MyBacteria design
Version 1.0

M. Dinh and S. Fischer

August 30, 2017

Contents

1. Introduction

The aim of this document is go present the general design of MyBacteria's implementation (the code is documented using Doxygen, technical details are therefore best found in the Doxygen-generated manual). We present the main classes used in the simulator and the concepts behind them. We also show how we tackled a certain number of efficiency issues (should it be speed or maintainability).

We start by listing the central principles that guided the development of MyBacteria. Then we present the base components of the simulator, organized around reactants and reactions. The following section describes the same element again, but goes further into hypotheses and critical design elements. Finally, appendices are added to describe elements that have been important in the simulator development but did not fit naturally in the main document (testing strategies, utility classes, etc.).

2. Design principles

In this section, we present the ideas that guided the development of MyBacteria.

2.1. General requirements for the simulator

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

- Simulate as many bacterial processes as possible.
- Being able to integrate various levels of description (coarse-grained processes and low level stochastic processes).
- Generic formalism, applicable to any bacteria.
- Efficient (not more than a couple of hours for one cell cycle).
- Easy to extend and reuse (either by extending the core or coupling with other modules).

2.2. Design choices

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

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

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

The standard Gillespie algorithm only implements chemical reactions. This does not meet our requirement of simulating a wide variety of processes. For example, it is extremely tedious (nearly impossible) to simulate translation accurately using only chemical reactions. We created a variant of the Gillespie algorithm where new types of reactants and reactions can be plugged in. We defined a minimal set of reactants and reactions that handles all sequence-based reactions (*e.g.* binding, translation elongation). All reactions remain low-level, enabling flexible descriptions of complex processes. A lot of information that is provided as an input for these reactions comes from standard sequence annotation. When switching from an organism to another, the key reactions that define the process remain the same. Only sequence information (DNA, position of genes, promoters, etc.) and rates need to be adated.

We evaluated performance by simulating protein production. Protein production (from gene to protein) is responsible for a large number of reactions in a bacterial cell (metabolism aside). Simulation is completed within hours even for detailed descriptions of all processes involved (stochastic base-by-base elongation with all cofactors). This objective was reached by using the latest implementations of the (exact) Gillespie algorithm. We also tuned all new types of reactions to be nearly as efficient as chemical reactions.

Finally, we created clear modules in MyBacteria's structure. This allows for core changes and facilitates communication with external modules. Typical core changes involve:

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

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

3. Global presentation of the components of the simulator

3.1. Components of the simulator

The simulator can be decomposed into several large modules that handle specific tasks during simulation (Fig. 1). First of all, there is an **input/output** module that creates

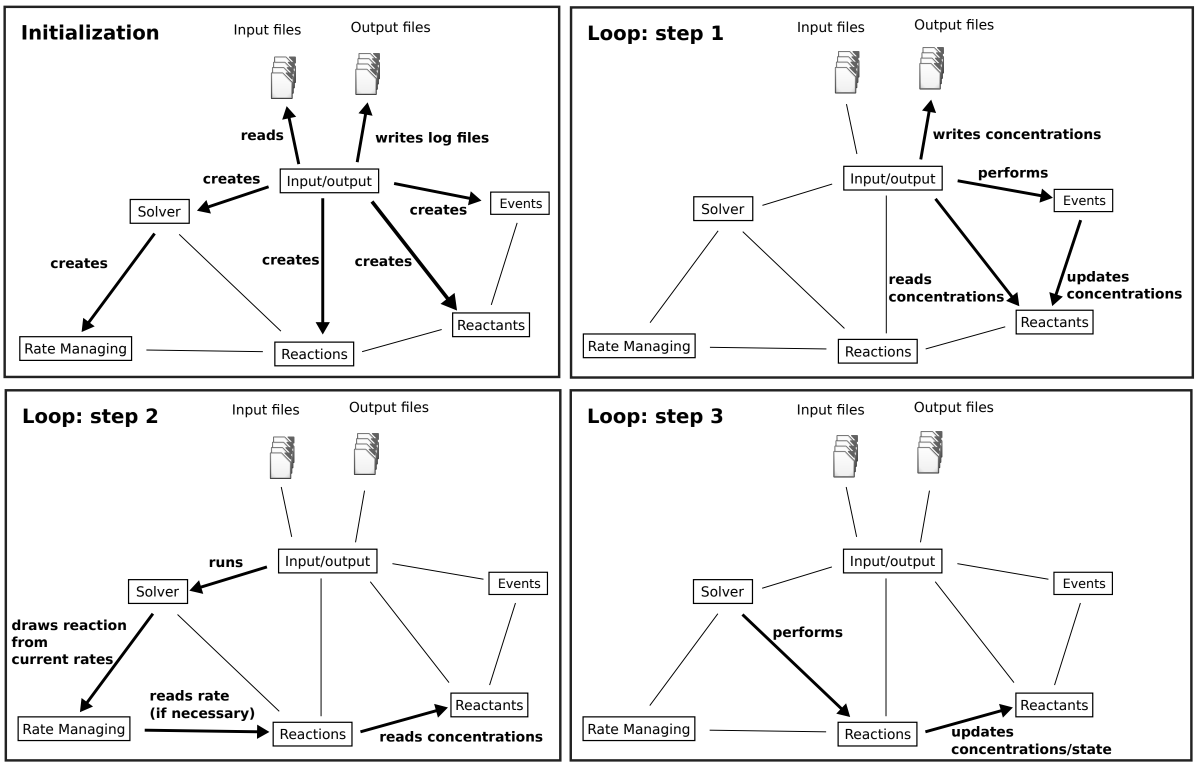


Figure 1: Schematical view of the simulator.

everything that is needed for the simulation from an input file. **Reactants** and **reactions** are user-specified and need to be created on demand, as well as **events** happening throughout the simulations and more technical aspects about which algorithm to use to perform the integration. Once everything is set up, the **solver** follows a simple loop that can be decomposed in three steps. Integration occurs reaction by reaction, at each loop, we go forward one reaction, update the simulation time, concentrations and reaction rates.

1. At the beginning of the loop, the **input/output** process checks whether **events** should occur at the current simulation time and whether it needs to write some concentrations to an output file.

2. It then hands control over to the **solver**, which is based on Gillespie's approach to integrate a network of chemical reactions. The Gillespie algorithm needs the current reaction rates of all **reactions** and draws a random reaction with a probability proportional to its rate. This task is delegated to a **rate manager**, which uses state-of-the-art methods to maintain the rate list updated and perform the drawing efficiently.

