph is both superior in speed and easier to implement. In more complex systems, the reactant to reaction graph proves to be significantly more flexible and possibly more efficient. We started by implementing both algorithms in our simulator. At the beginning, when using simple reaction systems, the reaction graph algorithm was slightly superior. When simulating more complex systems, the reaction graph proved to be nearly impossible to maintain. Even though it may be slightly more costly in simple cases, the reactant to reaction graph provides a better abstraction when extending towards more complex systems. Implementation remains easy and *safe*: propensities are updated the way they should. This became harder and harder to achieve using the reaction to reaction approach, so we ended up abandoning it in our simulator.

A. Utility classes

A.1. Exceptions

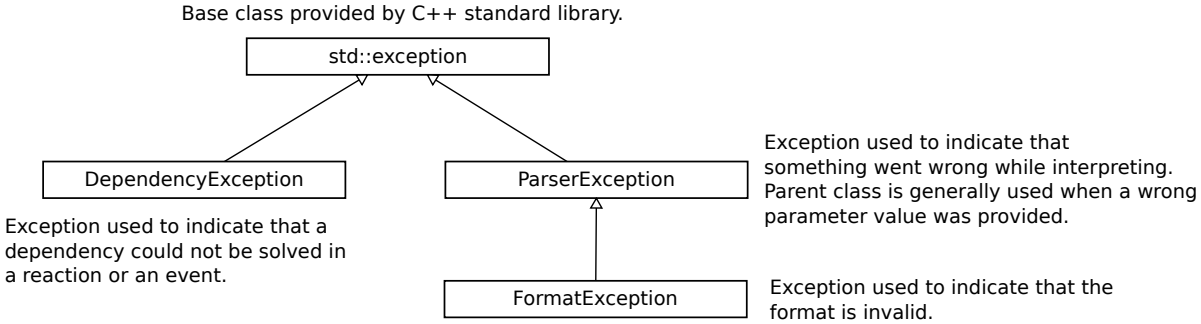


Figure 41: Exceptions used in the simulator.

Programming by contract (see Section B) covers most internal errors that might happen. Exceptions are only used when user input is treated. They are used to signify inconsistencies in input files during the parsing step (Fig. 41).

A.2. Random handler

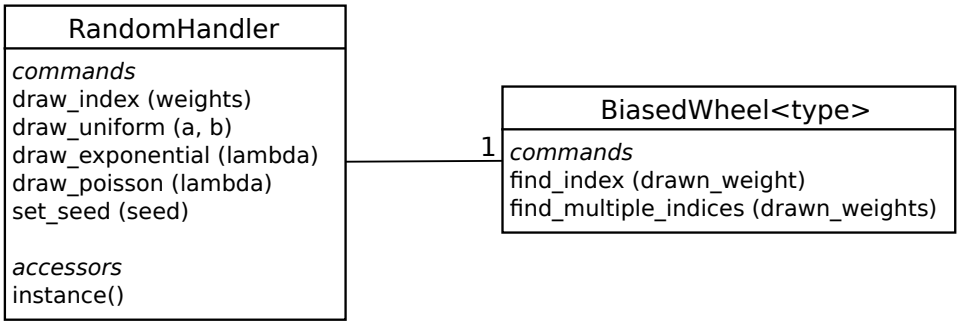


Figure 42: `RandomHandler` and `BiasedWheel` used for multinomial drawing. `RandomHandler` uses the Singleton pattern, meaning exactly one instance of the class is created and used throughout the simulator. It has to be accessed using `RandomHandler::instance()`.

A unique `RandomHandler` is used throughout the simulation to control the random seed by using a Singleton pattern (Fig. 42).

A.3. Factories

Factories are used to remember user options and handle memory more efficiently (Fig. 43).

