# Contents

**1 Introduction** **2**

**2 Global presentation of the components of the simulator** **2**
2.1 Components of the simulator . . . 2
2.2 Reactant hierarchy . . . 3
2.2.1 UML class diagram . . . 3
2.2.2 Reactant . . . 4
2.2.3 Chemical . . . 4
2.2.4 FreeChemical . . . 4
2.2.5 BoundChemical . . . 4
2.2.6 ChemicalSequence . . . 4
2.2.7 DoubleStrand . . . 5
2.2.8 BindingSiteFamily . . . 6
2.3 Reaction hierarchy . . . 7
2.3.1 UML class diagram . . . 7
2.3.2 Reaction . . . 7
2.3.3 ChemicalReaction . . . 7
2.3.4 SequenceBinding . . . 8
2.3.5 Translocation . . . 9
2.3.6 Loading . . . 10
2.3.7 Release . . . 11
2.3.8 Degradation . . . 12
2.4 Event hierarchy . . . 13
2.5 Rate container hierarchy . . . 13
2.6 Solver hierarchy . . . 13

**3 Detailed design** **14**
3.1 Rate containers . . . 14
3.1.1 Rate container interface, communication with other classes . . . 14
3.1.2 Rate container implementations . . . 14

# 1 Introduction

The aim of this document is not to go into technical details of the implementation (the code is documented using Doxygen, technical details are therefore best found in the Doxygen-generated manual). Rather we wish to walk through the choices in design that have been made. The technical manual hopefully contains necessary information for understanding the concept behind each class and how to use it. However it does not tell you what classes are central in the architecture and it is difficult to see at a glimpse how classes interact.

Describing global design should be more agreeable to read than the purely technical document. It helps understand how we tackled a certain number of efficiency issues (should it be speed or maintainability). Most efficiency issues in programming are related to architecture design rather than class implementation. Efficiency of an architecture can be related to information flowing between classes. Restricting information access through classes and dependencies between classes is generally considered good style, as it limits data corruption and enhances maintainability. Achieving this increases the probabilities that the simulator behaves the way it should and facilitates further expansions.

Once data protection and maintainability are ensured, speed issues are addressed only if they can be identified. Most parts of the simulator are not critical in that regard and do not need a particularly refined design or implementation. If speed issues arise, two level of solutions can be worked on. At the lowest level, class implementations can be changed to perform some routines more quickly (generally leading to at most a couple-fold speed increase). At the highest level, communication between classes can be tuned to ensure that only the necessary computations are done (generally leading to a drastic speed increase and a complexification of the architecture with new classes that "filter" communications).

To sum up, description at a global level gives critical insight into how the simulator works and where speed/design issues have been identified during development. It should facilitate discussions even with non-programmers (or at least non-C++-programmers).

# 2 Global presentation of the components of the simulator

## 2.1 Components of the simulator

The simulator can be decomposed into several large modules that handle specific tasks during simulation (Fig. 1). First of all, there is an **input/output** module that creates everything that is needed for the simulation from an input file (Fig. 1 - Initialization). **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 (Fig. 1 - Loop). 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.