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

3.2. Reactant hierarchy

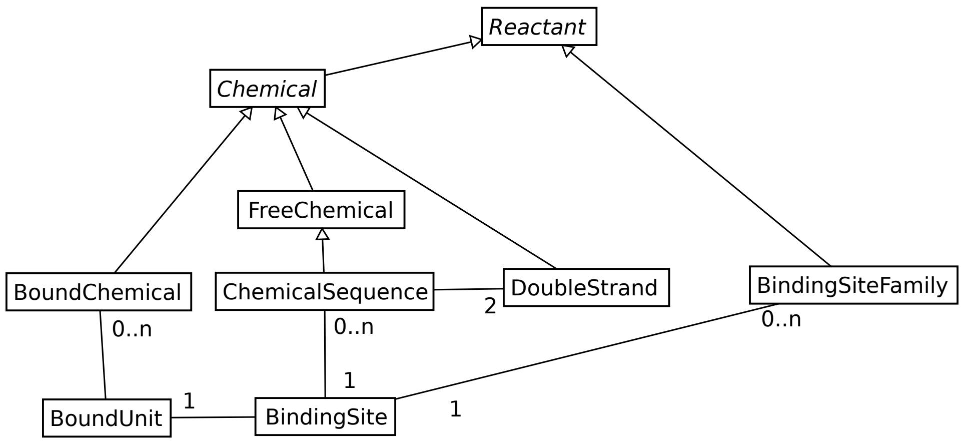


Figure 2: UML diagram of `Reactant` hierarchy

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

3.2.1. Reactant

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

3.2.2. Chemical

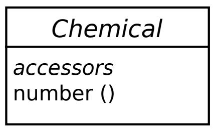


Figure 3: `Chemical` class

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

3.2.3. FreeChemical

Input format

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

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

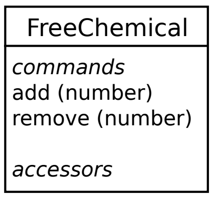


Figure 4: `FreeChemical` class

3.2.4. BoundChemical

Input format

```
BoundChemical <name>
```

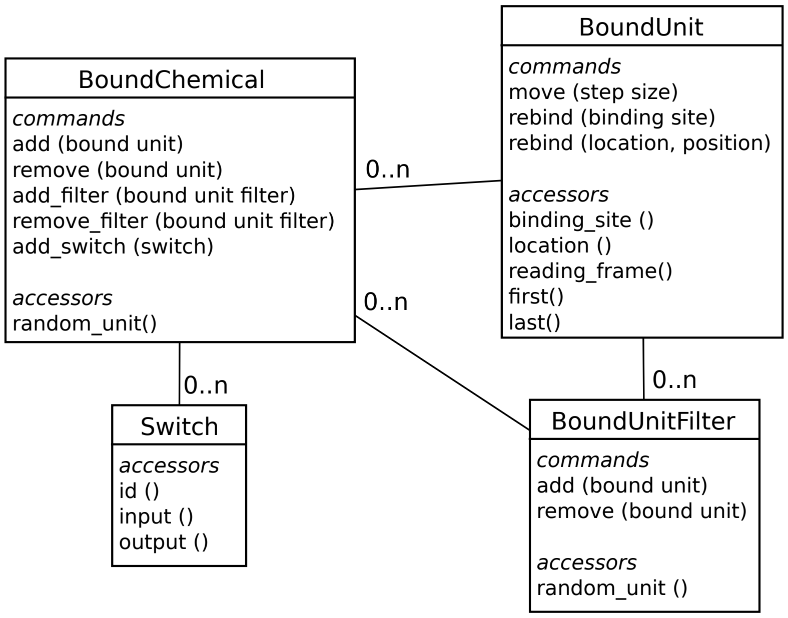


Figure 5: `BoundChemical` class

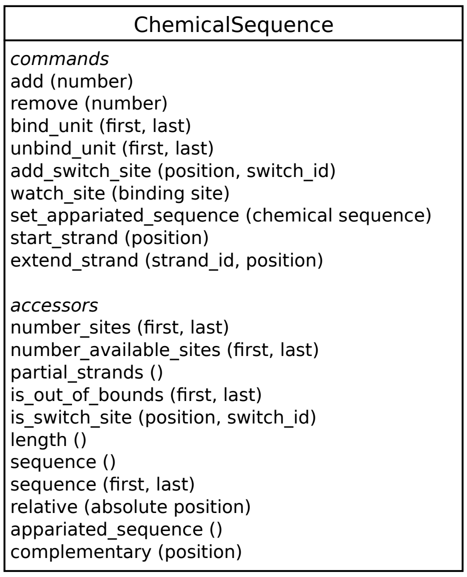
`BoundChemical` (Fig. 5) is a subclass of `Chemical` that represents chemicals that are bound to a sequence. Even though `BoundChemical` represents a pool of molecules, single elements are not interchangeable, they are defined by their position on a sequence. `BoundChemical` uses class `BoundUnit` to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

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

3.2.5. ChemicalSequence

Input format

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



ChemicalSequence

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

`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 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.

3.2.6. DoubleStrand

Input format

```
TransformationTable <name> [<letter> <complementary_letter>,]^{1..n}
```

```
DoubleStrandSequence <name> <name_sense_sequence> <sense_sequence> \
  <name_antisense_sequence> <transformation_table> [<initial quantity>]
```

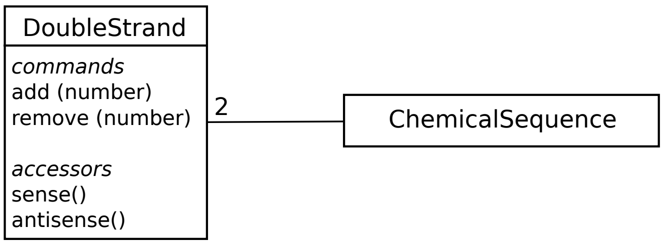


Figure 7: `DoubleStrand` class

`DoubleStrand` (Fig. 7) links two `ChemicalSequence` together that are biochemically linked (*e.g.* DNA), one sequence being complementary to the other. It enables segment extension on the 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, $A \to T$, $T \to A$, $C \to G$, $G \to C$).

3.2.7. BindingSiteFamily

Input format

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

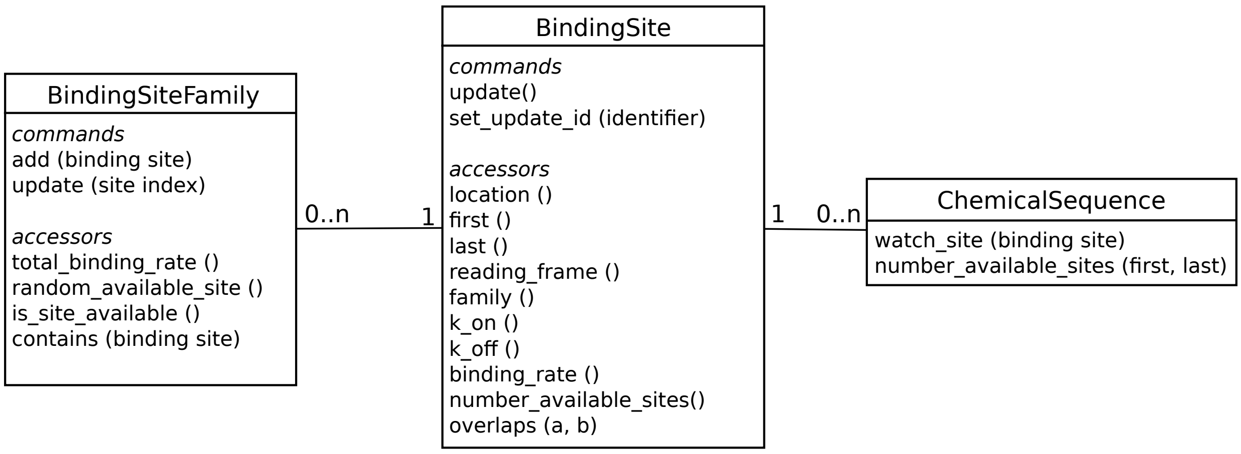


Figure 8: `BindingSiteFamily` class

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

3.3. Reaction hierarchy

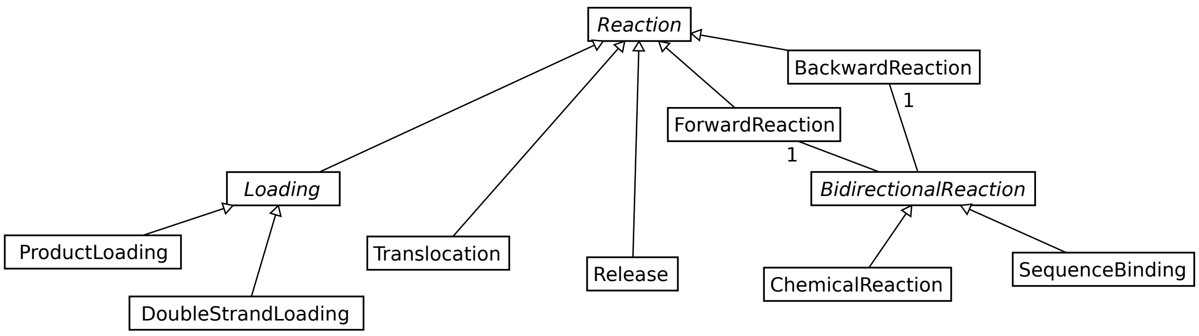


Figure 9: UML diagram of Reaction hierarchy.

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

3.3.1. Reaction

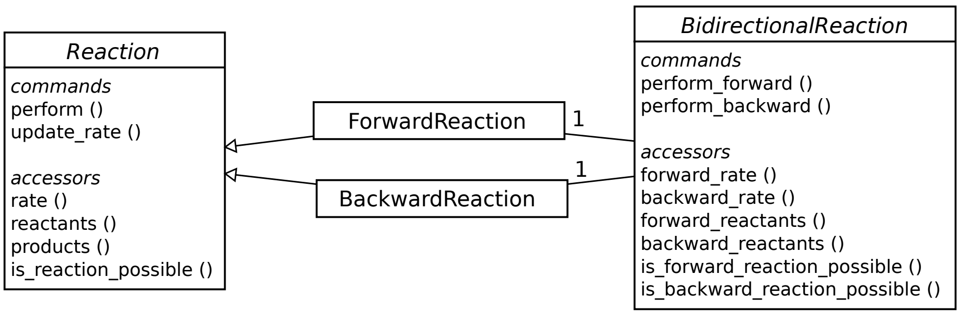


Figure 10: Reaction and BidirectionalReaction classes.