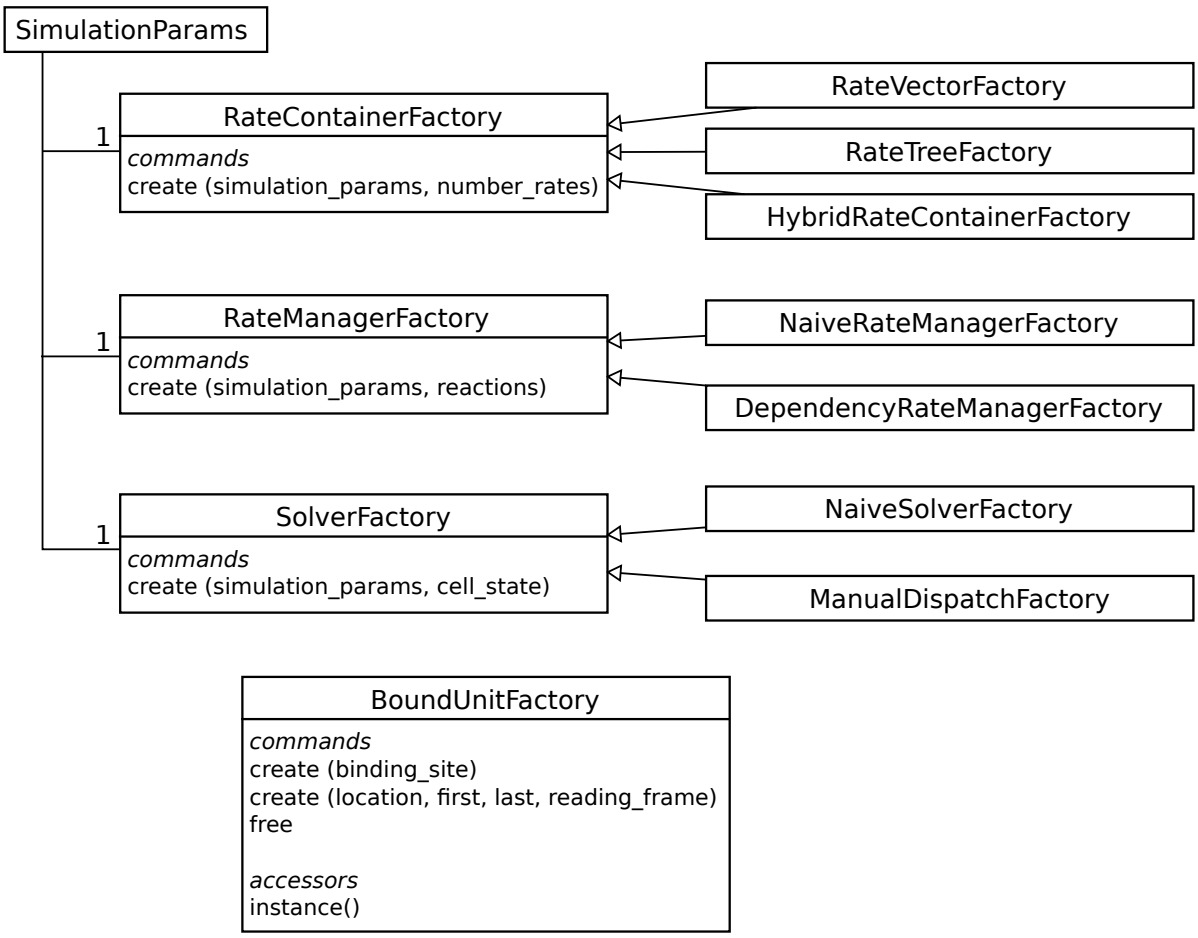


Figure 43: `RateContainerFactory`, `RateManagerFactory` and `SolverFactory` are used by `SimulationParams` to record what `RateContainer`, `RateManager` and `Solver` the user wishes to use (Abstract Factory Pattern). `BoundUnitFactory` is used to control memory usage by `BoundUnit`s. It enables recycling of bound units, avoiding memory reallocation. It uses the Singleton pattern to make sure all instances of `BoundUnit`s are stored at the same place.