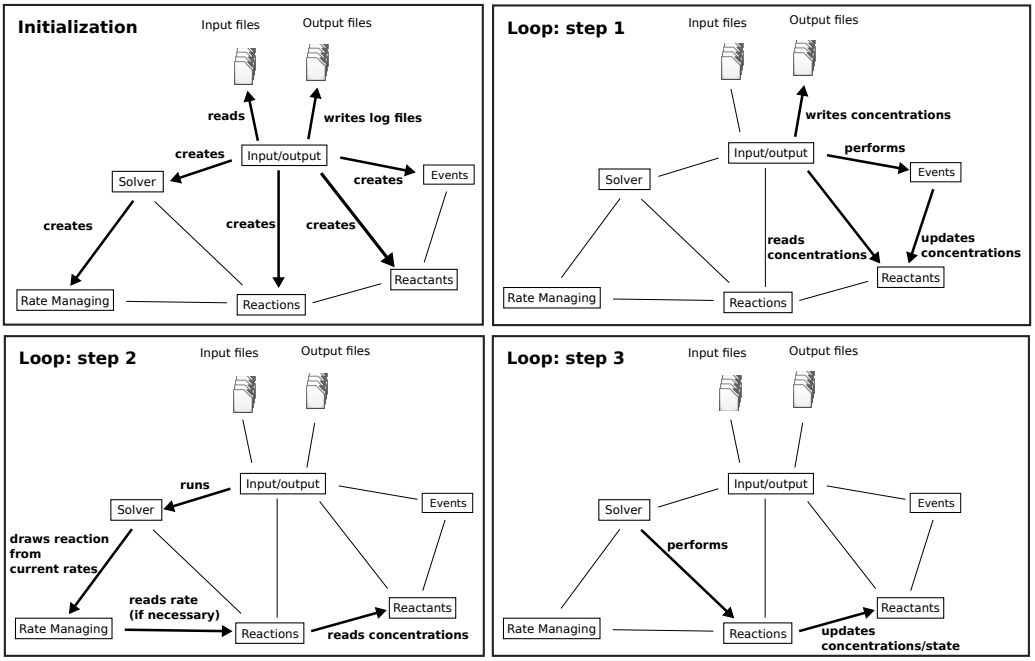
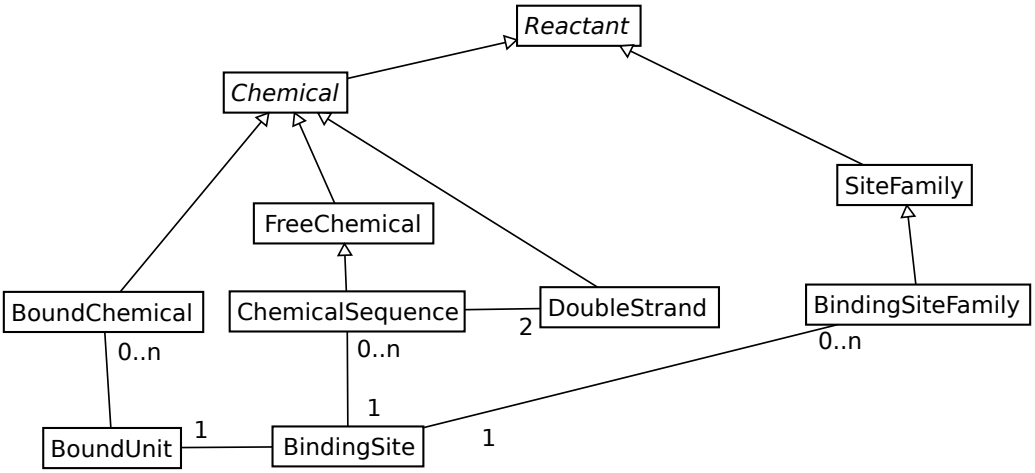



Figure 1: Schematical view of the simulator.

## 2.2 Reactant hierarchy

This section gives a quick overview of the contents of the **reactant** module. More details about how reactants are implemented can be found later.

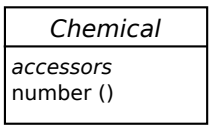
### 2.2.1 UML class diagram



### 2.2.2 Reactant

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

### 2.2.3 Chemical

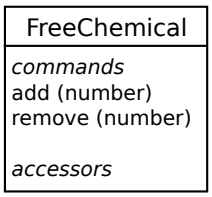



`Chemical` is an abstract class. It defines all standard chemical entities. `Chemical` represents a *pool* of a given chemical species, meaning that one may access its current number at any time.