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

3.3.2. ChemicalReaction

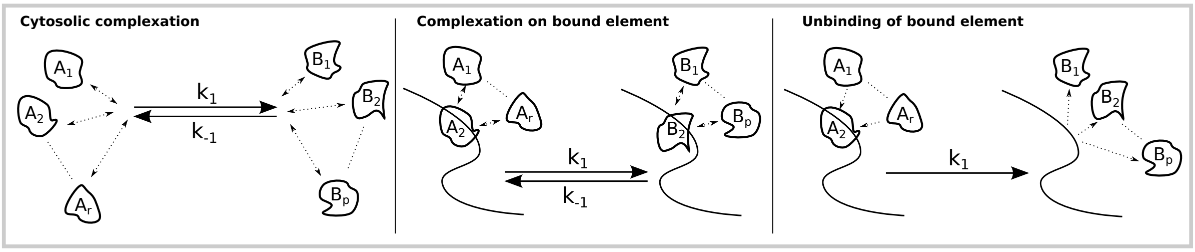


Figure 11: Schematic view of a `ChemicalReaction`.

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

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

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

Rate The rates are given by

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

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

3.3.3. SequenceBinding

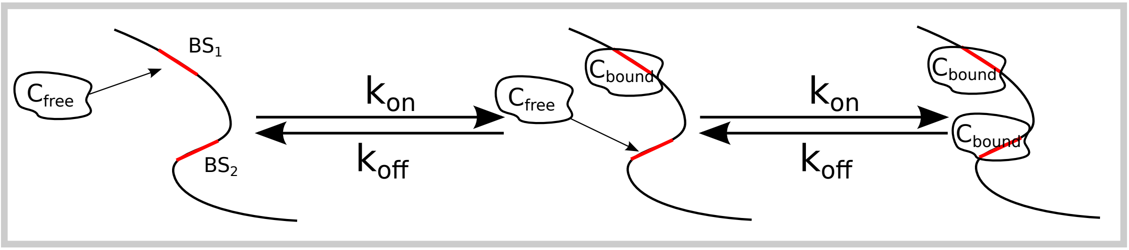


Figure 12: Schematic view of a `SequenceBinding`.

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

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

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

Rate The rates are given by

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

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

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

3.3.4. Translocation

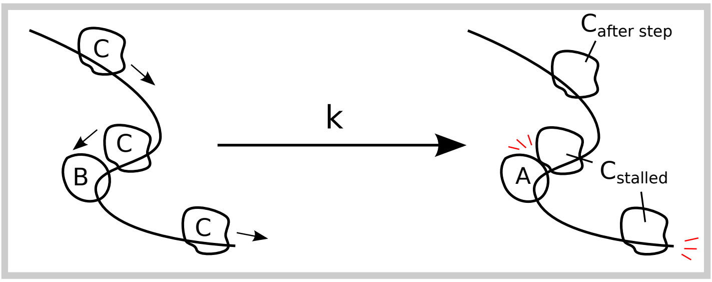


Figure 13: Schematic view of a `Translocation`.

Input format

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

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

$$\lambda = k[C]$$

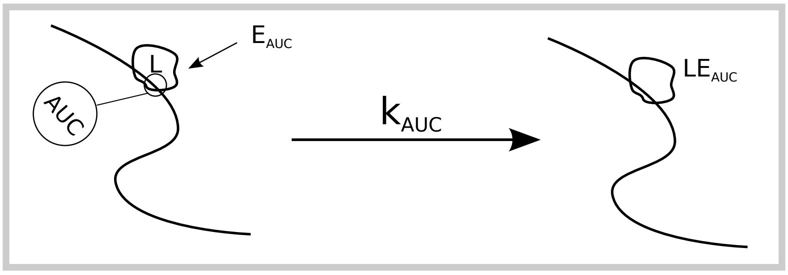


Figure 14: Schematic view of a `Loading`.

3.3.5. Loading

Input format

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

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.

3.3.6. DoubleStrandRecruitment

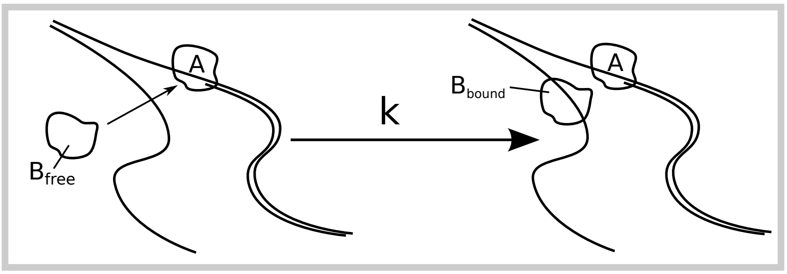


Figure 15: Schematic view of a `DoubleStrandRecruitment`.

Input format

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

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

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

where

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

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

Rate The rate is given by

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

3.3.7. Release

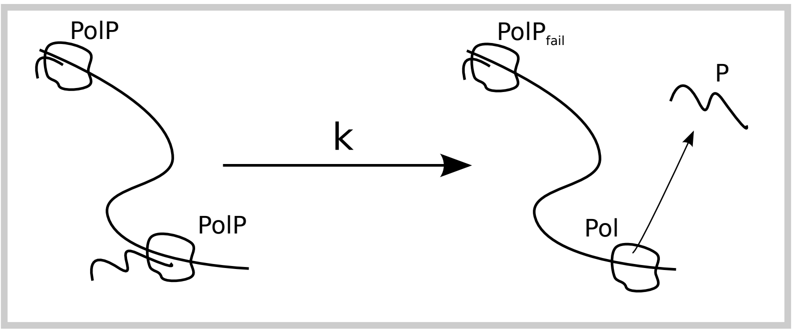


Figure 16: Schematic view of a `Release`.

Input format

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

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

$$\lambda = k[PolP]$$

3.3.8. Degradation

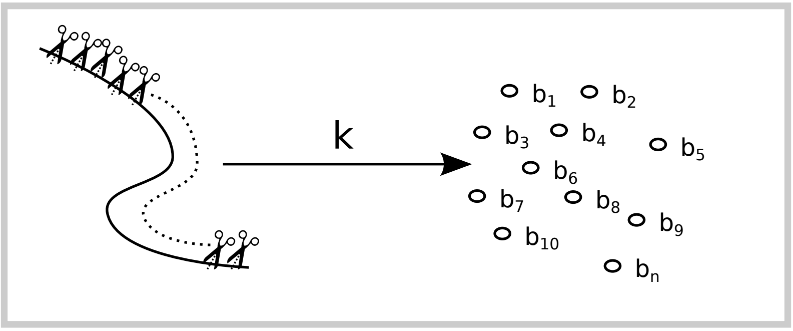


Figure 17: Schematic view of degradation reaction.

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

$$\lambda = k[CS]$$

3.4. 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`.

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

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

3.5. Solver loop

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

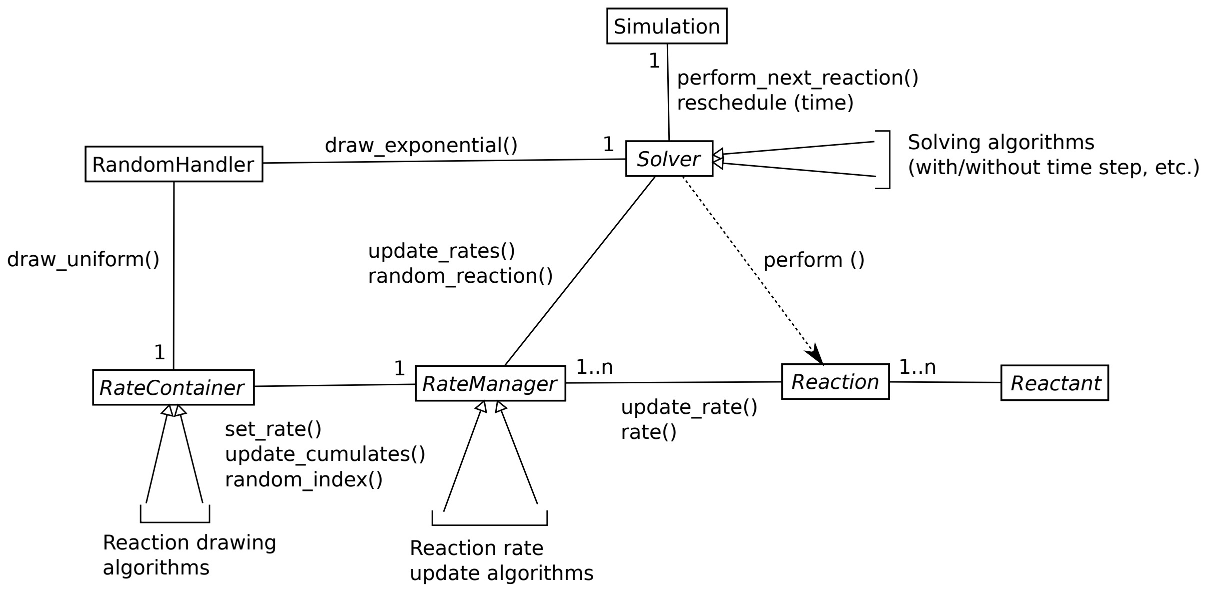


Figure 18: Solver loop. The loop is driven by the `Solver` class that defines how and when rates should be updated. The update task is performed by a `RateManager`. Once rates are known, multinomial drawing is delegated to a `RateContainer`. A central `RandomHandler` is used so that the solver only uses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

