BiPSim: a flexible and generic stochastic simulator for cell processes - Supplementary Information

December 9, 2019

Contents

1 Introduction

This document walks through the choices in design that we made while developing BiPSim (the code is documented using Doxygen, technical details are therefore best found in the Doxygen-generated manual). It highlights the central classes in BiPSim's architecture and how classes interact.

The first section focuses on the base components of the simulator, reactants and reactions. It starts with a global overview of all rectants and reactants. A second subsection describes the same element again, but goes further into hypotheses and critical design elements. 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.).

The second section focuses on the implementation of Gillespie's Stochastic Simulation Algorithm (SSA). It starts with an overview of the SSA, insisting the trade-off between two of its components: selecting the next reaction to perform and updating reaction rates. The second subsection presents various strategies to select the next reaction implemented in BiPSim: the direct method, the binary search and the composition-rejection method. The third subsection presents efficient strategies to update reaction rates as implemented in BiPSim, which play an essential role in the final performance of the simulator. Appendices provide additional information about the implementation and some perspectives.

2 Implementation of reactions and reactants

2.1 Global presentation of the 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

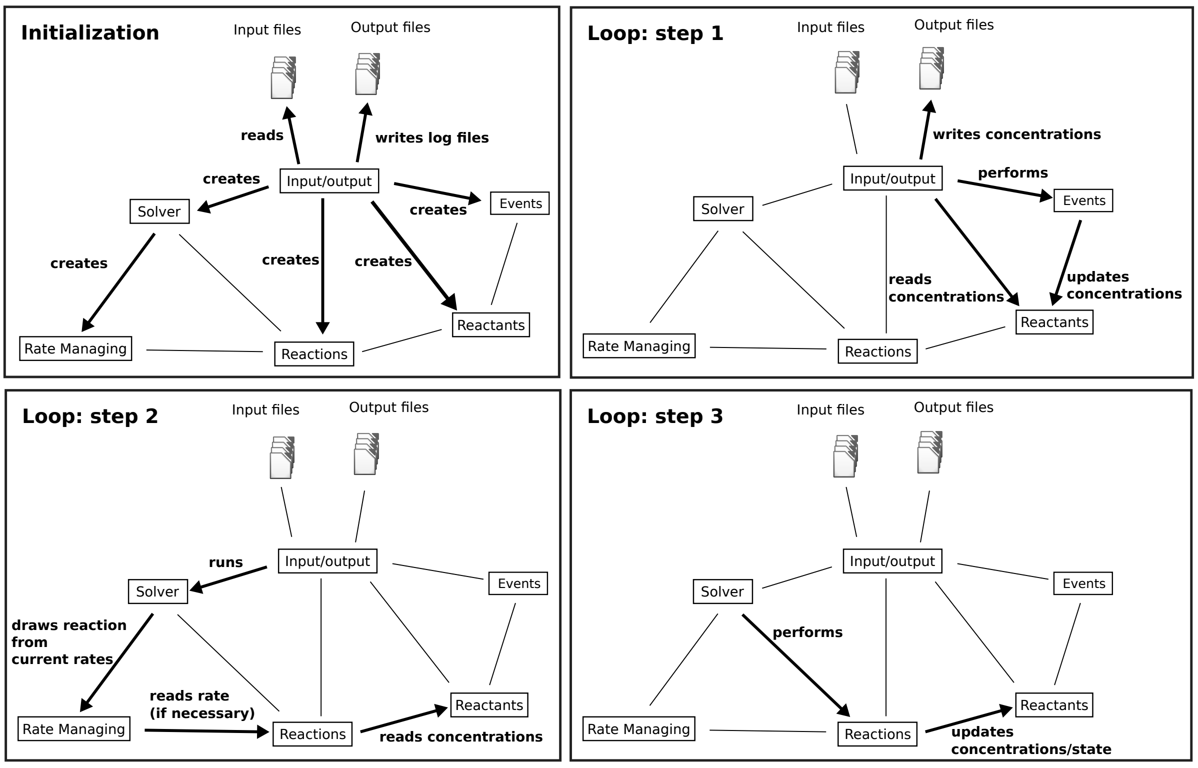


Figure 1: Schematical view of the simulator.

uses state-of-the-art methods to maintain the rate list updated and perform the drawing efficiently.

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

2.2 Reactant hierarchy

The following sections give a quick overview of the contents of the `Reactant` hirarchy (Fig. 2). More details about how reactants are implemented can be found later.

2.2.1 Reactant

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

2.2.2 Chemical

`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.

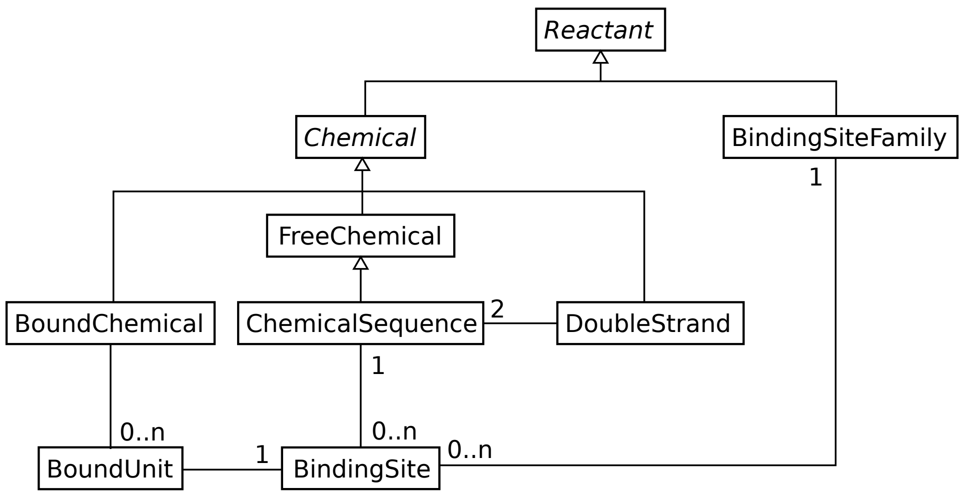


Figure 2: UML diagram of `Reactant` hierarchy

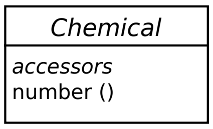


Figure 3: `Chemical` class

2.2.3 FreeChemical

Input format

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

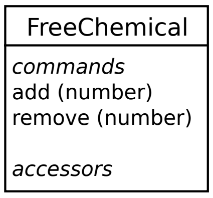


Figure 4: `FreeChemical` class

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

2.2.4 BoundChemical

Input format

```
BoundChemical <name>
```

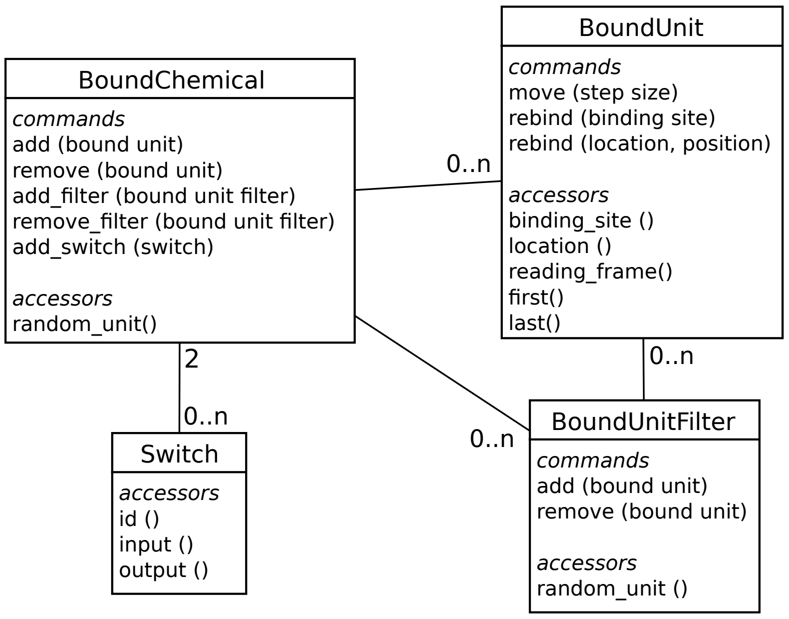


Figure 5: `BoundChemical` class

`BoundChemical` (Fig. 5) is a subclass of `Chemical` that represents chemicals that are bound to a sequence. It is important to note it only represents molecules bound to the sequence, *not* the complex formed by the chemical and the 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).

2.2.5 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 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

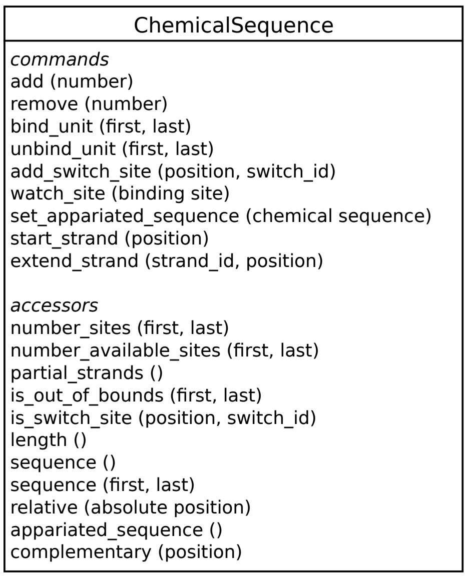

ChemicalSequence
commands
add (number)
remove (number)
bind_unit (first, last)
unbind_unit (first, last)
add_switch_site (position, switch_id)
watch_site (binding site)
set_appariated_sequence (chemical sequence)
start_strand (position)
extend_strand (strand_id, position)
accessors
number_sites (first, last)
number_available_sites (first, last)
partial_strands ()
is_out_of_bounds (first, last)
is_switch_site (position, switch_id)
length ()
sequence ()
sequence (first, last)
relative (absolute position)
appariated_sequence ()
complementary (position)

Figure 6: `ChemicalSequence` class

bound chemicals occupying the site from the number of instances of the mRNA currently in the cell. A `ChemicalSequence` can be appariated 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.

2.2.6 DoubleStrand

Input format

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

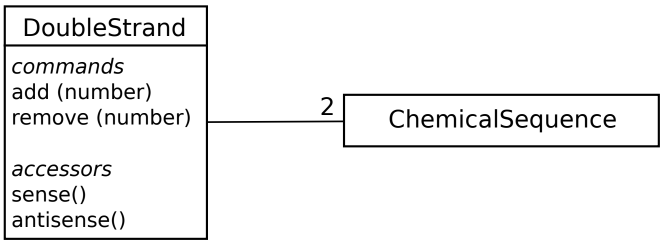


Figure 7: `DoubleStrand` class

`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 appariated 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, $A \to T$, $T \to A$, $C \to G$, $G \to C$).

2.2.7 BindingSiteFamily

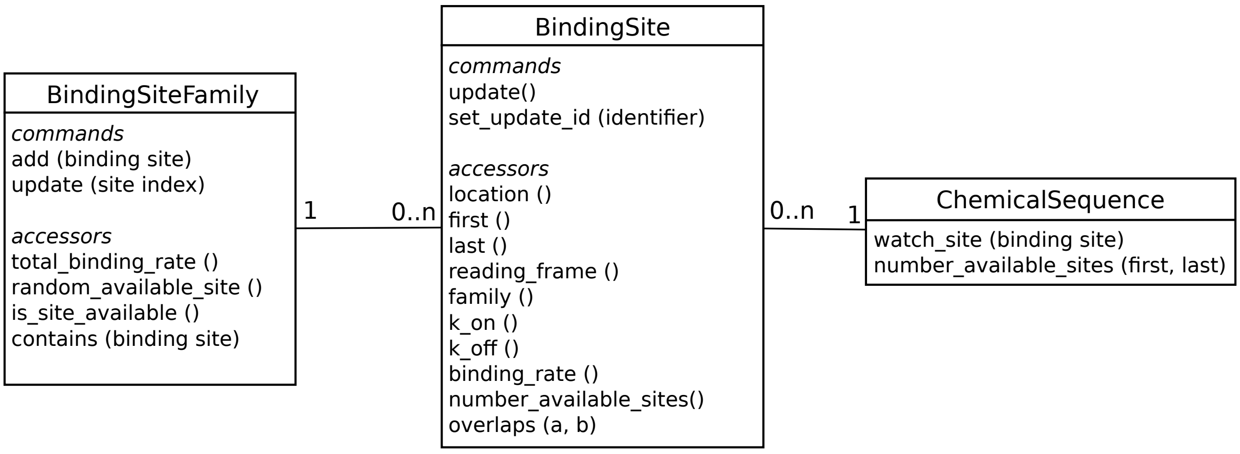


Figure 8: `BindingSiteFamily` class

`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.

Input format

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

2.3 Reaction hierarchy

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

2.3.1 Reaction

There are two abstract classes used to define reactions: `Reaction` for one-way reactions and `BidirectionalReaction` for reversible reactions. Two adapter classes

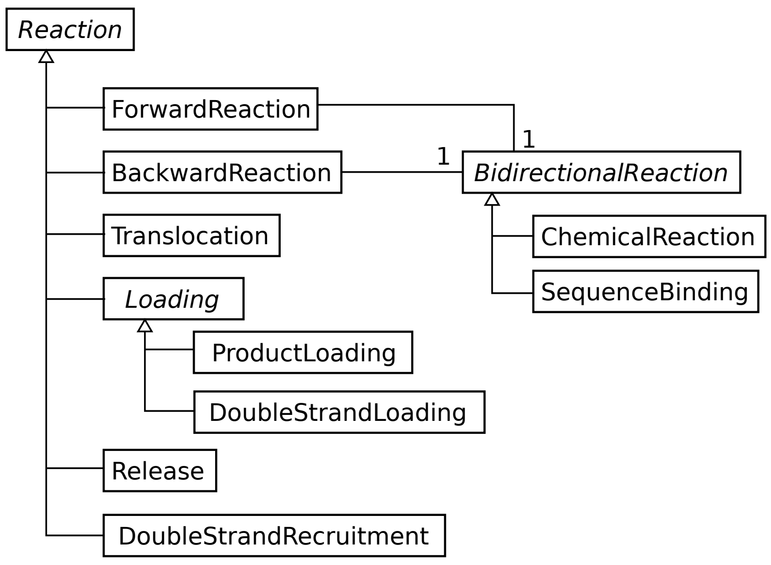


Figure 9: UML diagram of `Reaction` hierarchy.

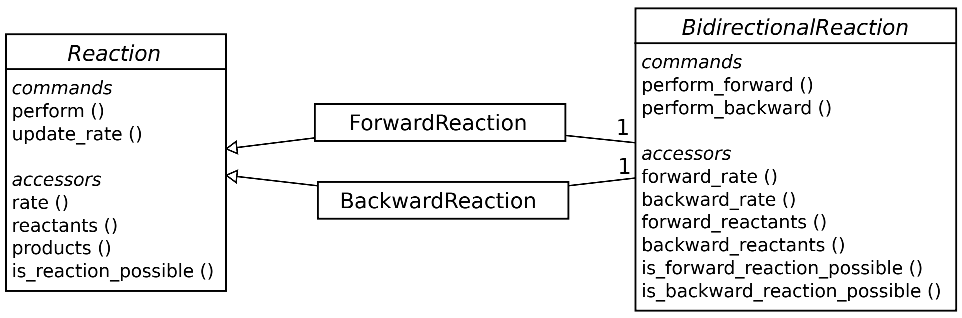


Figure 10: `Reaction` and `BidirectionalReaction` classes.