### 2.2.4 FreeChemical

#### Input format

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




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

### 2.2.5 BoundChemical

#### Input format

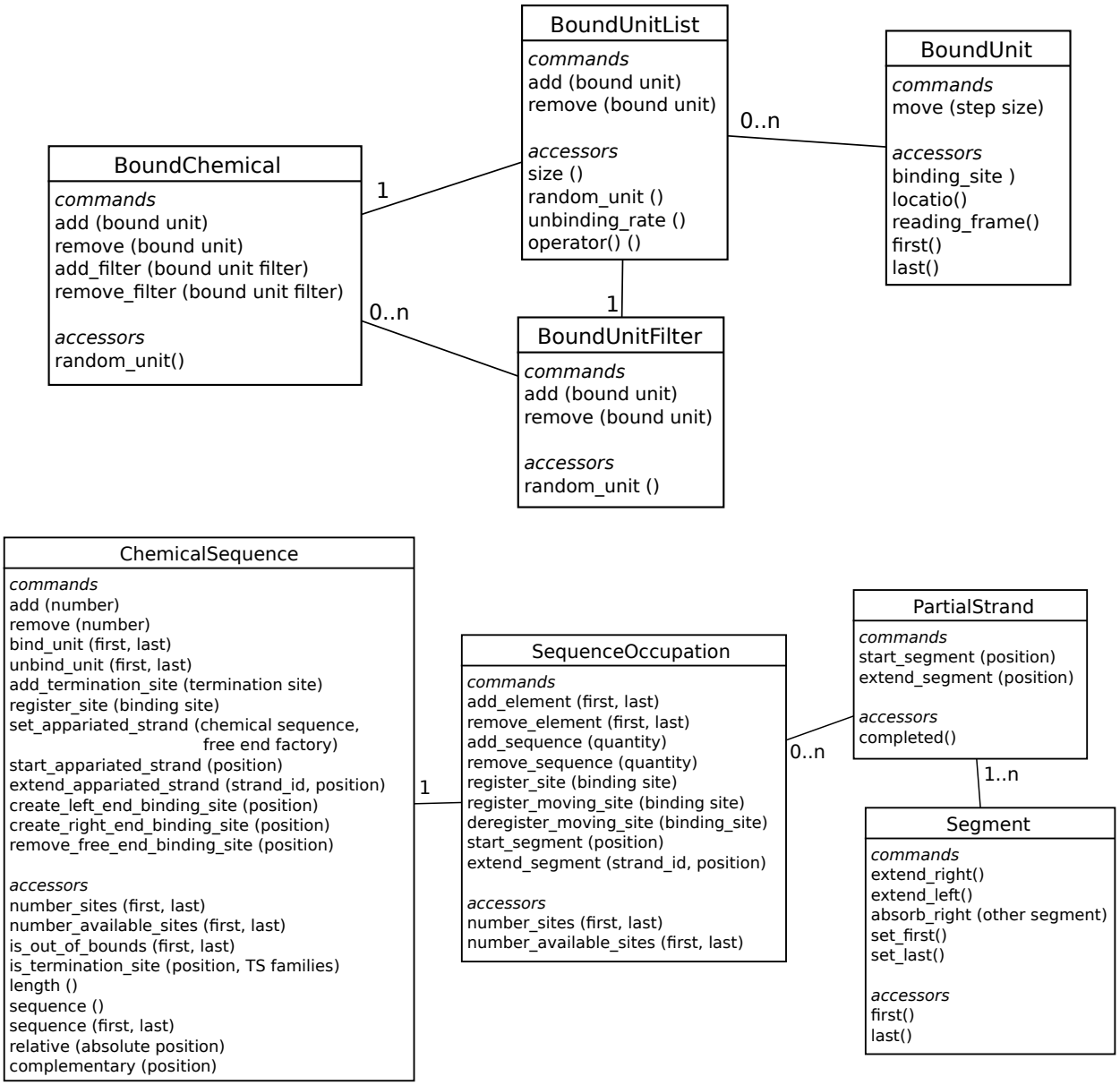
```
BoundChemical <name>
```

`BoundChemical` 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 two classes `BoundUnit` and `BoundUnitList` to represent molecules individually. It uses `BoundUnitFilter` to organize bound units according to outside criteria needed for reactions (classify according to binding sites, templates read, etc.).