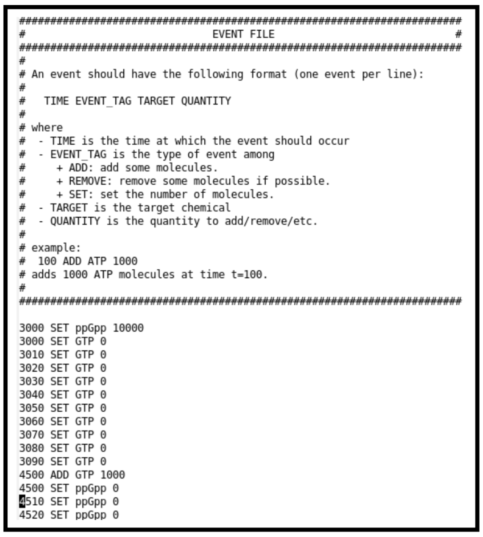
The solving loop is depicted in Figure 18. The Gillespie algorithm has many variants. We decided to implement it using three *abstract* classes. By using inheritance, variants can be combined for each step of the algorithm (how to update reactions, how to select a reaction). The three central classes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

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

3.6. 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.



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

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

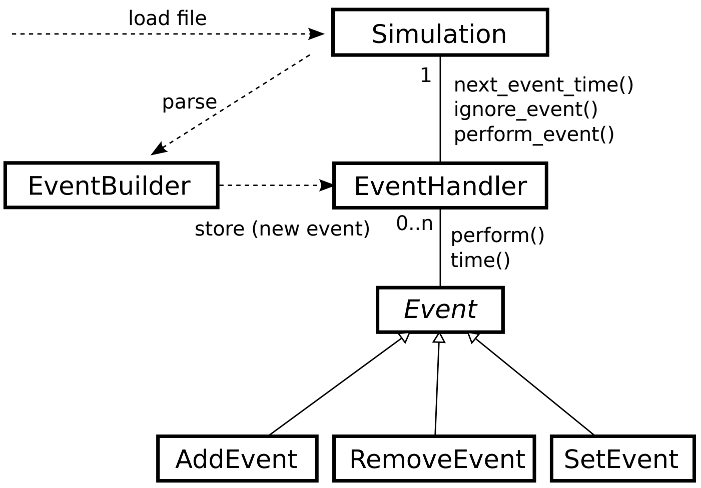


Figure 19: Events: another way to modify chemical concentrations aside from reactions, *e.g.* to simulate the injection of a chemical inside a cell.

3.7. Input/Output handling

3.7.1. Simulator Input

The simulator needs the following to work:

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

Caution:

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

3.7.2. Simulator Output

Outputs provided by the simulator are:

- A general output file logging 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`.

4. Detailed design

4.1. Reactants

4.1.1. `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.

4.1.2. `BoundChemical`

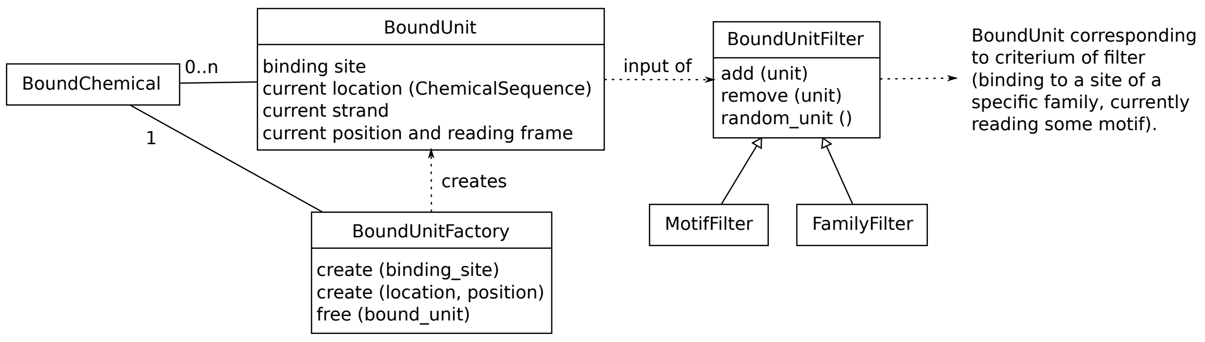

BoundChemical
0..n
1
BoundUnit
binding site
current location (ChemicalSequence)
current strand
current position and reading frame
input of
creates
BoundUnitFactory
create (binding_site)
create (location, position)
free (bound_unit)
BoundUnitFilter
add (unit)
remove (unit)
random_unit ()
MotifFilter
FamilyFilter
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` 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.

4.1.3. ChemicalSequence

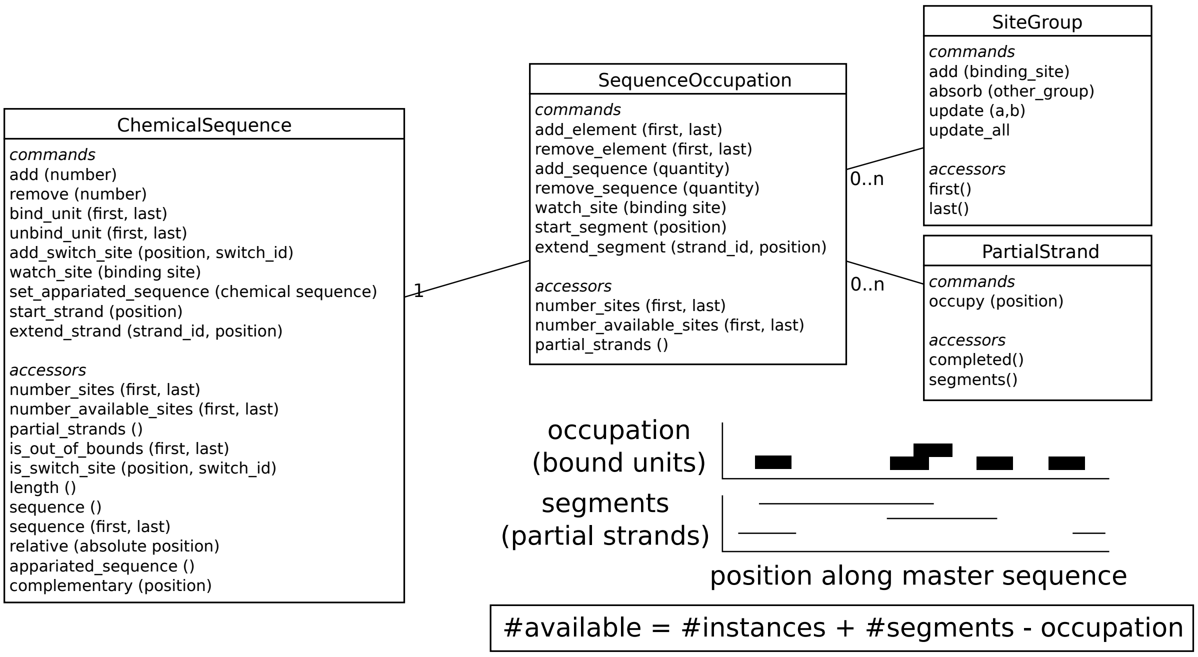

ChemicalSequence
commands
add (number)
remove (number)
bind_unit (first, last)
unbind_unit (first, last)
add_switch_site (position, switch_id)
watch_site (binding site)
set_appariated_sequence (chemical sequence)
start_strand (position)
extend_strand (strand_id, position)
accessors
number_sites (first, 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
0..n
SiteGroup
commands
add (binding_site)
absorb (other_group)
update (a,b)
update_all
accessors
first()
last()
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.

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.