`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.

2.3.2 ChemicalReaction

Input format

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

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

$$a_1A_1 + a_2A_2 + ... + a_rA_r \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} b_1B_1 + ... + b_pB_p$$

where

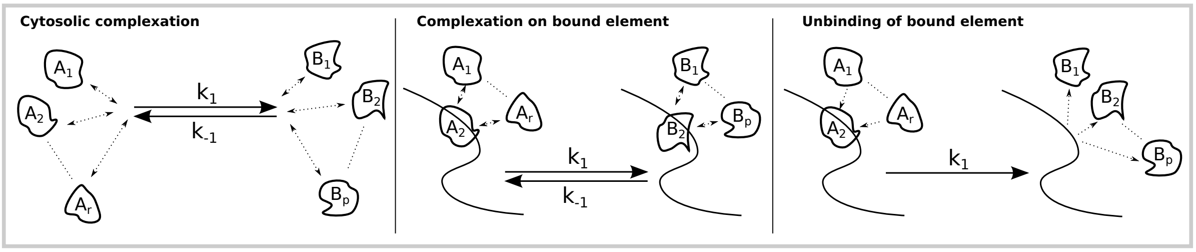


Figure 11: Schematic view of a `ChemicalReaction`.

- 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

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

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

2.3.3 SequenceBinding

Input format

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

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

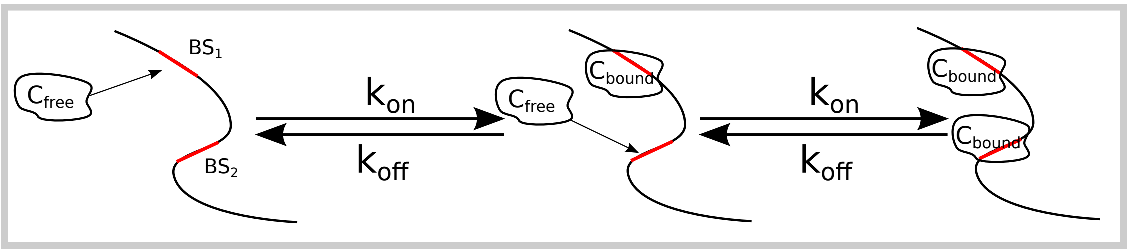


Figure 12: Schematic view of a `SequenceBinding`.

- 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.

2.3.4 Translocation

Input format

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

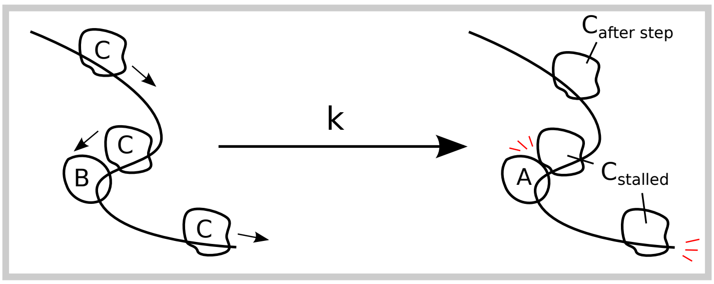


Figure 13: Schematic view of a `Translocation`.

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

$$\lambda = k[C]$$

2.3.5 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>
```

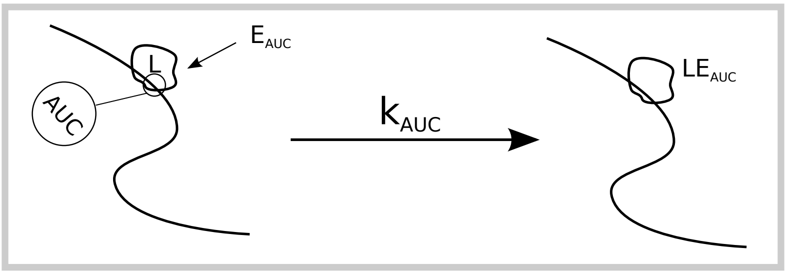


Figure 14: Schematic view of a `Loading`.

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.

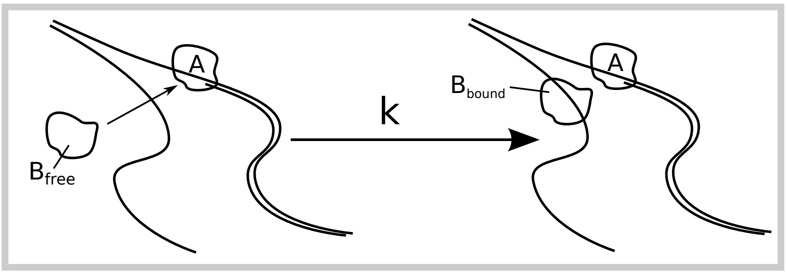


Figure 15: Schematic view of a `DoubleStrandRecruitment`.

2.3.6 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

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

2.3.7 Release

Input format

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

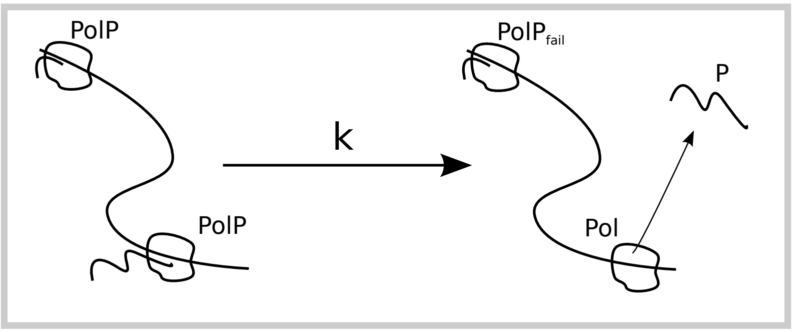


Figure 16: Schematic view of a `Release`.

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

$$\lambda = k[PolP]$$

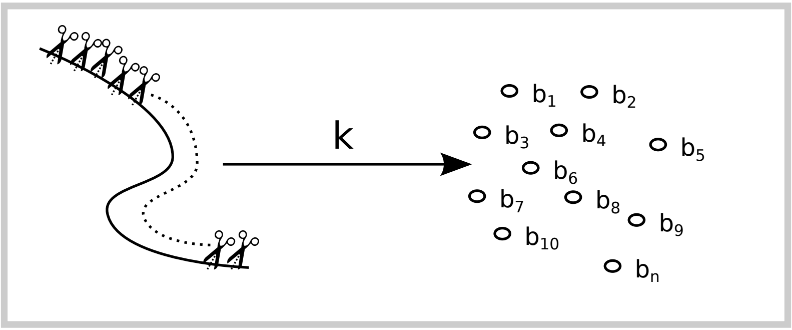


Figure 17: Schematic view of degradation reaction.

2.3.8 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

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

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

$$\lambda = k[CS]$$

2.3.9 Switches

Input format

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

`Switch`es are intrinsically linked to `BoundChemical`s but apply to specific `BoundUnit`s through `SwitchSite`s located on `ChemicalSequence`s. Every time an instance of `input_bound_chemica` steps on a switch site, it *immediately* becomes an `output_bound_chemical`.

A `Switch` is not considered a reaction because there is no rate associated with it (the solver does not actually know anything about switches). 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. Typically, termination sites for transcription are expected to be *SwitchSite*s. Similarly, important regulation sites can be implemented using *SwitchSite*s.

2.3.10 Solver loop

Once `Reaction`s and `Reactant`s 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.

- `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`: Childern 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.

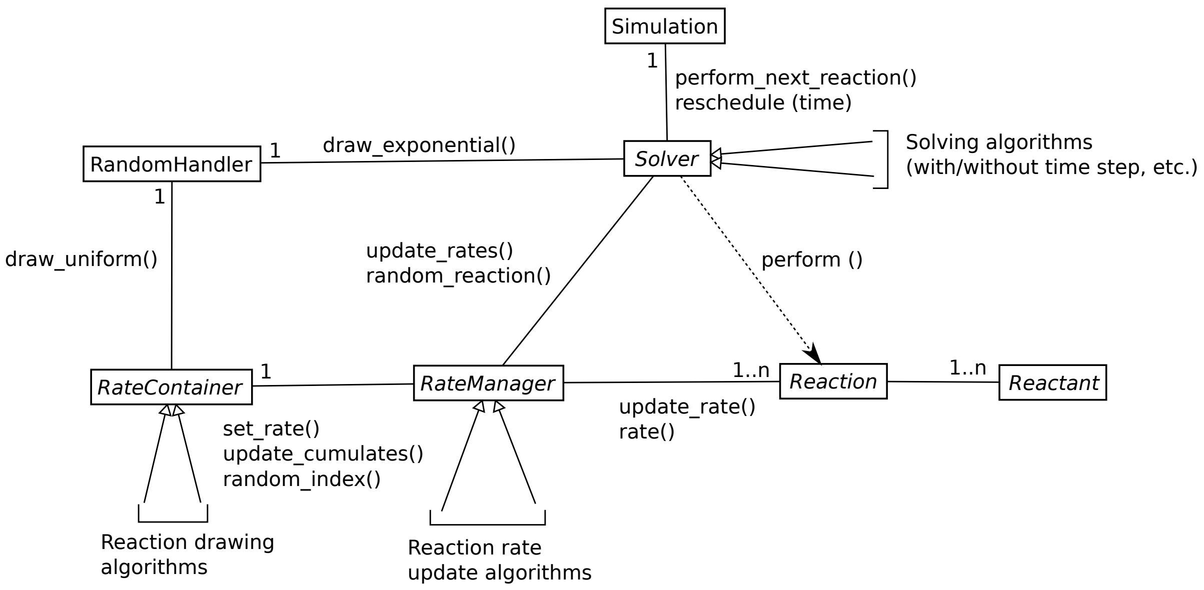


Figure 18: Solver loop. The loop is driven by the `Solver` class that defines how and when rates should be updated. The update task is performed by a `RateManager`. Once rates are known, multinomial drawing is delegated to a `RateContainer`. A central `RandomHandler` is used so that the solver only uses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

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

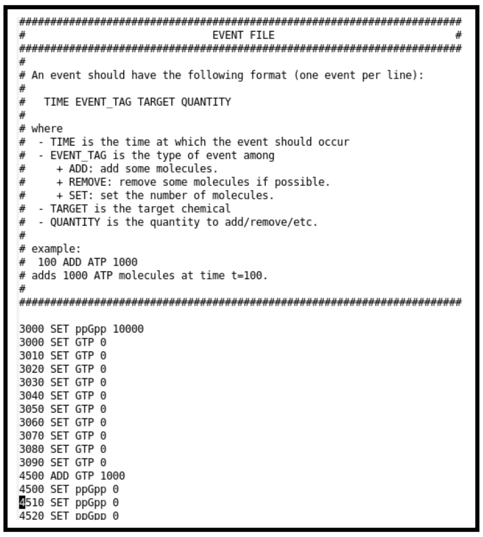
2.3.11 Events

`Events` 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.

2.3.12 Input/Output handling

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.