### 2.2.6 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>]
```

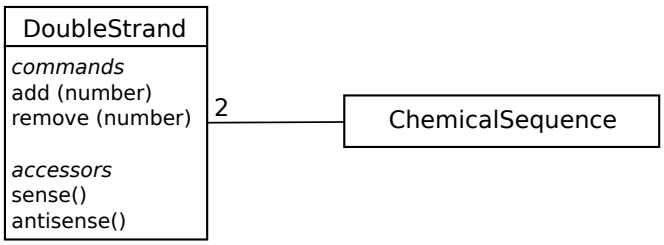



`ChemicalSequence` is a subclass of `FreeChemical`. It is defined by a sequence and the ability to bind elements. However, instances of a sequence are *not* treated individually, it is impossible to tell to which instance a given chemical bound. An object called `SequenceOccupation` maintains occupation 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`. New instances can be created segment wise, which is handled by classes `PartialStrand` and `Segment`. 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.

### 2.2.7 DoubleStrand

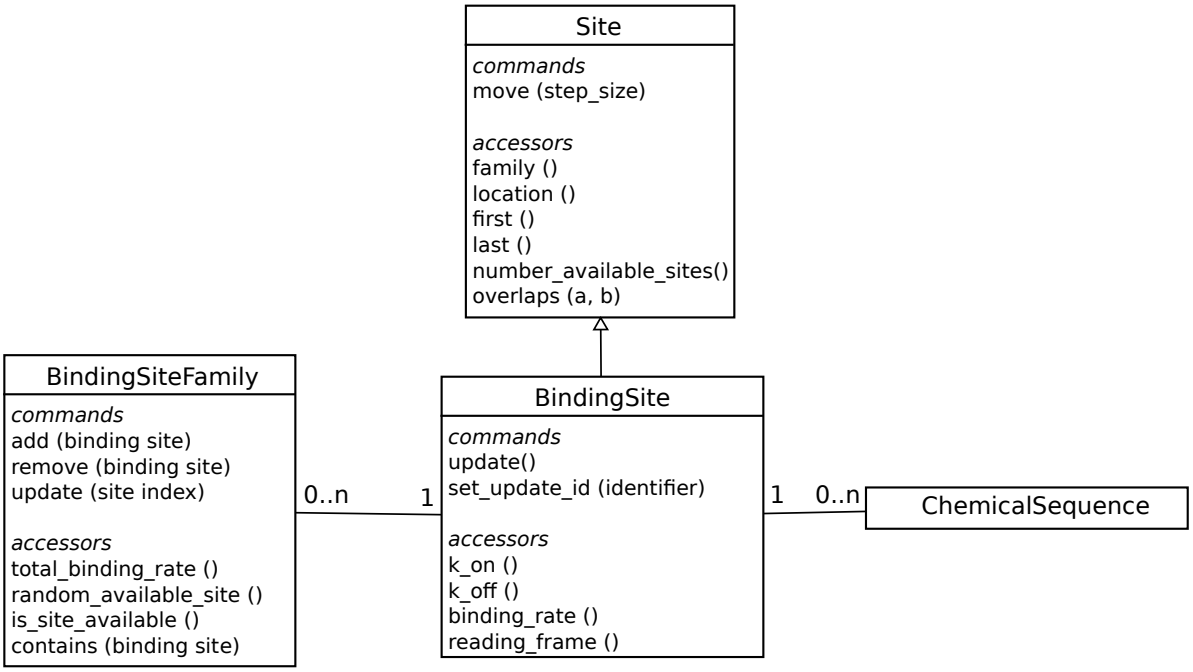
**Input format**

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




`DoubleStrand` 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, $A \rightarrow T$, $T \rightarrow A$, $C \rightarrow G$, $G \rightarrow C$).

### 2.2.8 BindingSiteFamily




`BindingSiteFamily` 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.

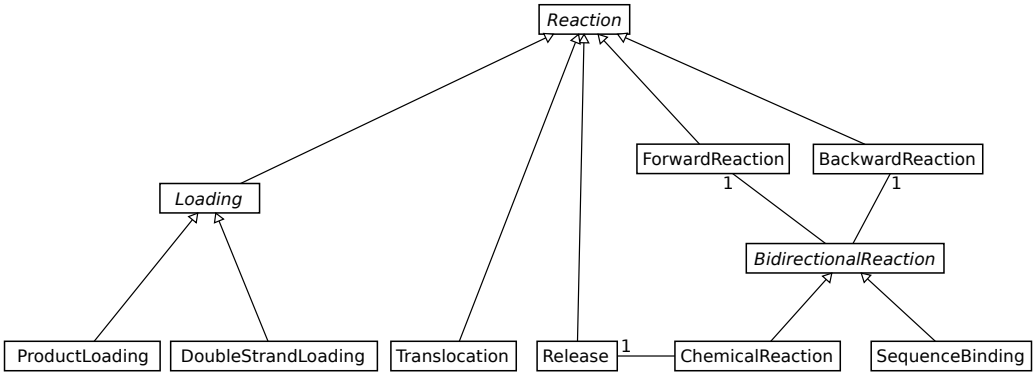
### Input format

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

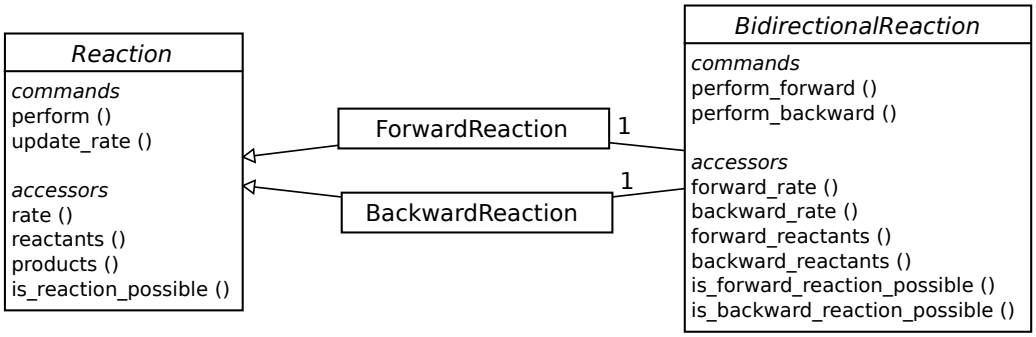
## 2.3 Reaction hierarchy

This section gives a quick overview of the reaction module. More details about how reactions are implemented can be found later.

### 2.3.1 UML class diagram




### 2.3.2 Reaction

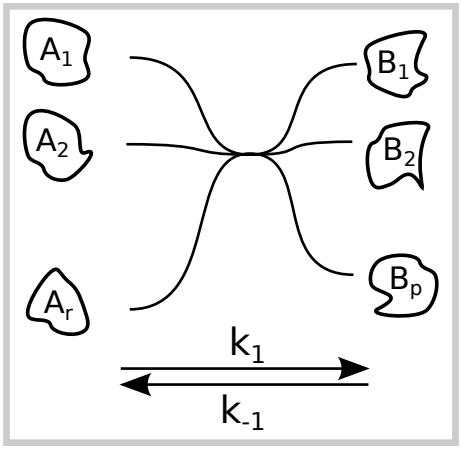



There are two abstract classes used to define reactions: `Reaction` for one-way reactions and `BidirectionalReaction` for reversible reactions. Two adapter classes `ForwardReaction` and `BackwardReaction` split reversible reactions in two one-way reactions. In the end, the solver only handles one-way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