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

4.1.4. `DoubleStrand`

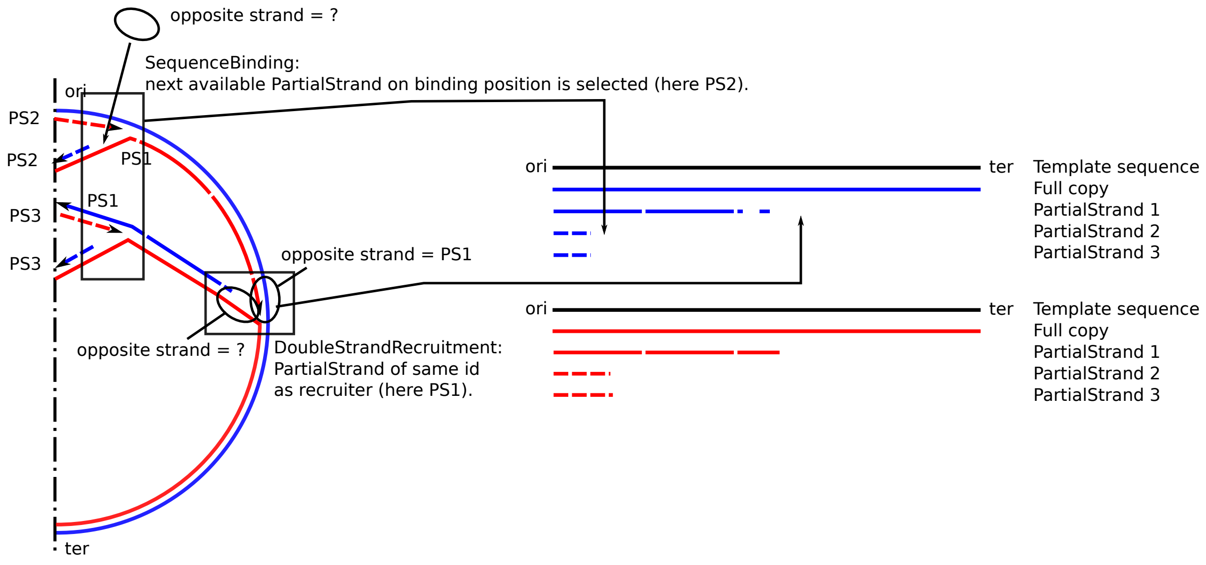


Figure 22: Strands of a `DoubleStrand` are identified according to creation order. Every time a new segment is polymerized, it is necessary to determine which `PartialStrand` is elongated. If a polymerase has been recruited on the complementary strand by `DoubleStrandRecruitment`, it is automatically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

Strand identification Because `DoubleStrand` typically 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.

4.1.5. `BindingSiteFamily`

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

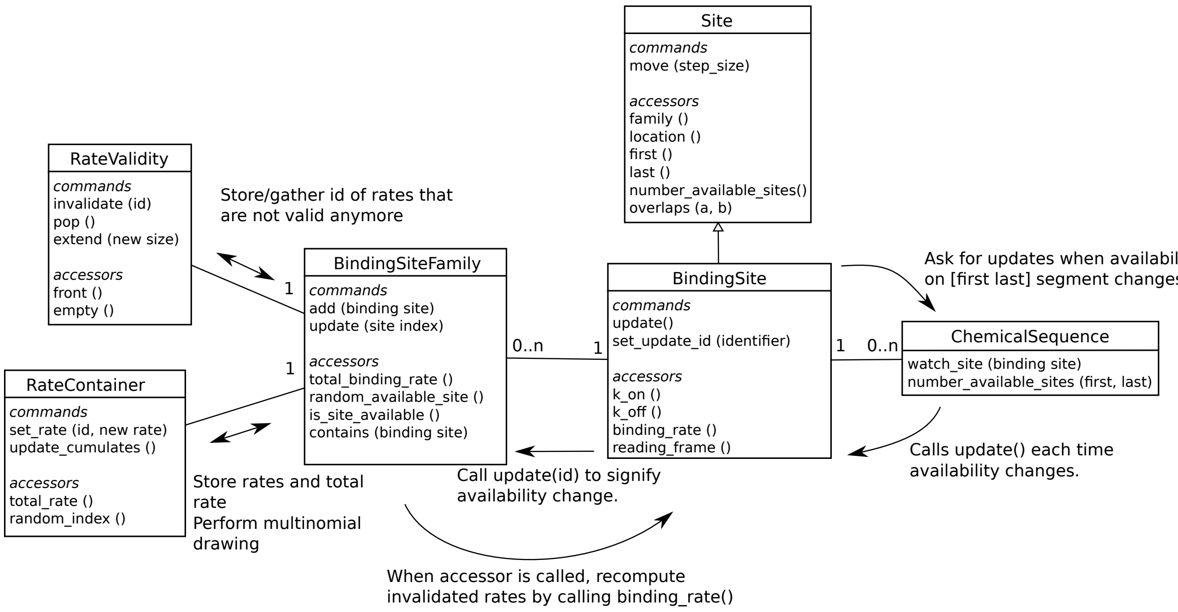


Figure 23: Schematical view of the Observer pattern used to keep availability of binding sites up to date for `SequenceBinding` reactions.

Every `BindingSite` is viewed as an *observer* by the `ChemicalSequence` it belongs to. Every time a change occurs on the site, the `BindingSite` is notified. The latter binding site notifies its `BindingSiteFamily` using a specific identifier, letting the family know which binding rate is out of date. This information is stored in a `RateValidity` class. It is only when it is really needed (*i.e.* when a `SequenceBinding` wants to access total rate or a random site) that rates are recomputed. This avoids useless computations *e.g.* in the case of a translocation, where a bound unit is first unbound from its `ChemicalSequence` then rebound. If the bound unit does not move away from the site, two updates will be sent, but the rate will only be recomputed once at the end.

4.2. Reactions

4.2.1. `ChemicalReaction`

Nothing particular.

4.2.2. `SequenceBinding`

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

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

4.2.3. `Translocation`

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

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

4.2.4. `Loading`

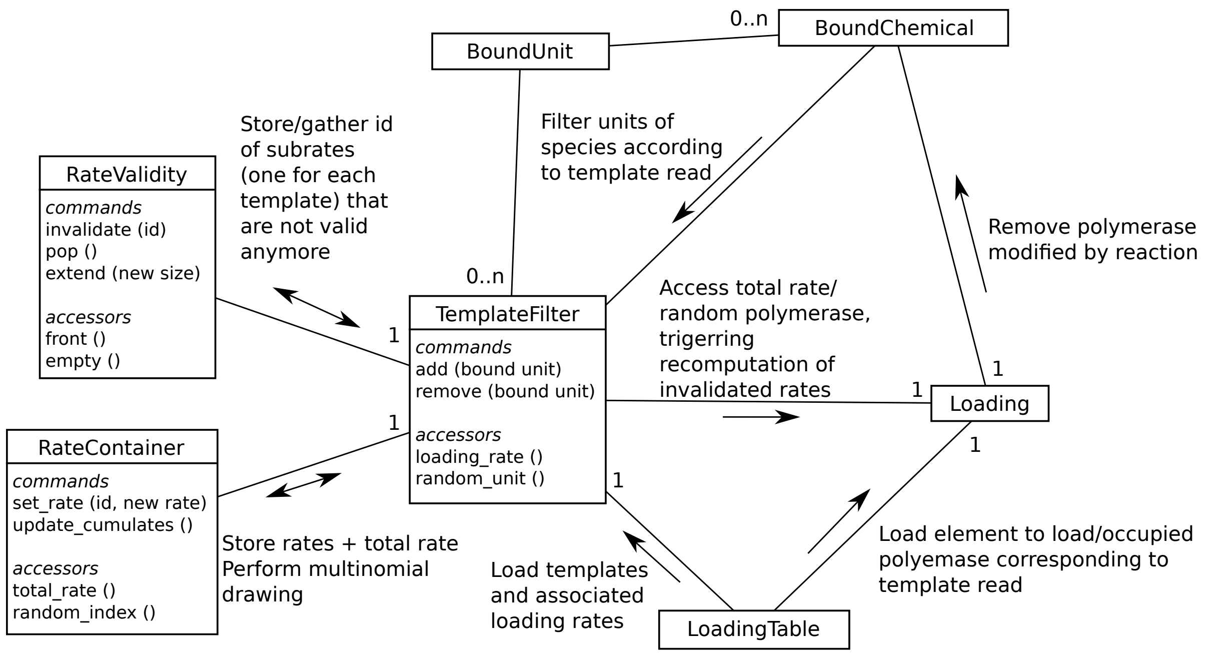

BoundUnit
0..n
BoundChemical
Store/gather id of subrates (one for each template) that are not valid anymore
Filter units of species according to template read
RateValidity
commands
invalidate (id)
pop ()
extend (new size)
accessors
front ()
empty ()
Remove polymerase modified by reaction
0..n
TemplateFilter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

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

Handling each polymerase individually The main challenge with `Loading` is to maintain the subrates associated with each motif up to date. It needs to maintain a list of all `BoundUnit`s reading a 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).

`ProductLoading` **vs** `DoubleStrandLoading` There difference between the two processes is rather small. We just added a failure condition in the case of `DoubleStrandLoading` for convenience. Depending on what reactions are used to synthesize a `DoubleStrand` it might be possible that a polymerase arrives upon a position that has already been synthesized. In this case, the `DoubleStrandLoading` fails and the polymerase is replaced by the polymerase in its stalled form.