```
########################################################################
#                              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
```

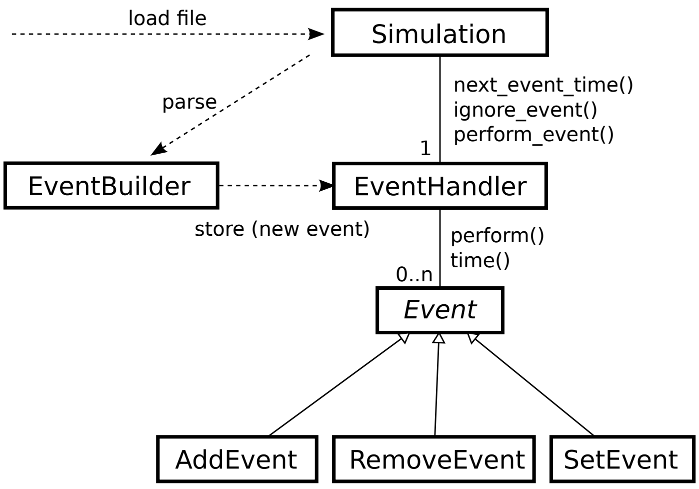


Figure 19: `Events`: another way to modify chemical concentrations aside from reactions, *e.g.* to simulate the injection of a chemical inside a cell.

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.

Simulator Output Outputs provided by the simulator are:

- A general output file logging parametes 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 ouput, a replication file describing replication advancement of that `DoubleStrand`.

2.4 Detailed design

2.4.1 Reactants

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

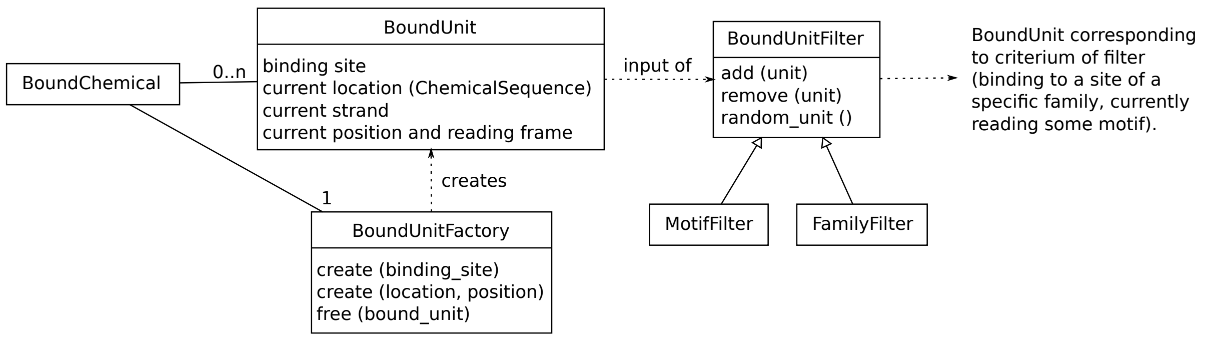

BoundUnit
binding site
current location (ChemicalSequence)
current strand
current position and reading frame
BoundChemical
0..n
1
creates
BoundUnitFactory
create (binding_site)
create (location, position)
free (bound_unit)
input of
BoundUnitFilter
add (unit)
remove (unit)
random_unit ()
MotifFilter
FamilyFilter
BoundUnit corresponding to criterium of filter (binding to a site of a specific family, currently reading some motif).

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` `BoundChemical` represents molecules of the same chemical species, but there are specifities 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 `BoundUnits`, avoiding memory reallocation throughout simulation. `BoundUnitFilters` are used to sort `BoundUnits` according to criteria useful for reactions (Fig. 20).

`BoundUnits` 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`.

`ChemicalSequence` `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.
- `BoundUnits` 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.

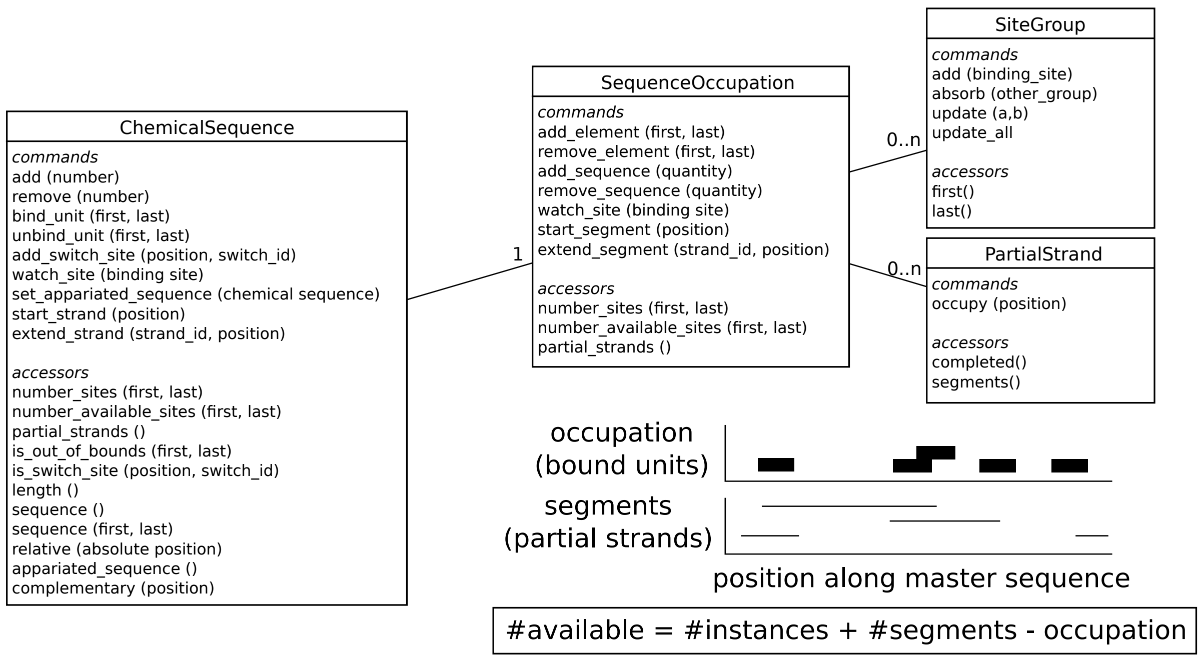

ChemicalSequence
commands
add (number)
remove (number)
bind_unit (first, last)
unbind_unit (first, last)
add_switch_site (position, switch_id)
watch_site (binding site)
set_appariated_sequence (chemical sequence)
start_strand (position)
extend_strand (strand_id, position)
accessors
number_sites (first, last)
number_available_sites (first, last)
partial_strands ()
is_out_of_bounds (first, last)
is_switch_site (position, switch_id)
length ()
sequence ()
sequence (first, last)
relative (absolute position)
appariated_sequence ()
complementary (position)
1
SequenceOccupation
commands
add_element (first, last)
remove_element (first, last)
add_sequence (quantity)
remove_sequence (quantity)
watch_site (binding site)
start_segment (position)
extend_segment (strand_id, position)
accessors
number_sites (first, last)
number_available_sites (first, last)
partial_strands ()
0..n
SiteGroup
commands
add (binding_site)
absorb (other_group)
update (a,b)
update_all
accessors
first()
last()
0..n
PartialStrand
commands
occupy (position)
accessors
completed()
segments()
occupation (bound units)
segments (partial strands)
position along master sequence
#available = #instances + #segments - occupation

Figure 21: `ChemicalSequence` represents a pool of polymeres that can be elongated and on which `BoundUnits` bind through `BindingSites`. 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 `BoundUnits` and elongation of `PartialStrands`. `SiteGroup` is used to notify sites of availability changes more efficiently.

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).

`DoubleStrand`

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

`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

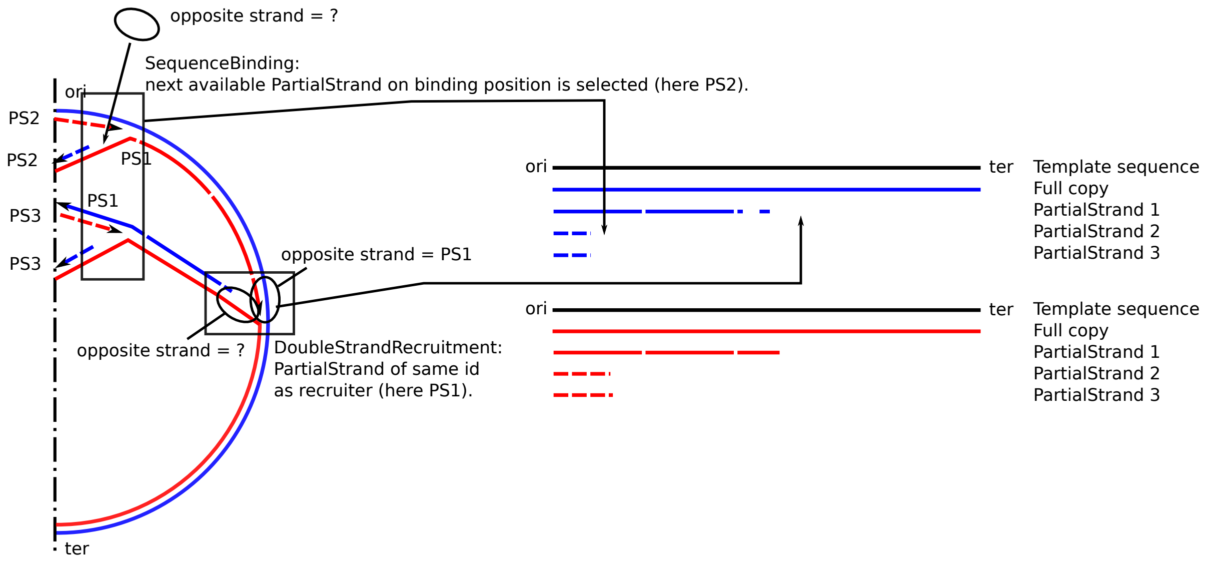


Figure 22: Strands of a `DoubleStrand` are identified according to creation order. Every time a new segment is polymerized, it is necessary to determine which `PartialStrand` is elongated. If a polymerase has been recruited on the complementary strand by `DoubleStrandRecruitment`, it is automatically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

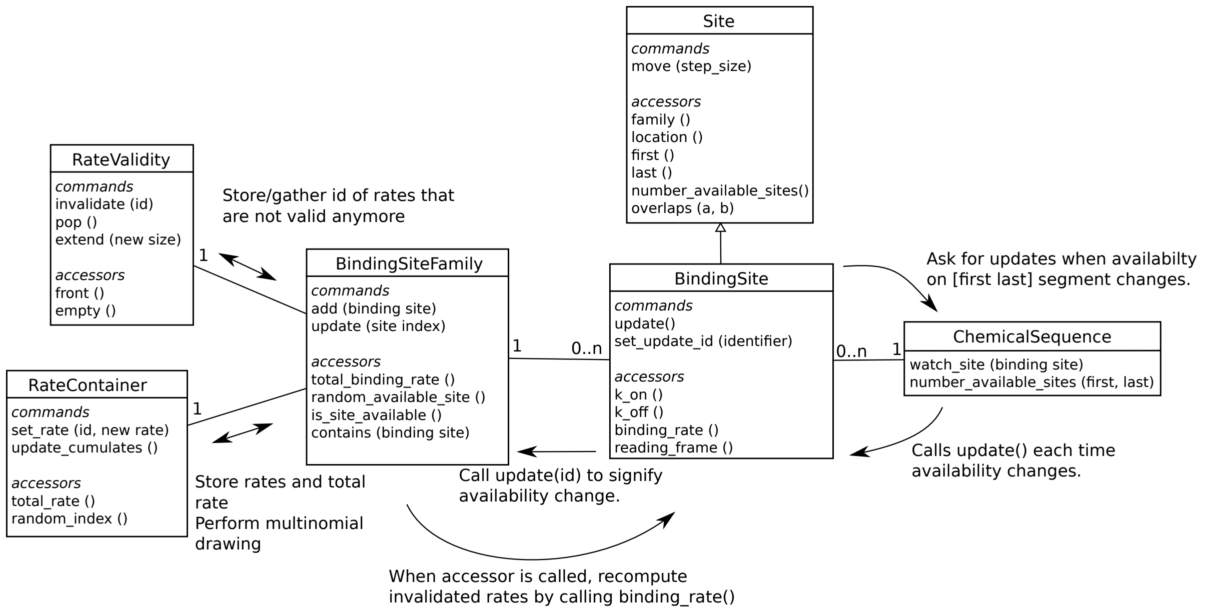


Figure 23: Schematical view of the Observer pattern used to keep availability of binding sites up to date for `SequenceBinding` reactions.

site notifies its `BindingSiteFamily` using a specific identifier, letting the family know 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.

2.4.2 Reactions

`ChemicalReaction` Nothing particular.

`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 unbiding.

`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.

`Loading`

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 `BoundUnits` reading a specifing 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).

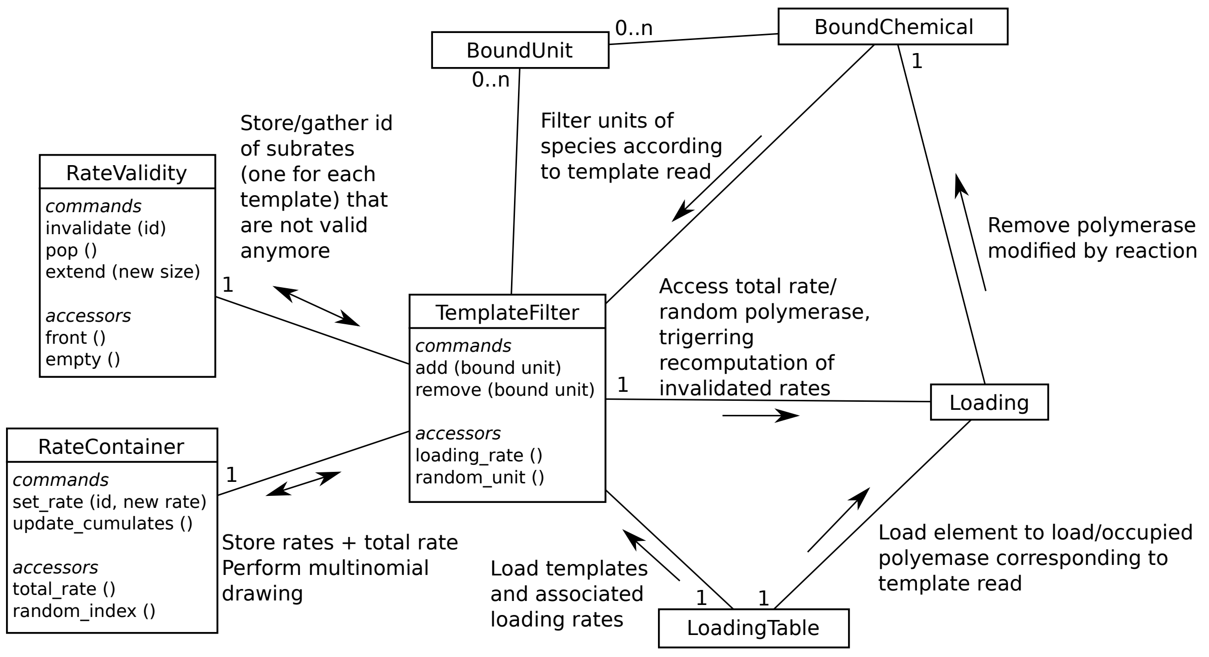

BoundUnit
0..n
0..n
BoundChemical
1
Store/gather id of subrates (one for each template) that are not valid anymore
Filter units of species according to template read
Remove polymerase modified by reaction
RateValidity
commands
invalidate (id)
pop ()
extend (new size)
accessors
front ()
empty ()
1
TemplateFilter
commands
add (bound unit)
remove (bound unit)
accessors
loading_rate ()
random_unit ()
Access total rate/ random polymerase, trigerring recomputation of invalidated rates
1
Loading
RateContainer
commands
set_rate (id, new rate)
update_cumulates ()
accessors
total_rate ()
random_index ()
1
Store rates + total rate
Perform multinomial drawing
Load templates and associated loading rates
1
1
LoadingTable
Load element to load/occupied polyemase corresponding to template read

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

`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.

`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 sythesizing step (*e.g.* `ProductLoading`) so synthesis will resume until the next termination site is reached.

2.4.3 Solver loop

Here we describe the implementations provided for each step of the algorithm. Most of the details are explained in a side document (?). We only give a quick overview here.

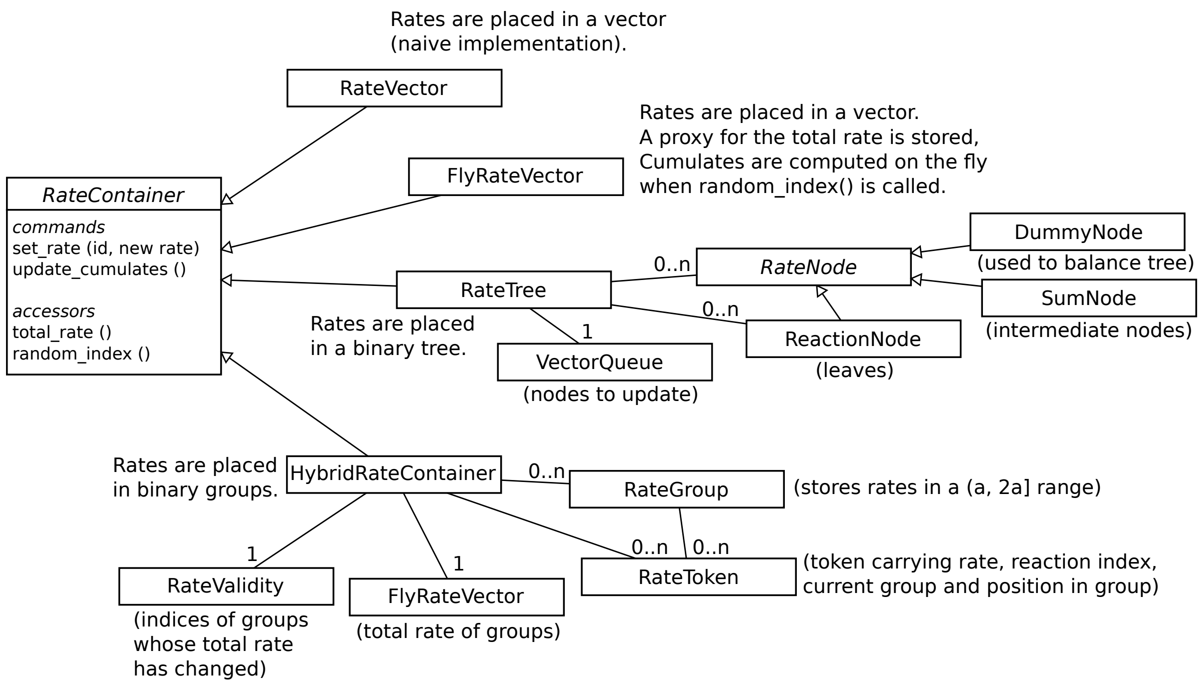

Rates are placed in a vector (naive implementation).
RateVector
Rates are placed in a vector. A proxy for the total rate is stored, Cumulates are computed on the fly when random_index() is called.
FlyRateVector
RateContainer
commands
set_rate (id, new rate)
update_cumulates ()
accessors
total_rate ()
random_index ()
DummyNode
(used to balance tree)
RateNode
0..n
RateTree
0..n
SumNode
(intermediate nodes)
Rates are placed in a binary tree.
1
VectorQueue
(nodes to update)
ReactionNode
(leaves)
Rates are placed in binary groups.
HybridRateContainer
0..n
RateGroup
(stores rates in a (a, 2a] range)
0..n
0..n
RateToken
(token carrying rate, reaction index, current group and position in group)
1
RateValidity
(indices of groups whose total rate has changed)
1
FlyRateVector
(total rate of groups)

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

`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 propopose several implementation of the algorithm (Fig. 25). Comparison and description of theses classes are given in ?.

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.

Perspectives These classes are the most sensitive classes from a numerical point of view. Stability of implementation should be checked more thouroughly (postconditions and or unit tests). `HybridRateContainer` needs a parameter to work. It is user provided for the moment but I think it should be determined automatically, probably by extending the group structure dynamically.

`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).

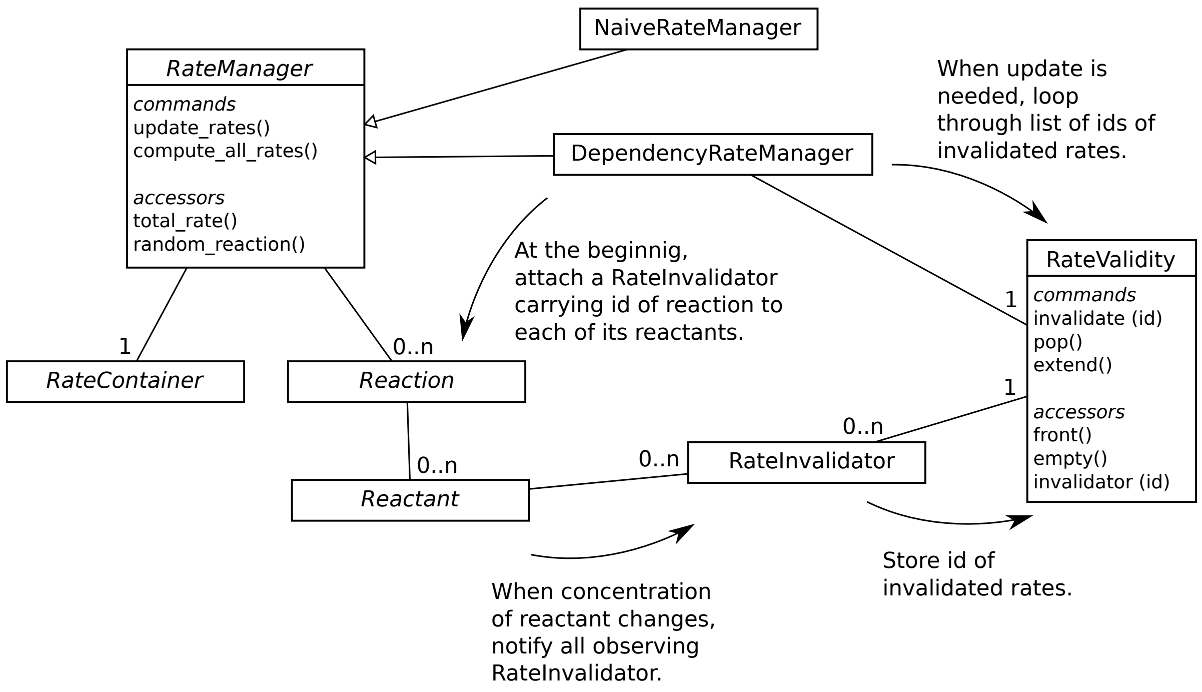


Figure 26: Implementations provided to update reaction rates. Note that the drawing part of the algorithm is always delegated to a `RateContainer`. `DependencyRateManager` uses an Observer pattern to monitor which rates have changed.

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 **?** for further details).

Perspectives While `DependencyRateManager` performs fairly well in the general case. It is particularily inefficient if there is a molecule A involved in a lot of reactions because a lot of rates have to be recomputed. It could be interesting to pool the reactions involving A into a `RateContainer` of their own, the latter containing only the contribution to the rate of the *other* reactants. The total rate of reactions involving A would then be the total rate of the container multiplied by the concentration of A. This would be viewed by the system as *one* mega reaction. When the concentration of A changes, virtually *nothing* is recomputed, except the total rate of the mega reaction (we suppose the contribution of other reactants has remained constant). If the mega reaction is to be performed, the `RateContainer` is used to draw which reaction will actually be performed according to their contributions. This change is not possible with the current architecture, the definition of `RateManager` and the computation of rates would have to be changed.

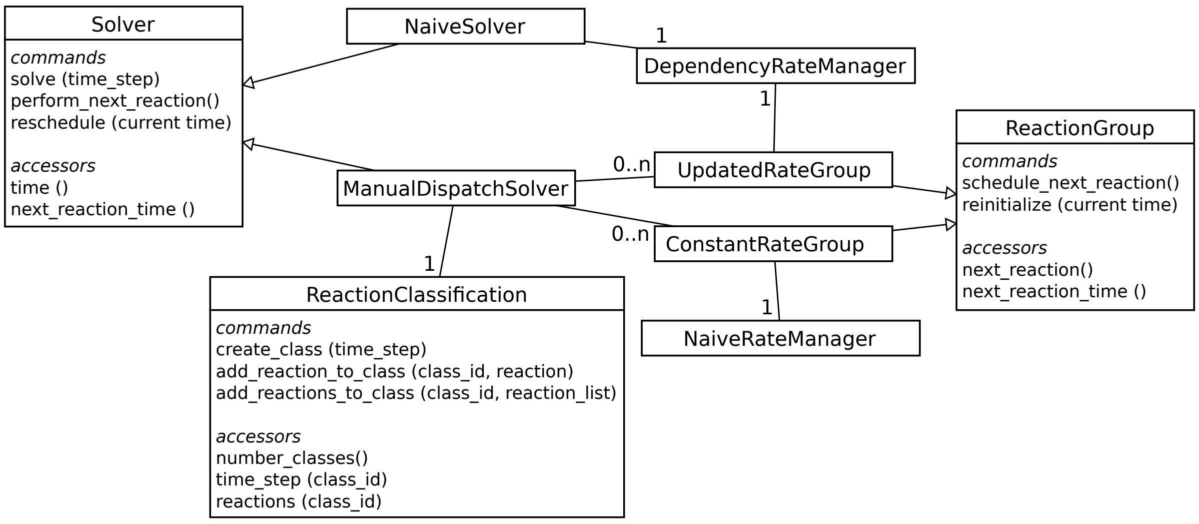


Figure 27: Two implementations of the `Solver` class organizing rate updating. `NaiveSolver` forces recomputation of rates at each time step. `ManualDispatchSolver` puts reactions into groups: reactions in `UpdatedRateGroup` are updated after every reaction while those in `ConstantRateGroup` only at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