A.4. Vector-based containers

The simulator uses two non-standard containers, `VectorList` and `VectorQueue`. A typical example is a `BoundChemical` storing all its `BoundUnit`s. There is typically a high turnover of bound units and units reacting are drawn uniformly within the list of bound units. Naively, we would use a list to perform such a task, but there are huge performance issues. Using standard C++ `std::list` would imply a lot of memory reallocation each time a `BoundUnit` is added/removed (because a node of the list is created/deleted). What is more, if, by random drawing, we decide that it is the 10th unit that is going to react, we need to loop through 10 elements before accessing the correct element.

Using a `std::list` has a *huge* impact on performance. Therefore, in `BoundChemical` (and a lot of other places in the program), we replace `std::list` by a `VectorList`, where elements are placed in a `std::vector`. There is one trick to use: every time an element is removed, it is replaced by the last element in the vector, so that elements remain contiguous in memory. This means that order of elements is lost, but in the example above and every time we `VectorList`, order is not important. The size of `std::vector` is automatically adjusted by C++. Most of the time it will be larger than the number of elements it contains, but we prefer using a little more memory than constantly reallocating nodes. What is more, accessing the nth element is instantaneous.

The same general idea applies for `VectorQueue` except we need to know in advance how large the queue will be. It is used to update nodes in `RateTree` because we know how many nodes there are in the tree and they are updated at most once.

A.5. Handler classes for memory handling

C++ has no garbage collector and this program was designed according to old standards (that is without smart pointers). Therefore, we need to be extremely careful to delete elements at the right time. This is achieved by using as few storage places as possible. As described earlier, `CellState` is used to store all reactions and reactants read from files. We designed a `Handler` class that effectively stores objects to their definitive location and distributes references or pointers to this constant location. Similarly `EventHandler` and `BoundUnitFactory` are used to store `Event`s and `BoundUnit`s which are the other dynamical elements stored in the program. At the end of the simulation, all these handlers have to be carefully deleted. Observer patterns are particularly dangerous, as we must be sure that at destruction, there is no message sent to a non-existent observer. Every time an observer pattern is used, we made sure that observers are correctly unregistered when they are destructed or the object they observe is destructed.

B. Tests

B.1. Testing philosophy

We use three layers of tests: *programming by contract*, *unit tests*, *integration tests* (Tab. 5). They are integrated in an automated framework to detect bugs rapidly and at the lowest possible level.

Programming by contract These tests typically apply to attributes of classes and arguments of methods. They are usually divided into three subcategories: *preconditions*, *postconditions* and *invariants*. They check whether the class interacts correctly with the outside world, generally other classes. We left invariants out because they are hard to check in a language that does not support them natively. In the simulator, we defined two macros `REQUIRE` and `ENSURE` to test pre- and postconditions. Each time a precondition or a postcondition is broken, the program is interrupted and the condition that was

Test type	Preconditions Postconditions Invariants	Unit Tests	Integration Tests
Test level	Implementation details	Class interface	Systemic
Time per test	a few instructions (ns)	ms to a few seconds	seconds to several minutes
Use frequency	Permanent	Very frequent	Less frequent

Table 5: Comparisons of tests used to develop the simulator

broken is displayed (using `assert()`). Preconditions and postconditions are extremely useful for detecting simple typing mistakes, numerical issues and so on (Fig. 44).

```
int RateTree::find (double value) const
{
  /** @pre value must be smaller than total tree rate. */
  REQUIRE (value <= total_rate());
  /** @pre value must be strictly positive. */
  REQUIRE (value > 0);
  int index = _root->find (value);
  // rarely, the algorithm will fail because of rounding problems and
  // return a leaf with zero rate. We just take the next nonzero rate.
  while (_leaves [index]->rate() == 0)
    { ++index; if (index == _leaves.size()) { index = 0; } }
  /** @post Rate of returned leaf must be strictly positive. */
  ENSURE (_leaves [index]->rate() > 0);
  return index;
}
```

Figure 44: Example of the use of preconditions with `REQUIRE` and postconditions with `ENSURE`. Here the postcondition helped discover a rare bug due to rounding problems that is now addressed in the code.