4.2.5. `Release`

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

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

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 (Dinh et al., 2016). We only give a quick overview here.

4.3.1. `RateContainer` classes

We start with the lowest level classes, which perform one of the central tasks of the Gillespie algorithm: drawing a reaction from reaction rates. For efficiency reasons, we

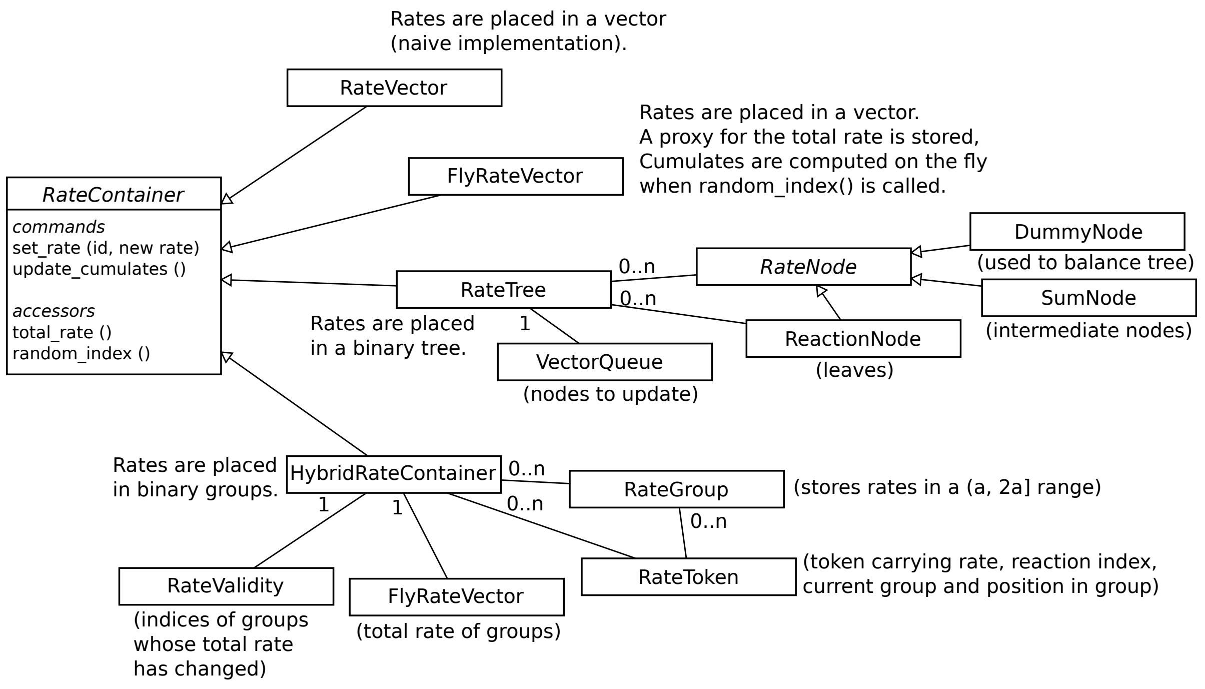


Figure 25: Implementations provided to store rates and perform a multinomial drawing. Implicitly, all theses classes use `RandomHandler` to perform their random drawings.

propopose several implementation of the algorithm (Fig. 25). Comparison and description of theses classes are given in Dinh et al. (2016).

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

4.3.2. `RateManager` classes

The second layer of the solver loop ensures that the rates are updated when needed to. Two implementations are proposed for this task (Fig. 26). The `NaiveRateManager` updates every rate. While it is inefficient, it can be used as a reference to test other managers. The `DependencyRateManager` uses an observer pattern to update only reactions for which a reactant concentration has changed (see Dinh et al. (2016) for further details).

4.3.3. `Solver` classes

For the moment, only one solver class is fully available to the user, `NaiveSolver`, which implements the exact Gillespie algorithm. Another variant called `ManualDispatchSolver` is implemented, 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

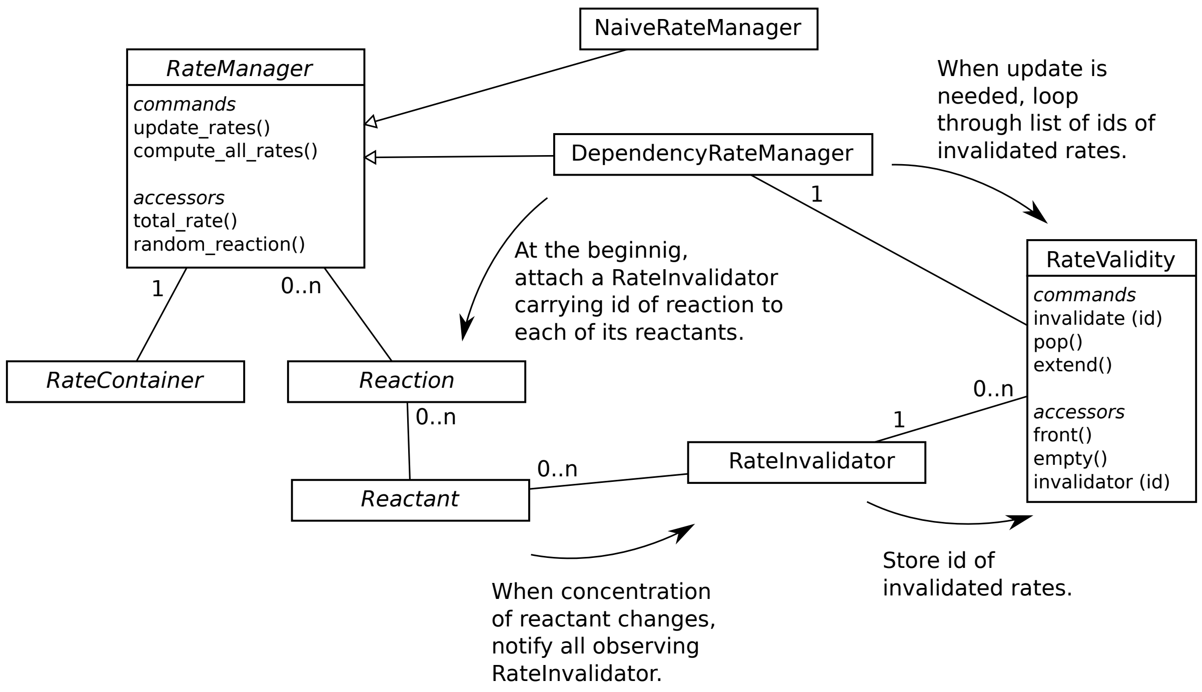


Figure 26: Implementations provided to update reaction rates. Note that the drawing part of the algorithm is always delegated to a `RateContainer`. `DependencyRateManager` uses an Observer pattern to monitor which rates have changed.

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.

4.4. Input/Output handling

4.4.1. Parsing system

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

4.4.2. Builder and interpreter

The parsing system is designed to cut each line into words, then the words are interpreted by `Builders` that try to create instances of each of the class of the simulator. The `Parser` loops through the `Builders` 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,

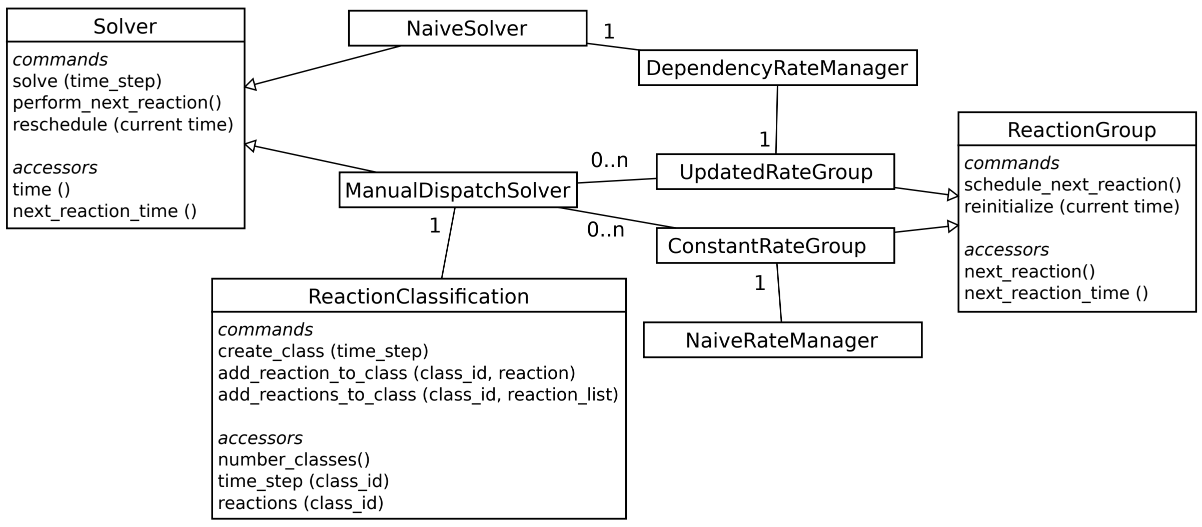


Figure 27: Two implementations of the `Solver` class organizing rate updating. `NaiveSolver` forces recomputation of rates at each time step. `ManualDispatchSolver` puts reactions into groups: reactions in `UpdatedRateGroup` are updated after every reaction while those in `ConstantRateGroup` only at 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.

a `DependencyException` is raised. In the latter case, the `Parser` will postpone the line until dependency can be successfully solved.