`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, were 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 inavailability.

Perspectives There is no user interface to `ManualDispatchSolver` so it is not really possible to use it in practice witout changing the program. It would probably be more interesting to implement a variant or several variants of **tau-leaping**, which automatically assigns time steps to reaction to avoid reactant shortage. This has to be done carefully, the variant chosen must be effective event if the number of a chemical becomes or remains fairly low.

2.4.4 Input/Output handling

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`

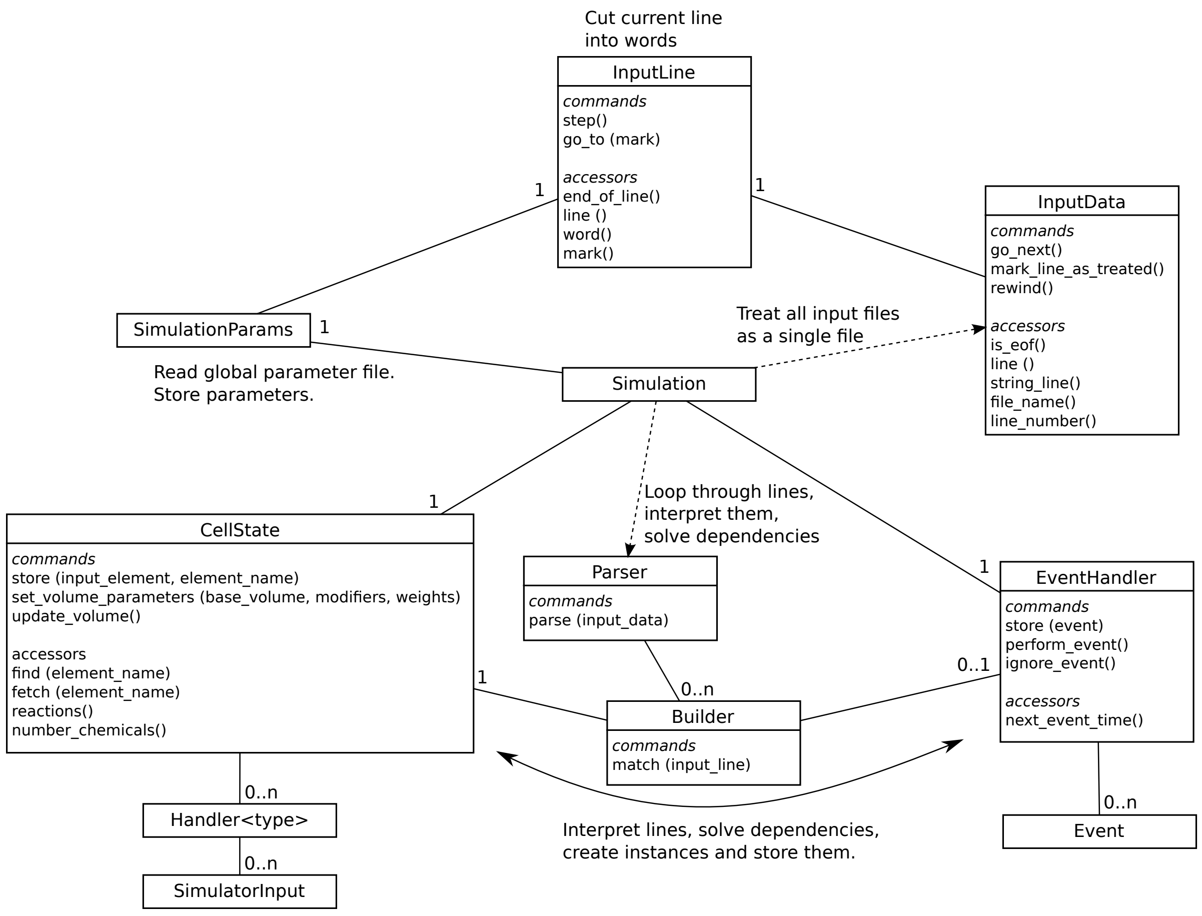
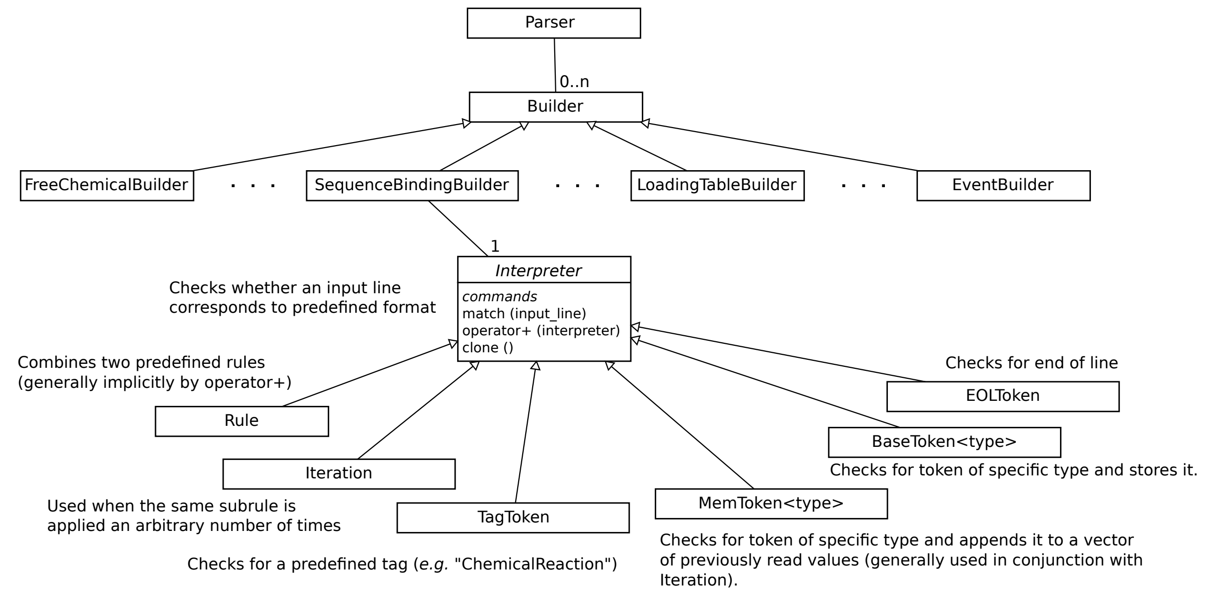


Figure 28: Architecture of the parsing system. `SimulationParams` reads the global parameter file and stores parameter values. `InputData` is used to provide a simplified interface to the input files and mark treated lines. A `Parser` and `Builders` are used to make sense of individual lines. Everything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

stores events. At the moment, an *ad hoc* input format is used.

Perspectives Review `SimulatorInput` arborescence which seems uselessly complicated. Design an architecture that can handle multiple input formats (plain text or XML).

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.


Parser
0..n
Builder
FreeChemicalBuilder
. . .
SequenceBindingBuilder
. . .
LoadingTableBuilder
. . .
EventBuilder
1
Interpreter
commands
match (input_line)
operator+ (interpreter)
clone ()
Checks whether an input line corresponds to predefined format
Combines two predefined rules (generally implicitly by operator+)
Rule
Iteration
Used when the same subrule is applied an arbitrary number of times
TagToken
Checks for a predefined tag (e.g. "ChemicalReaction")
MemToken<type>
Checks for token of specific type and appends it to a vector of previously read values (generally used in conjunction with Iteration).
BaseToken<type>
Checks for token of specific type and stores it.
EOLToken
Checks for end of line

example:
TagToken("SequenceBinding") + BaseToken<string> (unit_to_bind) + BaseToken<string> (bound_unit) + BaseToken<string> (binding_site_family)

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.

Perspectives Maybe create a `SimulatorToken` to automatically fetch the reference associated to a name? However, this has to be done with care because the right exceptions have to be raised.

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

Perspectives A `ReactionLogger` would actually be nice...

2.5 Utility classes

2.5.1 Exceptions

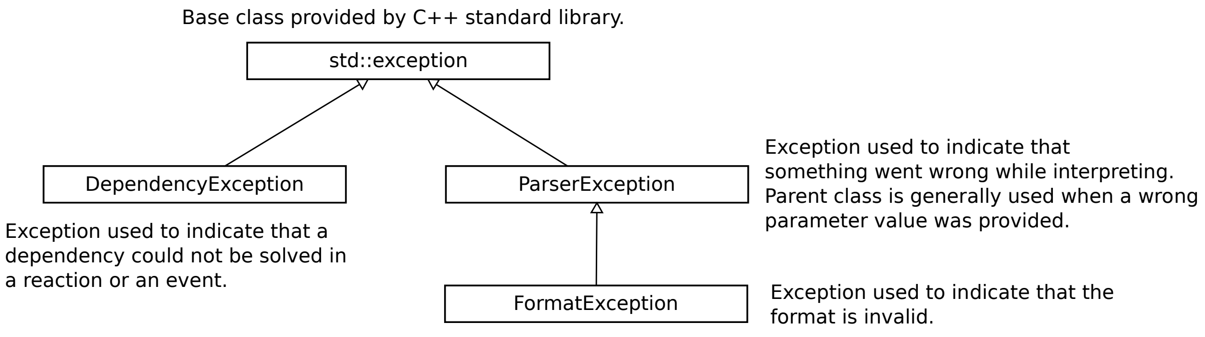

Base class provided by C++ standard library.
std::exception
DependencyException
ParserException
Exception used to indicate that something went wrong while interpreting. Parent class is generally used when a wrong parameter value was provided.
Exception used to indicate that a dependency could not be solved in a reaction or an event.
FormatException
Exception used to indicate that the format is invalid.

Figure 30: Exceptions used in the simulator.

Programming by contract (see Section 2.6) covers most internal errors that might happen. Exceptions are only used when user input is treated. They are used to signify inconstencies in input files during the parsing step (Fig. 30).

Why are exceptions useful? The nice thing about exceptions is that they can be caught and treated at a level that makes sense. In the parsing system, a format error is discovered by an `Interpreter`, which has no idea where the input line comes from and could not display a useful error message on its own. It is caught at the `Parser` level, enriched with information about input file, input line and so on, before being displayed.

Perspectives Errors occurring during simulation are displayed without using exceptions and suffer the bias described above. If a `Release` fails, there is a low level message such as “Unknown product”. We could throw an exception and catch it at the `Simulation` level, then use `CellState` to get the name of the `ChemicalSequence`, the `ProductTable` and the polymerase involved in the `Release`, providing useful information for the user.

2.5.2 Random handler

A unique `RandomHandler` is used throughout the simulation to control the random seed by using a Singleton pattern (Fig. 31).

2.5.3 Factories

Factories are used to remember user options and handle memory more efficiently (Fig. 32).

Why are factories useful? First, we have a context problem. For example, `SimulationParams` knows what kind of `RateContainer` the user wishes to use, but cannot instantiate it because it lacks parameters to instantiate it (the number of reactions in the system) and

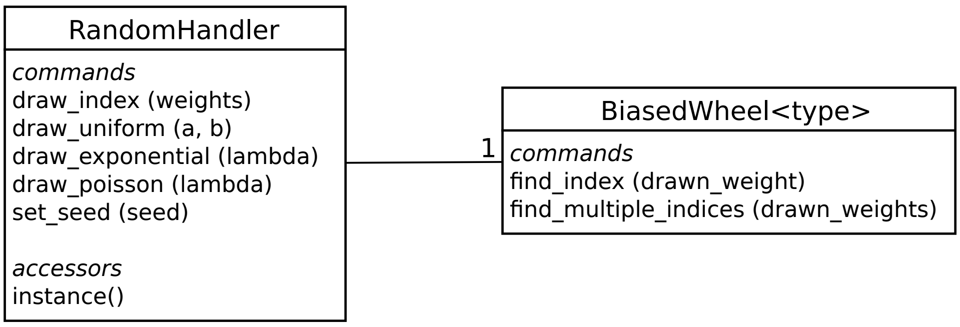


Figure 31: `RandomHandler` and `BiasedWheel` used for multinomial drawing. `RandomHandler` uses the Singleton pattern, meaning exactly one instance of the class is created and used throughout the simulator. It has to be accessed using `RandomHandler::instance()`.

would not know where to instantiate it. The naive way of doing would be recording a string saying for example `‘‘HybridRateContainer’’` then use a `if` structure at the right place to instantiate the correct `RateContainer`. *This is bad!* This means that every time a new `RateContainer` is added/removed, we have to track down every place in the program where a `RateContainer` is used and modify the `if` structure. In object-oriented programming we want to avoid the use of such structures because of such maintenance problem. This is done by using inheritance and factories. There is a `if` structure in `SimulationParams` to create the correct factory, this factory is passed as an abstract `RateContainerFactory` to client classes. Clients have no idea how many variants of the factory there are and they should not care, they just call `create()` on it, no `if`s involved. When a new `RateContainer` is added, we add a new child to `RateContainerFactory` and adapt the `if` structure in `SimulationParams`, *but literally nothing changes in the code of client class.* Factories = better maintenance (Abstract Factory pattern).

2.5.4 Vector-based containers

The simulator uses two non-standard containers, `VectorList` and `VectorQueue`. A typical example is a `BoundChemical` storing all its `BoundUnits`. 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

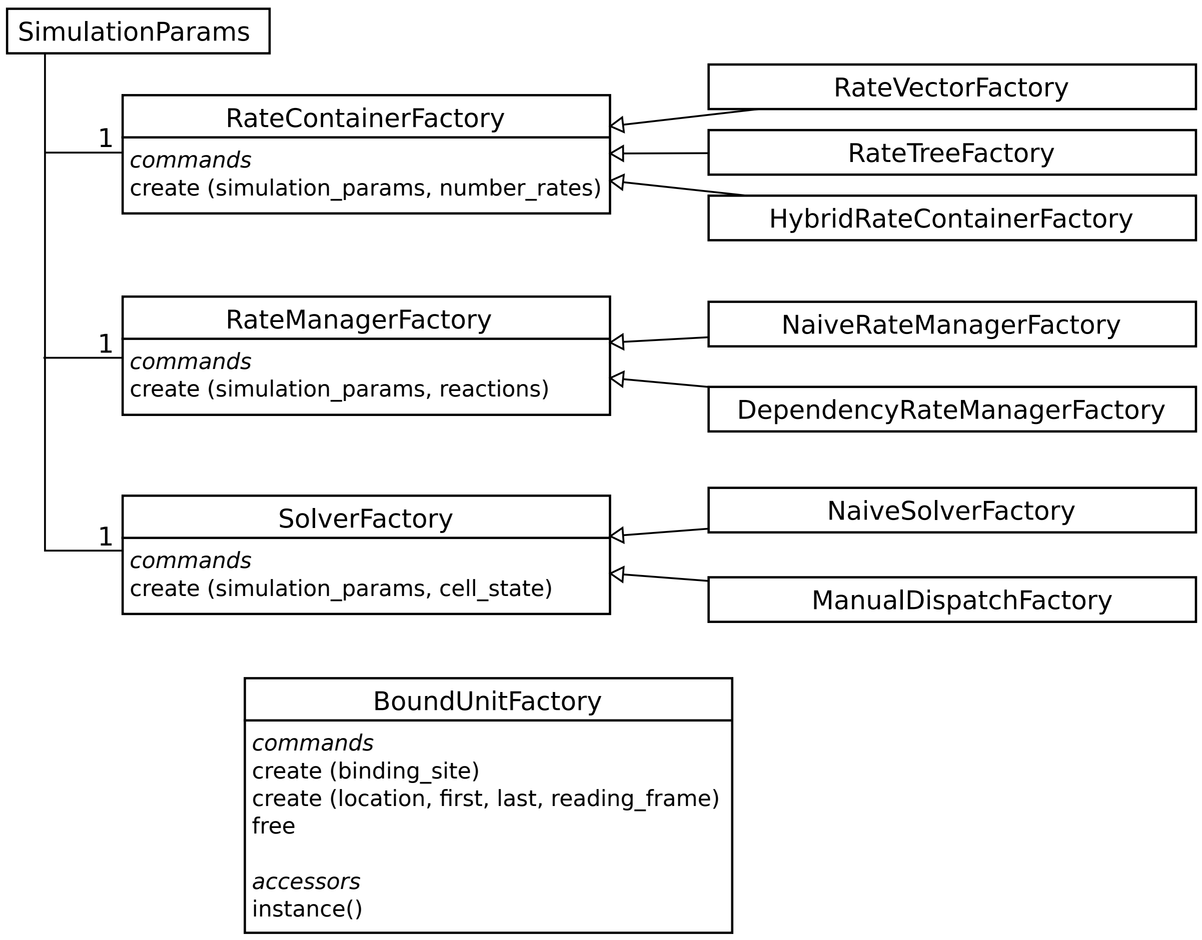


Figure 32: `RateContainerFactory`, `RateManagerFactory` and `SolverFactory` are used by `SimulationParams` to record what `RateContainer`, `RateManager` and `Solver` the user wishes to use (Abstract Factory Pattern). `BoundUnitFactory` is used to control memory usage by `BoundUnits`.It enables recycling of bound units, avoiding memory reallocation. It uses the Singleton pattern to make sure all instances of `BoundUnits` are stored at the same place.

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 contantly 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.

2.5.5 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 `Events` and `BoundUnits` 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 unsuscribed when they are destructed or the object they observe is destructed.

2.6 Tests

2.6.1 Testing philosophy

Fail fast Tests are designed to make the program fail as rapidly as possible and help find the origin of the problem.

Automated testing The only plausible way to fail fast is to automate testing. A framework must be designed where tests can be easily incorporated, activated/deactivated and run in a short time. Every time a piece of code is added, it should be easy to write tests for it and rerun all existing tests to detect simple mistakes as well as bad interactions. Without a real testing framework, there is a risk that some tests will be forgotten or that tests will be run more rarely. In the latter case, large bunches of code might have been added between tests and it will be hard to find out exactly what piece of added code lead to program failure.

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 1: Comparisons of tests used to develop the simulator

Testing layers Tests are usually divided in several layers. Because of the size of the project, the program includes three types of tests: *programming by contract*, *unit tests*, *integration tests* (Tab. 1). The lower the level of failure, the easier the debugging. Failing at the precondtion/postcondition level will give a very precise picture of what went wrong, while failing at the integration level might lead to a more thorough investigation. Ideally, we will always try to *fail faster*. If something went wrong at the integration level, we check whether this could have been detected already at the unit level or the precondition level. If yes, we implement these tests, and we know that next time the problem comes up, we will fail faster and solve the problem more rapidly.

2.6.2 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 theory, invariants are conditions on attributes that must always be true, from construction to destruction. They are checked after construction and every method call. In C++, we would have to implement these calls manually for all methods, which is a little tedious. In the simulator, we defined two macros `REQUIRE` and `ENSURE` to test pre- and postconditions.