Unit tests Unit tests test the behavior at the class level. We used some simple guidelines to try and write useful and maintainable tests. We only wrote tests for the most important classes of the simulator.

Integration tests This is the last layer of test. The whole simulator or large pieces of it are used. The idea is to test more systemic behaviors, in which the interaction of classes is crucial. Typical examples are:

- If we provide known DNA and define RNAs and proteins correctly according to simulator input, the sequence of proteins as processed by the simulator should match known proteins.
- If we provide DNA, define RNAs, provide a transcription pathway but activate only one promoter, only the RNA associated to that promoter should be transcribed.

B.2. Organizing and running tests

Tests are associated to the source code of the simulator in order to be run as frequently and as simply as possible. Tests are driven by Unix Autotools and use the BOOST Test Framework. Preconditions and postconditions are written inside the code, unit tests are regrouped in a `tests/unit_tests` directory and integration tests are stored in `tests/integration`.

Options of `./configure` are used to activate every layer of test individually. By default, all tests are turned off. Several layers can be activated simultaneously.

- `--enable-pre-check` enables preconditions.
- `--enable-post-check` enables postconditions.
- `--enable-unit-tests` enables unit tests and the possibility to create mock objects.
- `--enable-integration-tests` enables integration tests and the possibility to create mock objects.

Code needs to be recompiled after it has been configured.

- Preconditions and postconditions are automatically checked every time the program is run (for unit tests, integration tests or any kind of other run). Remember to turn them off for real simulations as they are very time consuming.
- Unit tests and/or integration tests are run by running `make check`. BOOST automatically generates useful and human readable messages about tests that failed or clearly indicates that all tests have passed.

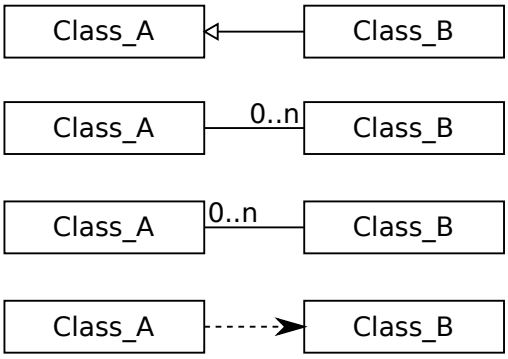
C. Formats and Conventions

C.1. Input format description

- A plain word indicates a tag, that needs to be written.
- `<...>` indicates a variable that has to be completed with an existent element of the specified type.
- `[...]` indicates an optional part.

- `[...]^{0..n}` indicates an optional part that can be repeated an arbitrary number of times.
- `[...]^{1..n}` indicates an part that can be repeated an arbitrary number of times, at least once.
- `[...,]^{0/1..n}` indicates a part that can be repeated an arbitrary number of times, each repetition being separated by a `,` (*but there is actually no* `,` *after the last repetition*).

C.2. UML



Inheritance: Class_B inherits Class_A.

Composition: Class_A contains/knows 0 to n elements from Class_B.

Composition: Class_B contains/knows 0 to n elements from Class_A.

Temporary Composition: Class_A temporarily contains/knows an element from Class_B.

Figure 45: UML format used.

References

Daniel T. Gillespie, Andreas Hellander, and Linda R. Petzold. Perspective: Stochastic algorithms for chemical kinetics. *J Chem Phys*, 138 (17), May 2013. ISSN 0021-9606. doi: 10.1063/1.4801941. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3656953/.

Santiago A. Serebrinsky. Physical time scale in kinetic Monte Carlo simulations of continuous-time Markov chains. *PhysRev E*, 83(3), March 2011. ISSN 1539-3755, 1550-2376. doi: 10.1103/PhysRevE.83.037701. URL http://link.aps.org/doi/10.1103/PhysRevE.83.037701.

Alexander Slepoy, Aidan P. Thompson, and Steven J. Plimpton. A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks. *J Chem Phys*, 128(20):205101, 2008. ISSN 00219606. doi: 10.1063/1.2919546. URL http://scitation.aip.org/content/aip/journal/jcp/128/20/10.1063/1.2919546.