4.4.3. Output

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

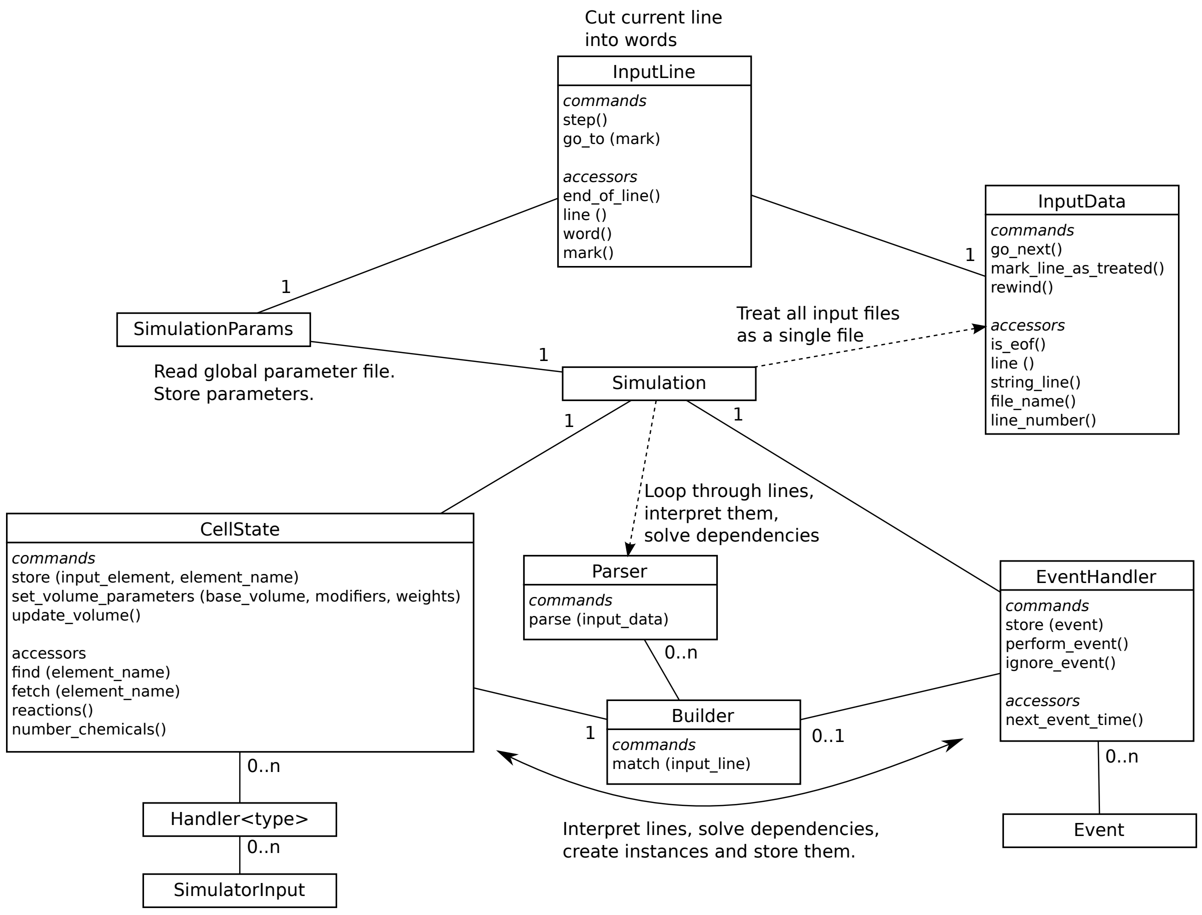
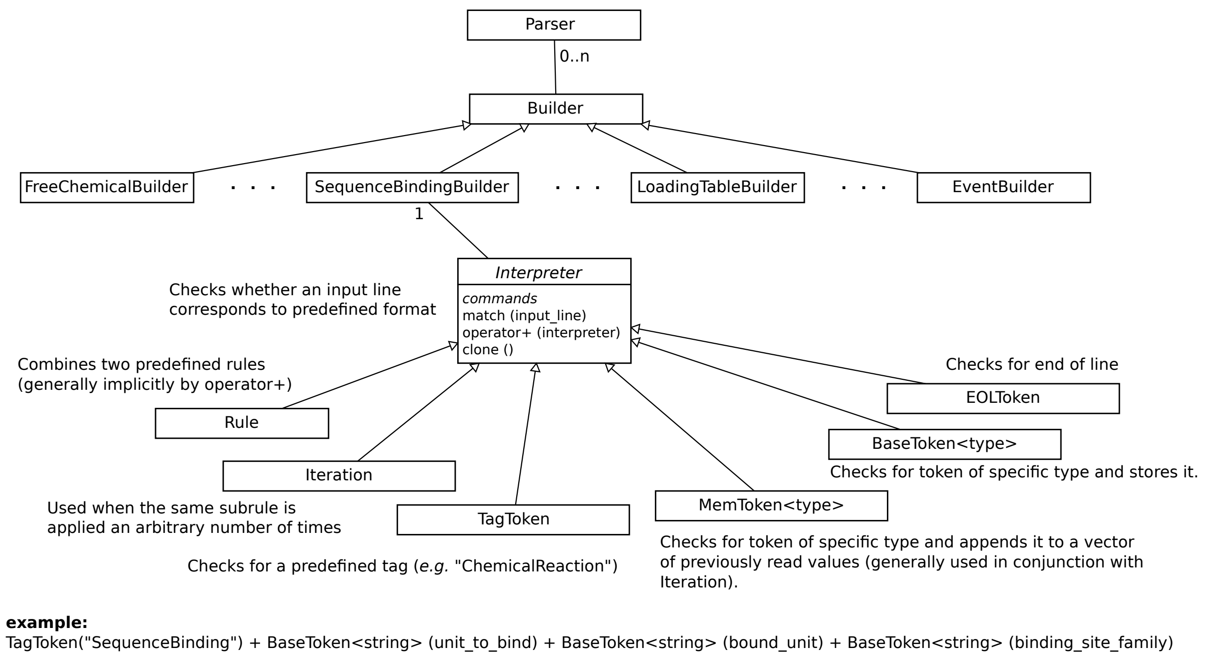


Figure 28: Architecture of the parsing system. `SimulationParams` reads the global parameter file and stores parameter values. `InputData` is used to provide a simplified interface to the input files and mark treated lines. A `Parser` and `Builders` are used to make sense of individual lines. Everything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



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.

A. Utility classes

A.1. Exceptions

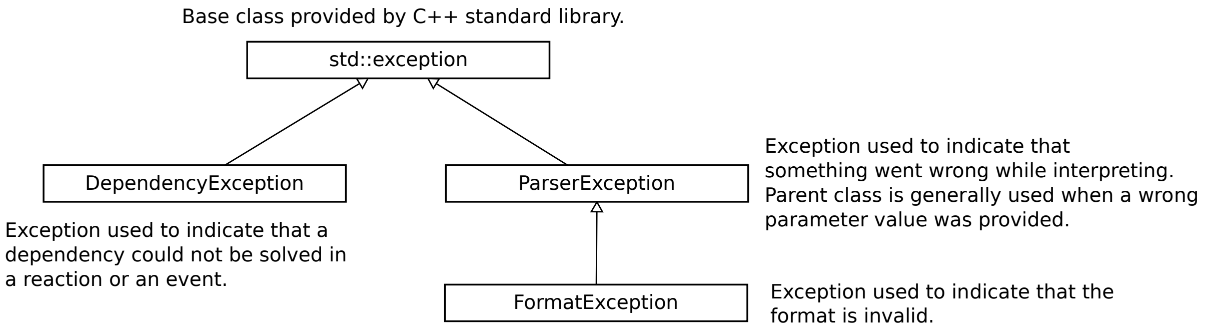

Base class provided by C++ standard library.
std::exception
DependencyException
ParserException
FormatException
Exception used to indicate that something went wrong while interpreting. Parent class is generally used when a wrong parameter value was provided.
Exception used to indicate that a dependency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

Figure 30: Exceptions used in the simulator.