```
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 33: 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 adressed in the code.

The example shown in Figure 33 shows typical uses of pre- and postconditions. With preconditions, we test that the user provides valid parameter values. This is the part of the contract the user has to respect. With postconditions, we check that return values or attribute values are valid. This is the part of the contract the class/method has to respect. As mentioned, invariants are rules that attributes must respect throughout the life cycle of the class instance (for `RateTree`, an invariant rule is that all nodes of the tree carry a value that is positive). Preconditions and postconditions are extremely useful for detecting simple typing mistakes, numerical issues (as in the example shown) and so on. Each time a precondition or a postcondition is broken, the program is interrupted and the condition that was broken is displayed (using `assert()`).

Why not use tests? Testing parameter validity and return value for every method of every class is very expensive. For the simulator, testing preconditions and postconditions nearly doubles simulation time. By using macros, we can activate/desactivate them during compilation (by using `./configure --enable-pre-check --enable-post-check`). In developpment phase, they are usually left on (except when performance is tested) and they are desactivated for real runs. Therefore, the programer should not worry about their cost and add them **for every. single. method**: our only goal is to fail fast!

Exceptions instead of preconditions? Exceptions and preconditions are used in different cases. *Preconditions* are used for internal interactions, when the *user is another class of the program.* In this case, the programer can actually control the input and make sure precondtions are true. *Exceptions* are used when the *user is an external user* and provides free input. In this case, the programer does not control input. However, bad input can be filtered using *exceptions* until all outside input complies with internal *preconditions.* In the simulator, exceptions are used extensively in the `Parser` and `Builder` classes that treat input files. But afterwards, when input is cleaned and the simulation is run, there are nearly no exceptions, everything is checked through preconditions.

2.6.3 Unit tests

Unit tests are probably the most famoust tests used. Unit tests generally test the behavior at the class level. There is a lot of documentaion on them, we used some simple guidelines to try and write useful and maintainable tests.

Test a use case scenario, not the class implementation This is the basis of test-driven developpement. The test should not care about how a class has been implemented, only what it has been designed to do. Usually a test is described by three elements: the method tested, the scenario tested and the expected output (example for `FreeChemical`: `add_addTenMolecules_numberIsTen`). If the class was designed correctly, it is likely that its interface will not change much, but its implementation may change over time. Use case tests will remain valid as long as the interface to the class is the same, no matter whether implementation changed. Implementation details should be tested using preconditions or postconditions.

Tests should be simple The aim is to find problems by failing. The simpler the tests, the easier the debugging. Several simple tests are better than a single complex test.

Tests should be independent The test framework should reset the class after every test to be sure that a test does not fail because of previous operations. Even if a test fails, the framework should run all other tests to detect as many problems as possible, yielding a better picture of the problem we are facing.

Use mock classes to test interaction with other classes If some input is needed or the output expected is some interaction with another class, mock classes should be used. For example, `SequenceOccupation` is responsible of notifying a `BindingSite` when its availability changes. The scenario is: when the availability of a site changes, the `update()` method of the `BindingSite` is called. In the simulator, this action has several implications: the `BindingSiteFamily` should be notified, the rate of possible associated `SequenceBinding` should be invalidated, etc. Therefore we could assess the interaction by testing whether the rate has been invalidated. However, this would be rather complex and hard to maintain if at some point this cascade is changed. The original intention was simply to test whether `SequenceOccupation` and `BindingSite` interact. The solution is fairly easy: we derive and use a child class `MockBindingSite` which overrides the `update()` and records whether it has been called.

2.6.4 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. Often, user input will be used and the scenario are very high level. Typical examples are:

- Check that we control the random seed correctly by testing whether the same input leads to exactly the same output.
- 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.

Tests should remain as simple as possible and easy to write. However, because they depend a lot on the global interface of the simulator, they are much harder to maintain. Some programs try to translate scenarios into a form adapted to the current interface to simplify maintainability, but we did not have the time to investigate the matter.

2.6.5 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. We did not implement a fancy infrastructure, 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 intergration 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).
- 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.

2.7 Perspectives

There are many perspectives left open by the project. Some have already been proposed throughout the document. Here is a brief overview of some others. The late introduction of `Switch`es actually does have a lot of impact on some of these perspectives, making them way more efficient as originally thought.

2.7.1 Handle partial polymerization products

For the moment, `PartialStrand`s are only used to represent nascent `DoubleStrand` products. The system could be extended for normal products of `ProductLoading` reactions. The two `Loading` reactions could maybe event be fused.

2.7.2 Collision handling

Upon `Translocation`, it could be easy to check whether a position on a `ChemicalSequence` is already occupied. The `Translocation` could then fail, but a more subtle behavior could also be implemented (a DNA polymerase ejecting a RNA polymerase). It is necessary to specify exactly what is wanted and what the user should input to make it work.

2.7.3 Product format, position table?

The way products are handled is somewhat tedious. It could be interesting to specify the binding site, the +1 of the product and the termination site all at once (in something like a `PositionTable`). Other elements could be added in the table for each product (pause regions, alternative terminators and so on). This could simplify existing reactions and be closer to the way biological data is actually stored. Anyway, the current system seems to be a little too obscur, something in that direction would be nice. From an implementation point of view, `Switch`es are great candidates to implement most of these elements (in conjunction with `BindingSite` for the binding part).

2.7.4 Less base reactions?

`ChemicalReaction` represents a lot of different reactions, what is done actually depends on the input. It could be nice continuing fusing other reacitons into `ChemicalReaction`. For example, the `Loading` reaction can be described as a reaction taking as an input a motif of a sequence, a polymerase and a free element and outputs an occupied polymerase. If we add a reactant type `SequenceMotif`, a `Loading` could be defined as multiple `ChemicalReaction`s which would replace the `LoadingTable`. This change is actually quite dangerous for performance reasons. Replacing a factorized reaction like `Loading` by multiple `ChemicalReaction`s is very expensive unless the latter reactions are automatically factorized. In other words, this would mean that for the use, only `ChemicalReaction`s are defined but, internally, the simulator would still factorize them as a `Loading` reaction. Before doing this change, we would have to make sure that it is actually more convenient for the user to only have `ChemicalReactions` (think of the ontology as being a user!!!).

Another way to replace loading would be using `Switch`es based on motifs. This approach could be better performance-wise and very easy to implement. (The previous paragraph was written before the introduction of `Switch`es, but it was kept for reference.) Anyway, before doing anything, we should compute exactly how performance would be impacted.

2.7.5 Serialization

An important missing feature is the ability to save the current state of the simulator and (a) resume it or (b) use it as an initial condition. This is important for example for `BoundChemical`, as they are originally no `BoundUnit` at the start of the simulation. C++ does not implement a way to do this natively, so a we need a design to make this automatically. I made a few tests with BOOST Serialization library which could be useful, as it automatically stores instances in a maintainable way. To make things easier, I think only a few classes need to be stored (`Handler`s, `BoundUnit`s for example), most can be reconstructed from scratch.

2.7.6 Known bugs

BOOST random library The simulator has been designed with an obsolete version of the BOOST random library. We need to implement two versions of `RandomHandler` depending on BOOST library version. We need the implementation for the current interface of the library!!!

Circularity of DNA For the moment all sequences are considered linear. When a translocation (*e.g.* of the replication fork) tries to go past the origin of replication, it fails with an `out of bounds` error.

Number of `DoubleStrand` Updates of the number of sequences is not done correctly. For instance, when a `PartialStrand` is completed, the carrying `ChemicalSequence` is not notified, it is not aware that there is a new sequence in the pool. The same goes for `DoubleStrand`, which does not know whether its strands have been completely replicated or not.

`DoubleStrand` strand identificaton Strand identification is used to know what `PartialStrands` correspond to each other on a `DoubleStrand`. Once a `PartialStrand` is completed, its id is freed and can be reaffected. However, it is possible that its counterpart is not completely replicated, so there is a small window where two strands might be mismatched because the id was reaffected to a newly created strand too rapidly.

2.8 Formats and Conventions

2.8.1 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*).

2.8.2 UML

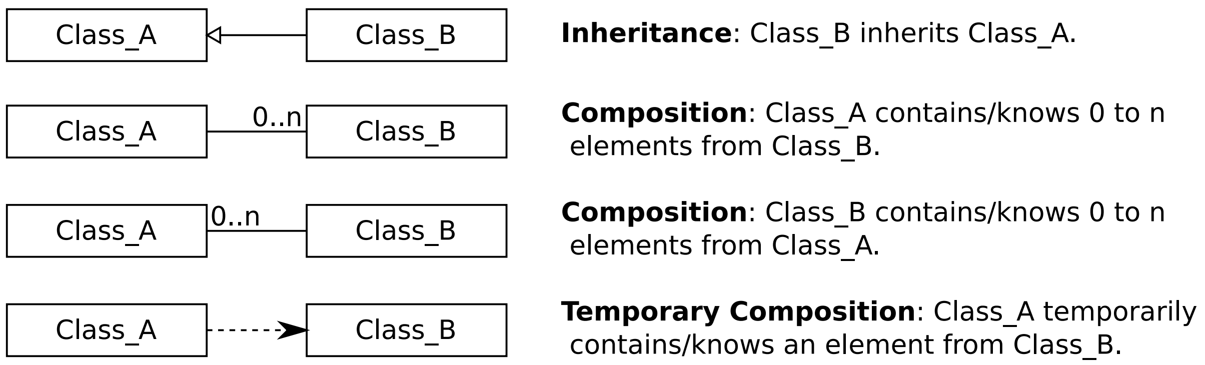

Class_A
Class_B
Inheritance: Class_B inherits Class_A.
Class_A
0..n
Class_B
Composition: Class_A contains/knows 0 to n elements from Class_B.
Class_A
0..n
Class_B
Composition: Class_B contains/knows 0 to n elements from Class_A.
Class_A
Class_B
Temporary Composition: Class_A temporarily contains/knows an element from Class_B.

Figure 34: UML format used.

3 Implementation of Gillespie's Stochatic Simulation Algorithm

3.1 Introduction

Gillespie's algorithm was designed to simulate chemical reaction networks, even in conditions where there are only a few molecules. In this section we briefly present how it was conceived and implemented.

3.1.1 Principles underlying Gillespie's algorithm

We suppose we are given a set of chemical reactions that we can write, in general as

$$a_1A_1 + a_2A_2 + ... + a_rA_r \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} b_1B_1 + .. + b_pB_p$$

the simplest reaction being a complexation reaction of the form

$$A + B \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} C$$

Assumption 3.1 (Well-mixed medium)**.** The first important hypothesis is that the molecules of every chemical species are distributed uniformly across space at all times. Then the probability that a molecule of species A encounters a molecule of species B is proportional to n_An_B where n_A is the number of molecules of species A and n_B the number of molecules of species B. Note that the probability that a molecule of species A encounters another molecule of species A is proportional to $n_A(n_A - 1)$.

Assumption 3.2 (Low-order reactions)**.** The second hypothesis is that reactions represent low-order phenomena, *i.e.* they result uniquely of the concomitant encounter of all species involved (as opposed to a sequence of events implying one or the other species).

Definition 3.3 (Propensity)**.** The propensity of a reaction is the rate at which it occurs. In the simplest case

$$A + B \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} C$$

the propensity of the complexation reaction is $k_1n_An_B$ and the propensity of the decomplexation is $k_{-1}n_C$. In a more general reaction

$$a_1A_1 + a_2A_2 + ... + a_rA_r \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} b_1B_1 + .. + b_pB_p$$

verifying the two assumptions above, the propensity of the forward reaction is

$$r = k_1 \prod_{i=0}^{r} n_{A_i}(n_{A_i} - 1)...(n_{A_i} - a_i + 1)$$

Property 3.4 (Exponential distribution of reaction timing.). *Under the two assumptions above, it can be shown that the timing of reactions are distributed exponentially, where the parameter of the exponential distribution is the propensity of the reactions. Because the propensity depends on the number of molecules, the probability density evolves over time.*

Assumption 3.5 (Memorylessness of reaction network.). We suppose that in the system of reactions considered, the occurrences of reactions are determined only based on their propensity. Occurrences of reactions are not independent in the sense that a reaction may change the propensity of another reaction (if they have chemicals in common). However, a reaction may not facilitate another reaction in any other way than changing its propensity.

Property 3.6 (Next reaction in a network.). *Under the current assumptions, at any time, the next reaction timing is determined by the first reaction that fires. Because the next reaction timings of all reactions are given by exponential distributions, the next reaction timing is the minimum of all these exponential drawings. Mathematically the minimum of exponentially distributed random variables is still exponentially distributed. More precisely, suppose there are N reactions and let r_1, ..., r_N be their propensities, then the next reaction timing follows an exponential distribution with parameter $\sum_{i=0..N} r_i$ and the probability that the next reaction is reaction k is $r_k / \sum_{i=0..N} r_i$.*

3.1.2 The Stochastic Simulaton Algorithm (SSA) and its variants

Gillespie et al. (2013) propose a good review of the original algorithm and some of its variants. The original SSA algorithm that can be summarized as follows:

- STEP 1: Update propensity functions.
- 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. Drawing statistics have to follow the properties described above. In a sense, the SSA and its variants are said to be (statistically) exact.

3.1.3 Aim of this section

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 and compared them on a benchmark inspired by our research project. We provide some perspectives that could be worked on to improve the algorithms or the way they interact with the updating step.

3.2 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 $\{r_1, r_2, ..., r_N\}$ 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.

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

3.2.1 Direct method

Principle The first method that was used historically is staightforward 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. 35).

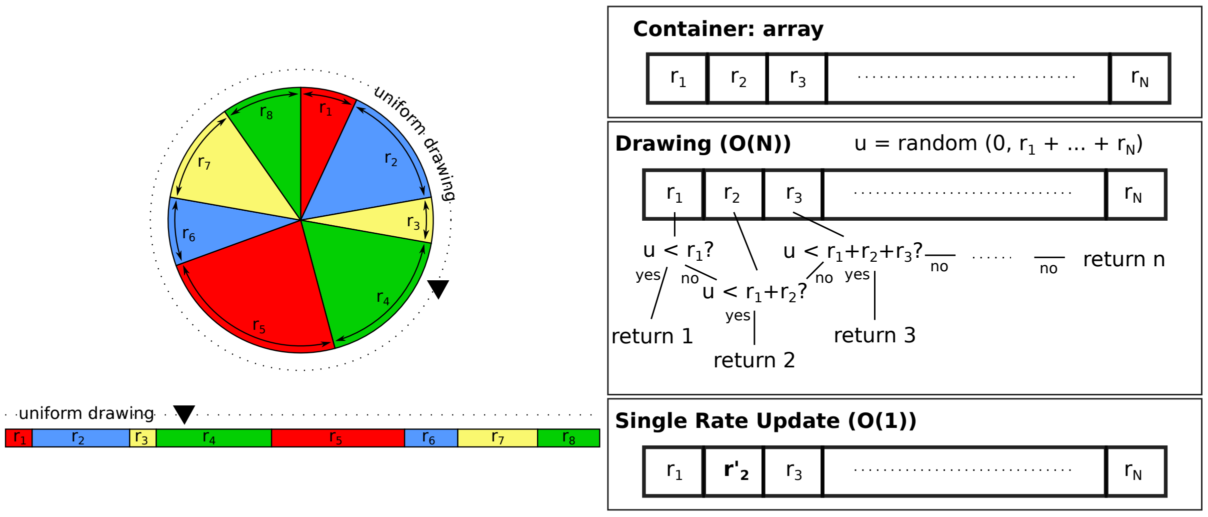
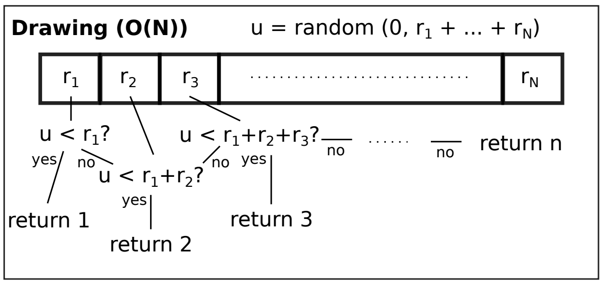


Figure 35: (Left) Illustration of drawing along a biased wheel and its equivalent representation as a segment. (Right) Container and algorithms used to maintain the structure.

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

Sketch of algorithm and complexity Worst case of the drawing (Fig. 36) occurs when u is in the last subsegment, so the loop has to be iterated N times, yielding $O(N)$ complexity.



```
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 $u \geq cum\_sum$ do
    index = index + 1;
    cum_sum = cum_sum + r [index];
end
return $index$
```

Figure 36: Direct drawing method

3.2.2 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 rougly) 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 furter details now. We will also omit it in complexity analysis, as it has the same worst-case complexity as the direct method. We will only reference it again in the Experiment section.

3.2.3 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. 37).

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 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

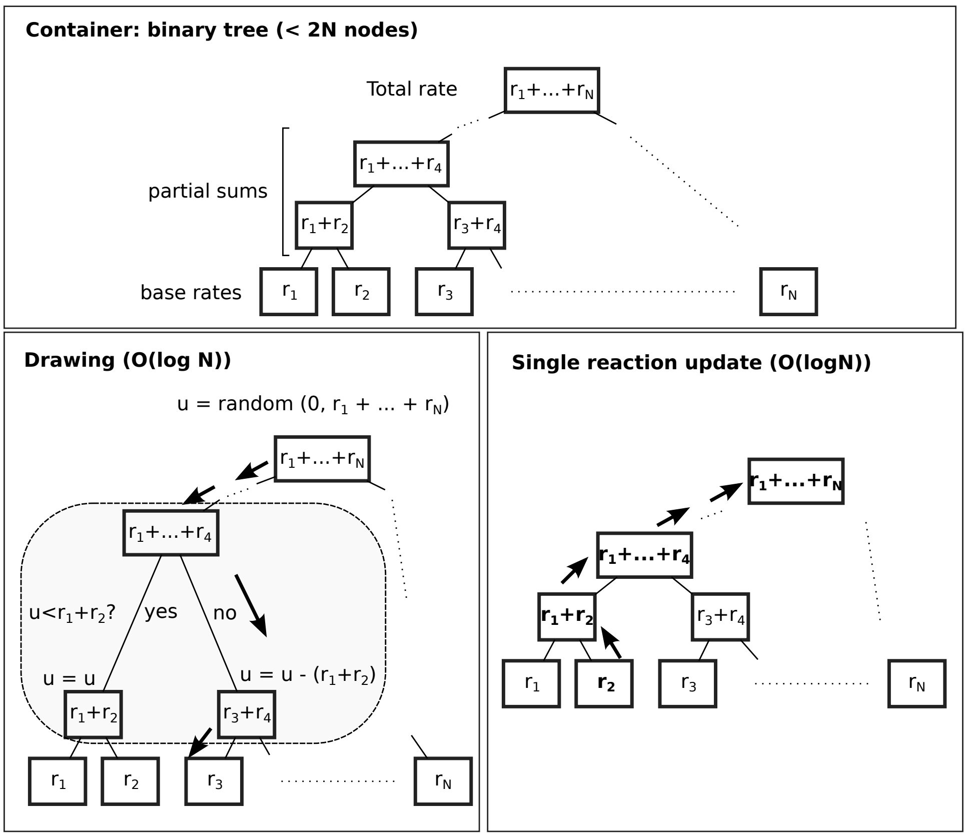


Figure 37: Binary tree containing propensities. Propensity values are found at the leaves of the tree. Intermediate nodes represent partial sums of nodes below, root holds the sum of all propensities.

some kind of progressive zooming in on the subsegments delimited by the propensity values (Fig. 37).