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

A.2. Random handler

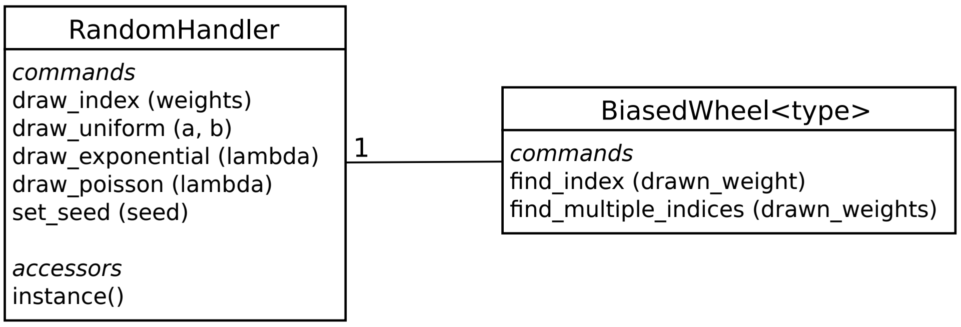


Figure 31: `RandomHandler` and `BiasedWheel` used for multinomial drawing. `RandomHandler` uses the Singleton pattern, meaning exactly one instance of the class is created and used throughout the simulator. It has to be accessed using `RandomHandler::instance()`.

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

A.3. Factories

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

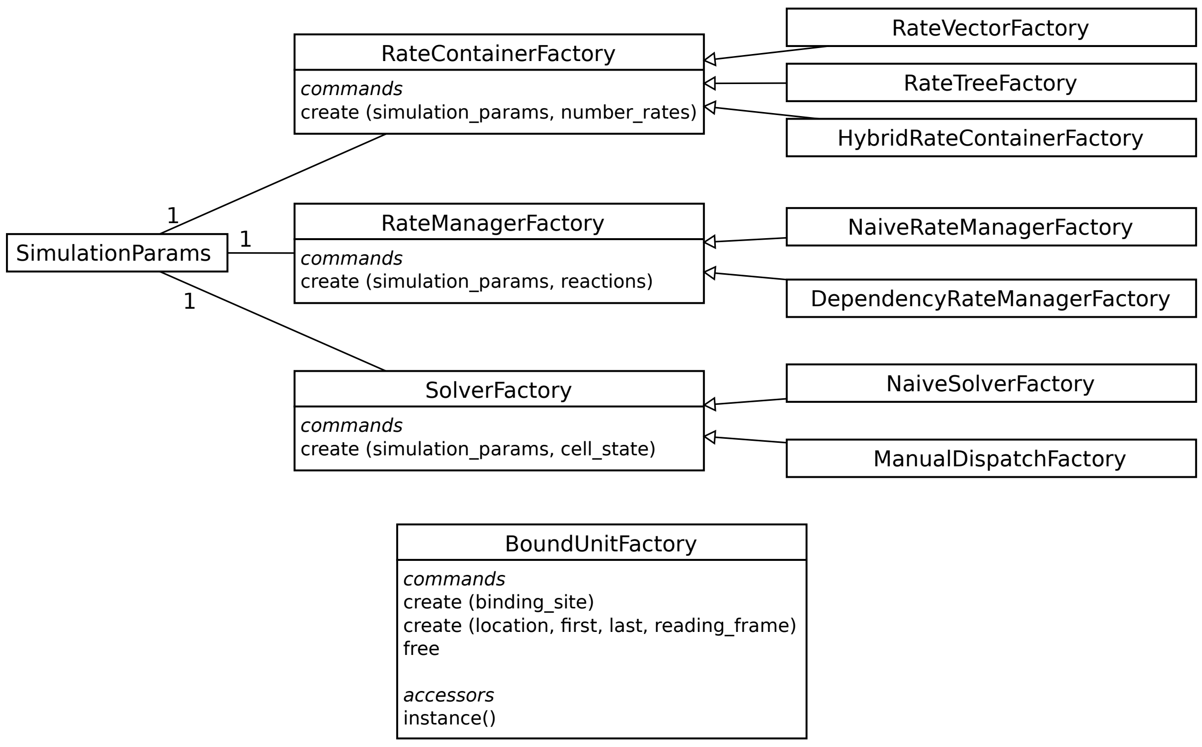


Figure 32: `RateContainerFactory`, `RateManagerFactory` and `SolverFactory` are used by `SimulationParams` to record what `RateContainer`, `RateManager` and `Solver` the user wishes to use (Abstract Factory Pattern). `BoundUnitFactory` is used to control memory usage by `BoundUnits`.It enables recycling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

A.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 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.

A.5. Handler classes for memory handling

C++ has no garbage collector and this program was designed according to old standards (that is without smart pointers). Therefore, we need to be extremely careful to delete elements at the right time. This is achieved by using as few storage places as possible. As described earlier, `CellState` is used to store all reactions and reactants read from files. We designed a `Handler` class that effectively stores objects to their definitive location and distributes references or pointers to this constant location. Similarly `EventHandler` and `BoundUnitFactory` are used to store `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.

B. Tests

B.1. Testing philosophy

We use three layers of tests: *programming by contract*, *unit tests*, *integration tests* (Tab. 1). They are integrated in an automated framework to detect bugs rapidly and at the lowest possible level.

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

Programming by contract These tests typically apply to attributes of classes and arguments of methods. They are usually divided into three subcategories: *preconditions*, *postconditions* and *invariants*. They check whether the class interacts correctly with the outside world, generally other classes. We left invariants out because they are hard to check in a language that does not support them natively. In the simulator, we defined two macros `REQUIRE` and `ENSURE` to test pre- and postconditions. Each time a precondition or a postcondition is broken, the program is interrupted and the condition that was broken is displayed (using `assert()`). Preconditions and postconditions are extremely useful for detecting simple typing mistakes, numerical issues and so on (Fig. 33).

```
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.

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

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

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

B.2. Organizing and running tests

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

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

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

Code needs to be recompiled after it has been configured.

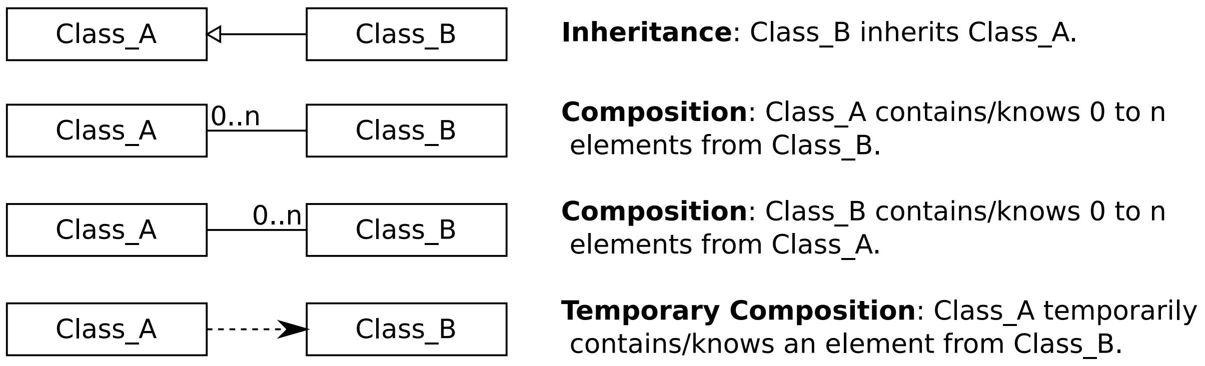
- Preconditions and postconditions are automatically checked every time the program is run (for unit tests, integration tests or any kind of other run). Remember to turn them off for real simulations as they are very time consuming.
- Unit tests and/or integration tests are run by running `make check`.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

C. Formats and Conventions

C.1. Input format description

- A plain word indicates a tag, that needs to be written.
- `<...>` indicates a variable that has to be completed with an existent element of the specified type.
- `[...]` indicates an optional part.
- `[...]^{0..n}` indicates an optional part that can be repeated an arbitrary number of times.
- `[...]^{1..n}` indicates an part that can be repeated an arbitrary number of times, at least once.
- `[...,]^{0/1..n}` indicates a part that can be repeated an arbitrary number of times, each repetition being separated by a `,` (*but there is actually no* `,` *after the last repetition*).

C.2. UML



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.

References

Marc Dinh, Stephan Fischer, and Anne Goelzer. CATI MIAGO: comparing algorithms for gillespie based simulations. 2016.