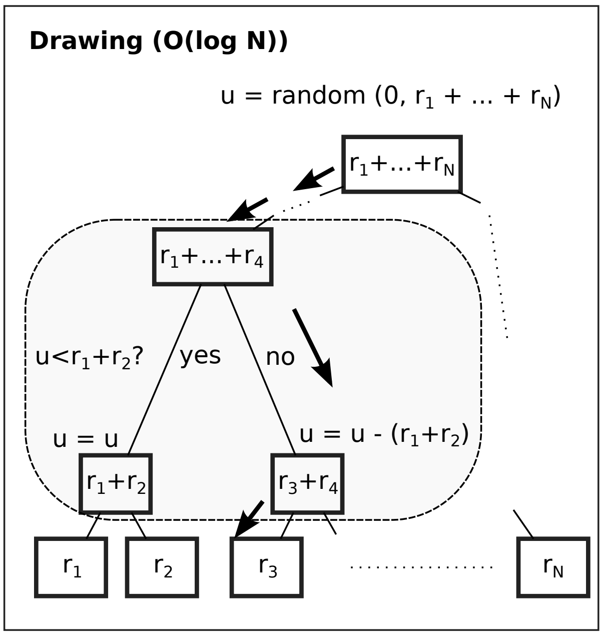
Sketch of algorithm and complexity Complexity for performing the drawing (Fig. 38) is given by the depth of the tree, $\lceil \log_2 N \rceil$, which is $O(\log N)$.

Complexity for updating the tree (Fig. 39) 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.

3.2.4 Hybrid method

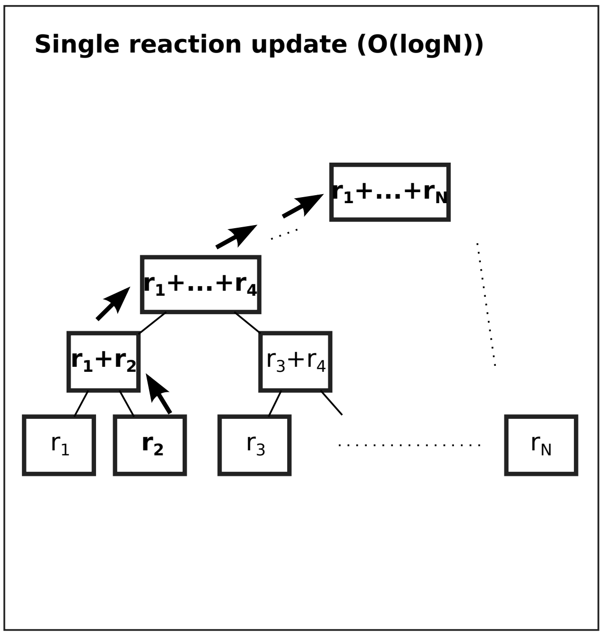
Principle In this approach, we organize propensities into groups and use a different drawing method: *rejection-base drawing.* This approach has been presented in Slepoy et al. (2008).

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



```
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 38: 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 39: Binary tree: update method.

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. 40A). 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. 40B).

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. 40C). Note that the number of groups K does not generally depend on N, it only depends on the highest propensity value. In general, it 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, we will see that 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 42 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. 43). However, the parts about removing or inserting a reaction into a group must be carefully implemented in order to achieve that result.

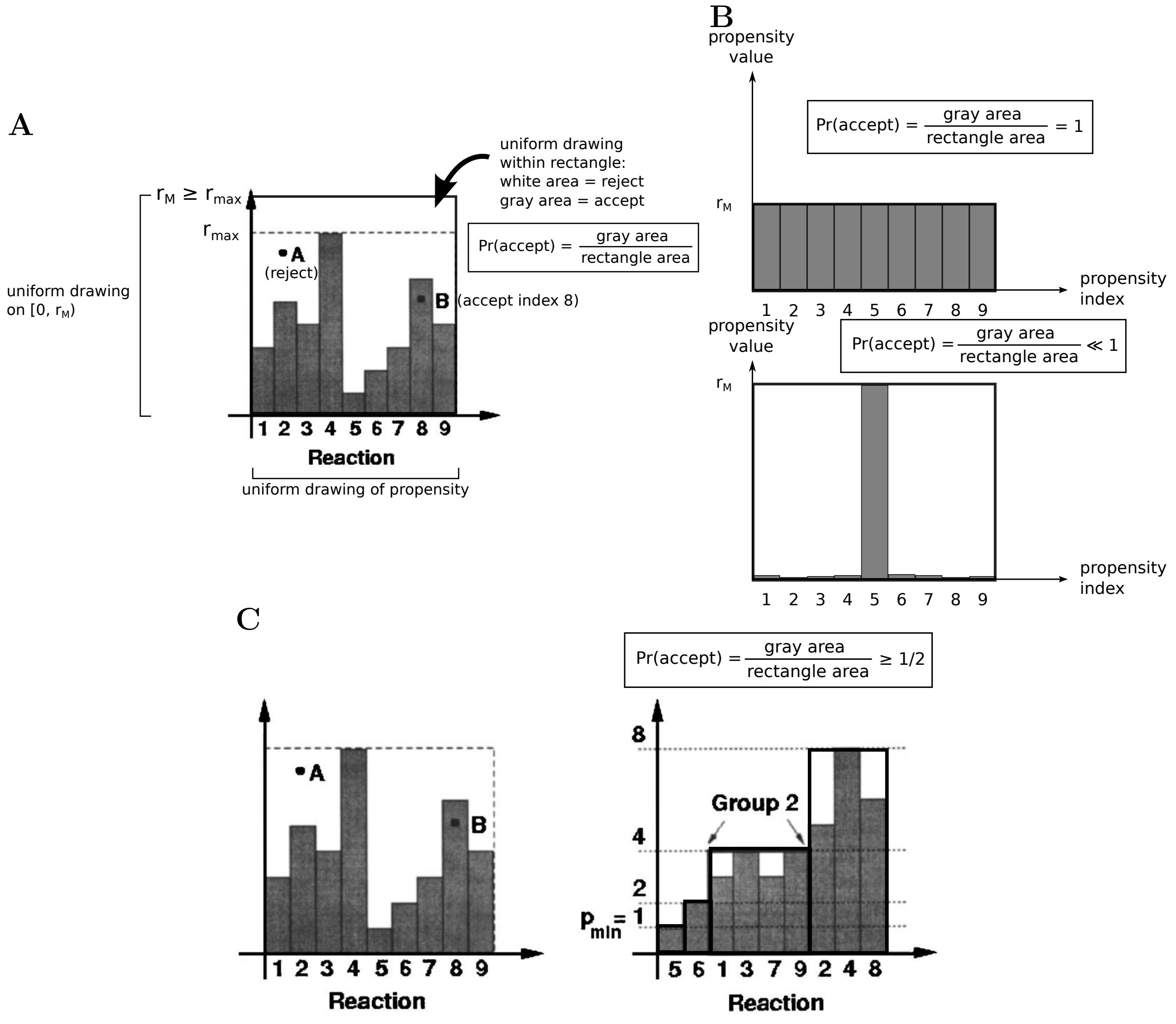

A
uniform drawing
within rectangle:
white area = reject
gray area = accept
$r_M \geq r_{max}$
r_{max}
A
(reject)
B (accept index 8)
uniform drawing
on $[0, r_M)$
$\text{Pr(accept)} = \frac{\text{gray area}}{\text{rectangle area}}$
1 2 3 4 5 6 7 8 9
Reaction
uniform drawing of propensity
B
propensity
value
$\text{Pr(accept)} = \frac{\text{gray area}}{\text{rectangle area}} = 1$
r_M
propensity
index
1 2 3 4 5 6 7 8 9
$\text{Pr(accept)} = \frac{\text{gray area}}{\text{rectangle area}} \ll 1$
C
$\text{Pr(accept)} = \frac{\text{gray area}}{\text{rectangle area}} \geq 1/2$
A
B
1 2 3 4 5 6 7 8 9
Reaction
8
4
2
$p_{min} = 1$
Group 2
5 6 1 3 7 9 2 4 8
Reaction

Figure 40: 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.*

3.2.5 Summary

Table 2 summarizes the worst case complexity of four methods presented. Note that the update complexity was derived in the case where onle one propensity needed to be updated. To obtain the overall complexity, we need to take into account the number of

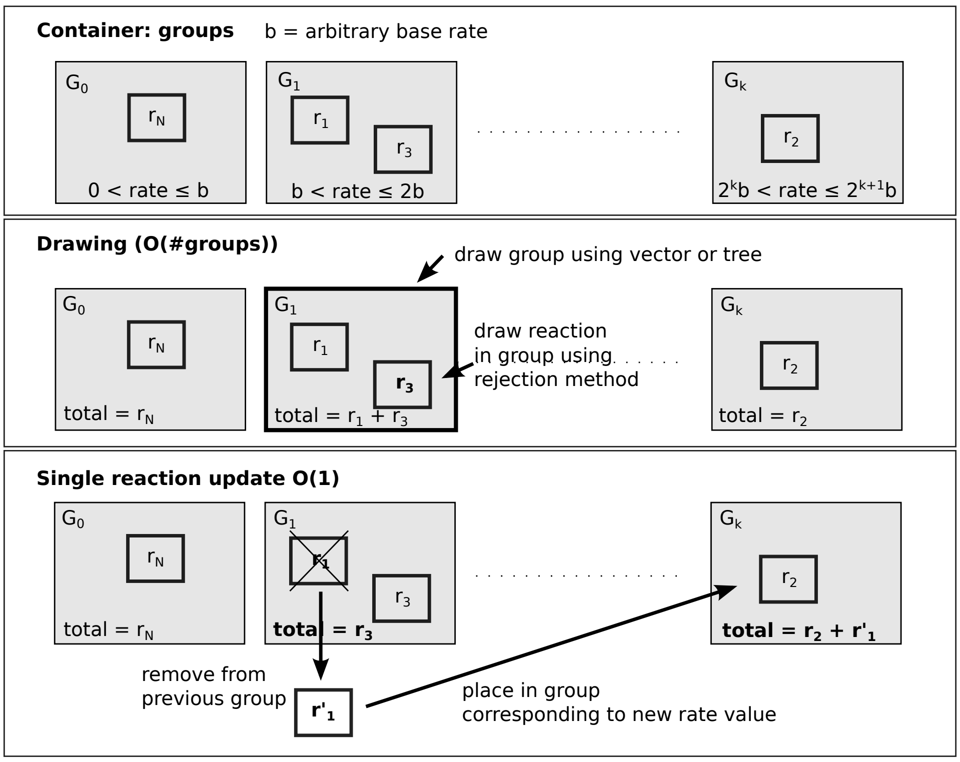
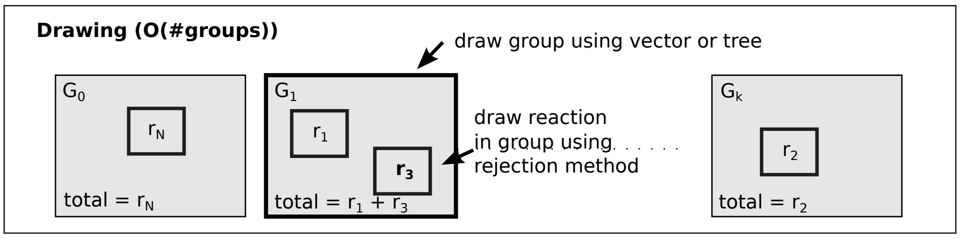
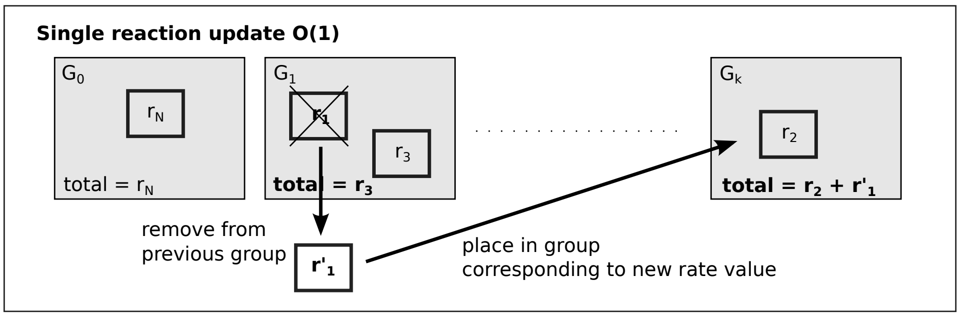


Figure 41: Hybrid method using group structure and rejection algorithm.


Drawing (O(#groups))
draw group using vector or tree
G0
rN
total = rN
G1
r1
r3
total = r1 + r3
draw reaction in group using rejection method
Gk
r2
total = r2

```
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 42: Hybrid method: drawing method.


Single reaction update O(1)
G0
rN
total = rN
G1
r1
r3
total = r3
Gk
r2
total = r2 + r'1
remove from previous group
r'1
place in group corresponding to new rate value

```
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 43: 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. 2). 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 2: 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.

3.3 Perspectives

Because the hybrid method leads to $O(1)$ complexity for drawing reactions, it is not possible to do substantially better on that part of the algorithm. Emphasis should be placed on winning time for updating reactions.

Selection reaction: tau-leaping The first idea to explore is abandon the exact framework and only update reactions once in a while by introducing time steps for updates. The problem with this method is that a reactant can run out during the time step and still be consumed, leading to negative chemical population. This has to be avoided at all costs. Automatical time step adjustments, termed **tau-leaping**, have been proposed to resolve this issue (Cao et al., 2005). However, these methods tend to be less efficient if there is constantly a reactant whose concentration is low, which would be the case in a whole-cell simulation scenario. It could be interesting to derive an algorithm that is able to classify reactions into sensitive (if they involve low-concentration reactants) and unsensitive. It would then assess each pool its own time step, with sensitive reacion rates being updated more often.

Propensity updating: factorizing reactions A situation were the algorithms we have used perform very poorly is when there is a reactant involved in nearly all reactions (hub in the reactant-reaction network). This means virtually every reaction will change the concentration of this reactant, and every reaction propensity will need to be updated, leading to poor performance (because structures underlying algorithms need to be completely rebuilt, as seen previously in the document). As propensities are of multiplicative nature, this is actually not necessary. Say all propensites in the system are of the form $[A] \times ...$, where $[A]$ is the concentration of the ubiquitous reactant. Then we could store $[A]$ on one side, and the remaining part of the propensites elsewhere. Updating would be easier, only $[A]$ would need to be updated. Drawing would only necessitate the second part of the propensity (renormalizing by $[A]$ does not change the drawing probabilities). This works even if not all reactions depend on A by pooling reactions that depend on a same reactant together. Pooling has to be done carefully to ensure the implementation remains efficient and statistically correct.

Implementation: parallel computing Using parallel computing can enhance performance. For example, in the case where updates are not a problem and propensities are updated constantly, generating random numbers takes up a large amount of time in the simulation (at least 25%). A first step would be using a node only for the purpose of computing and storing random numbers. Second, if updtating the system is a problem, this is a task that can be naturally parallelized. Concentration of reactants do not change during update, so there is a limited danger of data corruption by assigning updates to different nodes.

3.4 Implementation details

Implementation details fall into two categories. Optimization concerns ensure that implementations yield the claimed theoretical complexity. Numerical concerns address rounding problems that occur quite frequently in real-word situations.

When optimization is addressed, structures and algorithms are proposed that yield the asymptotic complexities presented in this document. Only rarely do we insist on optimizations that will only improve the multiplicative or additive constants of the algorithms. The reader is free to use its own adapted/further optimized structures.

Numerical issues are dangerous as algorithms ignoring them may lead to plausible, yet statistically flawed, results. They *must* be addressed at all costs. We propose simple ways to treat them. We do not doubt that more rigorous treatments exist.

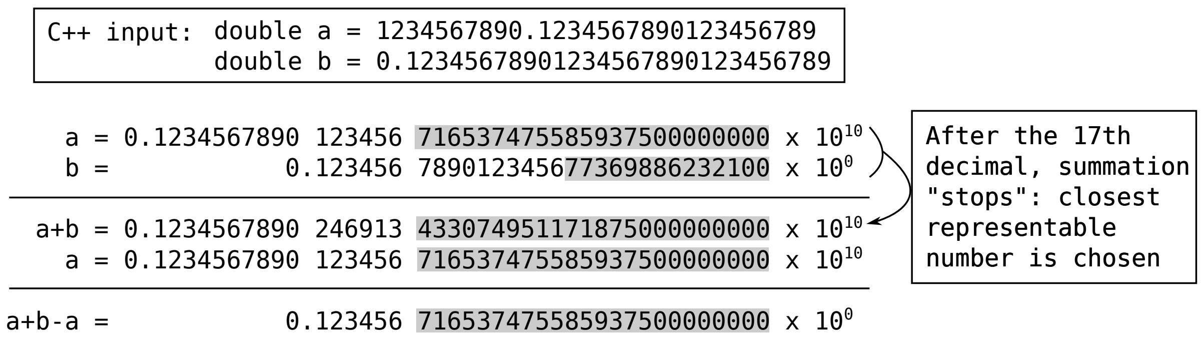
3.4.1 Numerical concern: updating total propensity on the fly

We start with numerical issues, as all methods are impacted by rounding problems. For illustrative purposes, we focus on the computation of the total of a set of real values.

Context Suppose we have a vector v of n real values. The task is to maintain the total value $T := \sum_{1 \le i \le n} v[i]$. Computing T anew every time a value changes is too expensive. Suppose we want to set $v[i]$ to a new value new_v_i. Theoretically, the new value of T will be $T - v[i] + new\_v\_i$.

Numerical issue: absorption On a computer, such operations will *always* lead to rounding problems. In other words, we must be aware that T will always be different from the real total. Question is: how different? Here we need to get a little technical and talk about the problem of *absorption*. Rougly speaking, a 64 bit `double` has a precision of 15 to 17 digits, the rest can be ignored. Take some number a. The next `double` a machine can represent is roughly $a + 10^{-15}a$. If we add b such that $b < 10^{-16}a$, rounding leads to $a + b = a$: *a absorbs b*. Now imagine $b \simeq 10^{-n}a$ where $1 \le n \le 15$. b is no longer absorbed by a, but its original precision will be lost in operations such as $b + a - a$. The first operation treated is $b + a$, where only $\simeq 16 - n$ digits of b that fall in the meaningful region of a are really kept (Fig. 44). When a is removed, only these digits are restored: b is left with only $16 - n$ meaningful digits. To track absorption, we need to know how the *largest* number treated in our operations compares to the current total. Let M be that number and T the total introduced in the context paragraph. The number of meaningful digits of T is $\simeq 16 - \lceil \log_{10} M - \log_{10} T \rceil$.

Solutions One possible solution is to switch to higher precision. 128 bit `quadruple` offer 33 to 36 meaningful digits, spanning a larger range of magnitudes. But even with larger precision, it may be necessary to track the current precision of T. This can be achieved by recording the largest number M used to compute T and use $d = 16 - \lceil \log_{10} M - \log_{10} T \rceil$ (replace 16 with 34 for `quadruple`) as a proxy for the number of meaningful digits. Because of rounding problems affecting meaningful digits, this is



```
C++ input: double a = 1234567890.1234567890123456789
           double b = 0.12345678901234567890123456789

     a = 0.1234567890 123456 71653747558593750000000 x 10^10
     b =            0.123456 789012345677369886232100 x 10^0
-------------------------------------------------------------
   a+b = 0.1234567890 246913 43307495117187500000000 x 10^10
     a = 0.1234567890 123456 71653747558593750000000 x 10^10
-------------------------------------------------------------
 a+b-a =            0.123456 71653747558593750000000 x 10^0
```

After the 17th decimal, summation "stops": closest representable number is chosen

Figure 44: Illustration of a rounding problem due to absorption. Gray areas show the decimals the computer has no real control of. They are chosen according to closest representation available. When summing terms, the sum can only be exact up to 17 decimals at most, so a part of the decimals of the smaller term of the sum are ignored and cannot be recovered.

rather an optimistic estimate. When d falls below a predefined limit (somewhere between 5 and 10), T has to be recomputed from scratch. This operation can also be dangerous in various ways (if v has a lot of small values and a large value, compensation of large numbers that have opposite sign). In practice, these dangers do not naturally arise in our case, but they can be treated by sorting the numbers by magnitude and carefully considering the order in which they are added up.

3.4.2 Direct Method

The only concern with this method is to update the total propensity properly (Sec. 3.4.1).

3.4.3 Binary Tree

Updating intermediate nodes at most once This optimization is pretty straightforward. The tree is updated starting from the highest depth. We suppose the tree is built so that all leaves are at the same depth. We use an update queue that initially contains all nodes whose propensities are outdated.

We follow this simple rule: every time a node is updated, its parent is added to the update queue. Each node holds an `outdated` flag. If a node is marked `outdated`, we know it is already in the queue and we do not add it a second time. The flag is removed when the node is updated.

By applying this procedure, nodes are added layer by layer. As mentionned above, we start with leaves at the highest depth. As we progress through leaves, there parents are added at their succession and so on, until root is finally reached. By using the flag system, each node is queued and updated at most once.

Note that the update queue will at most contain all nodes of the tree, *i.e.* $\simeq 2N$ nodes. To avoid memory reallocation, it can (should) be represented by an array of size $2N$ and two pointers to the start and end of the queue.

This procedure can be adapted if the leaves are not all at the same depth.

Numerical concern: rounding problems Because of rounding problems in the summation, it is possible (but extremely rare) that a node carrying a zero rate will be selected. This must be avoided at all cost because a reaction that is theoretically impossible will be performed, yielding unknown results. Such an event must be avoided or at least simply tested. Because this happens extremely rarely we did not try to avoid but simply perform a test. If a zero leaf is selected we know the drawing was supposed to fall in the area and simply take the next nonzero leaf.

3.4.4 Hybrid Method

Numerical concerns Because the total propensities of groups are stored and drawn according to a classical multinomial drawing method, it must be made sure that the total is updated properly, as stated in 3.4.1.

Efficient inserting/removal in groups

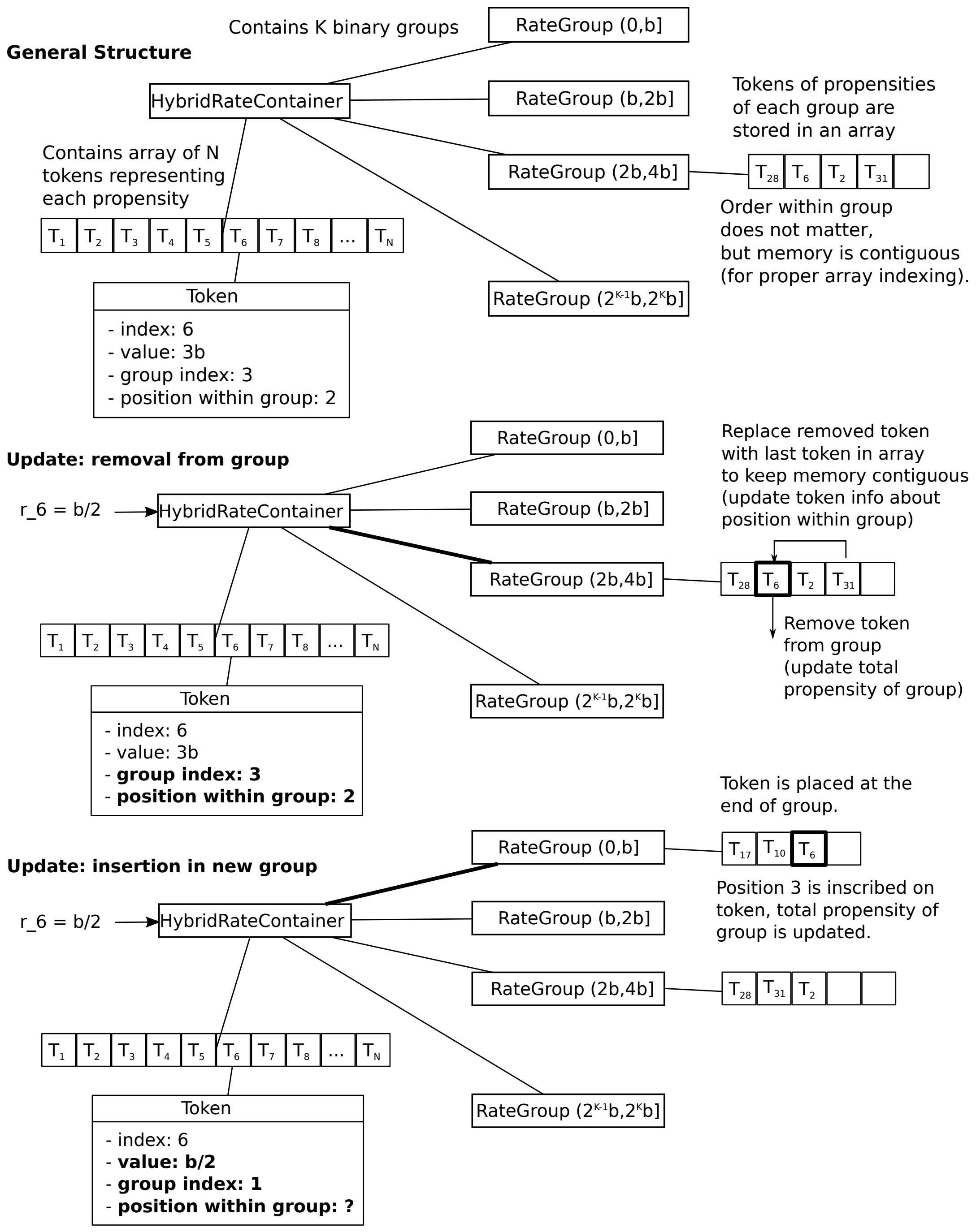
Principle In order to achieve $O(1)$ complexity, base operations of the algorithm have to be performed with care. Most importantly, no loop should be allowed in the algorithm. For example, if propensities were stored in a list and we needed to remove a propensity at some position in the list, we would need to loop through the list to find it, which could lead to $O(N)$ worst-case complexity. These are the points were loops must be avoided:

- Update: when looking for a propensity of index i, its group and location within group must be instantly found.
- Uniform drawing within group: any propensity must be accessed instantly (propensities have to be stored in an array structure).

Example of implementation Figure 45 shows an example of a design achieving optimal complexity using tokens to represent propensities. The costliest step is to determine the new group where to place the token. A logarithmic-like function has to be used, but if groups are delimited by powers of 2, more efficient low level functions can be used, such as `frexp` in C.

It is easy to design a similar structure without actual tokens, but in a language like C++, all these classes can be inlined and do not actually exist in the compiled code. If the design is careful enough, it can be both human-readable by using classes representing clear abstractions AND efficient.

Choice of base rate This is the trickiest part of the algorithm. The choice can be left to the user, but it is also possible to imagine a structure that adapts dynamically. We did not have time to put much thought into it, but some ideas would be



General Structure
Contains K binary groups
HybridRateContainer
RateGroup (0,b]
RateGroup (b,2b]
RateGroup (2b,4b]
RateGroup (2^{K-1}b,2^Kb]
Tokens of propensities of each group are stored in an array
T28 T6 T2 T31
Order within group does not matter, but memory is contiguous (for proper array indexing).
Contains array of N tokens representing each propensity
T1 T2 T3 T4 T5 T6 T7 T8 ... TN
Token
- index: 6
- value: 3b
- group index: 3
- position within group: 2
Update: removal from group
r_6 = b/2
Replace removed token with last token in array to keep memory contiguous (update token info about position within group)
Remove token from group (update total propensity of group)
Update: insertion in new group
Token is placed at the end of group.
T17 T10 T6
Position 3 is inscribed on token, total propensity of group is updated.
T28 T31 T2
- value: b/2
- group index: 1
- position within group: ?

Figure 45: Example of structure warranting optimal complexity. Propensities are stored in their `RateGroup` as `Token`s containing additional information, enabling $O(1)$ operations.

- Create a base group at some arbirary rate. Each time a propensity (or too many propensities) falls into that group, subdivide the group into new groups, thus lowering the base group rate.

- Replace the base group with any container adapted to multinomial drawing, possible a hybrid drawing structure. If the value drawn falls into that group, the drawing method of the structure chosen is applied by reusing the same value to avoid additionnal random generator costs.

In both cases, the question is: at what point is it worth creating subgroups/substructures? This question has to be carefully analyzed before a choice of implementation is actually made.

References

Yang Cao, Daniel T. Gillespie, and Linda R. Petzold. Avoiding negative populations in explicit Poisson tau-leaping. *J Chem Phys*, 123(5):054104, 2005. URL `http://scitation.aip.org/content/aip/journal/jcp/123/5/10.1063/1.1992473`.

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`.