### 2.3.3 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. It is defined by

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

where

- $A_i$ and $B_i$ are of type `FreeChemical`. They can be of type `BoundChemical`, *if* there is exactly one `BoundChemical` on each side of the equation and the associated stoichiometric coefficient is 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, 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).

**Rate** The rates are given by

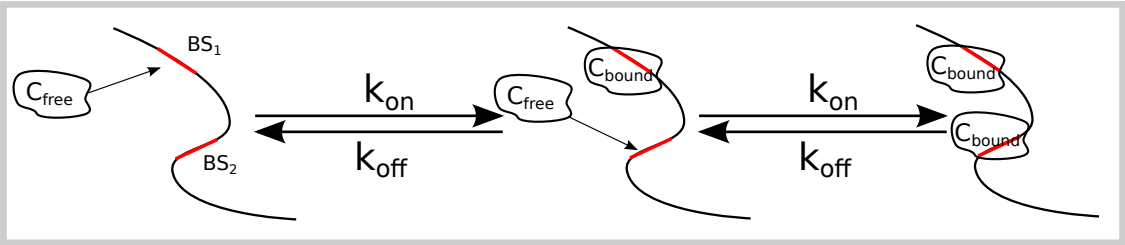
$$\lambda_{forward} = k_1 \prod_{i=1}^{r} [A_i]^{a_i}$$

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

### 2.3.4 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. 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.

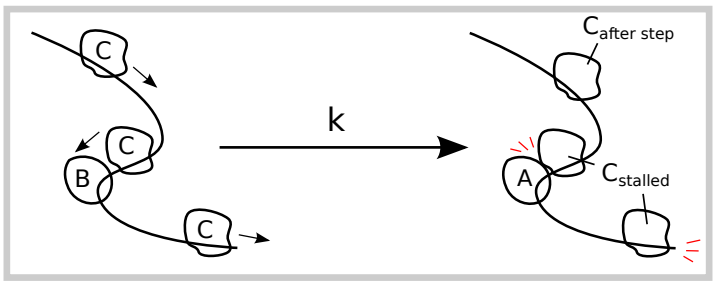
**Rate** The rates are given by

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

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

- $(k_{on})_s$ 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.

### 2.3.5 Translocation



### Input format

```
TerminationSite <family name> <chemical sequence> <start> <end>
Translocation <bound chemical> <form after step> <stalled form> \
  <step size> <rate> [<termination site family>]^{0..n}
```

**Formula** A `Translocation` represents movement of a bound element along a sequence. It is defined by

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

or

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

where

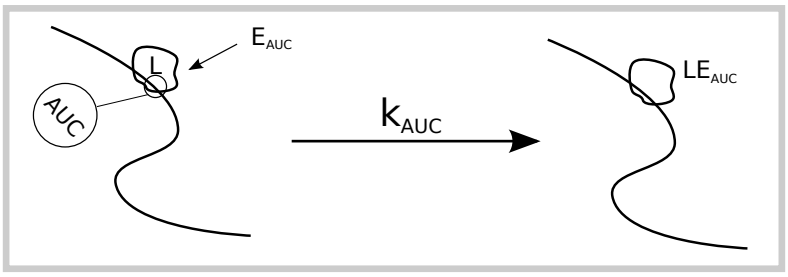
- $C$ is of type `BoundChemical`.
- $C_{\text{after step}}$ is of type `BoundChemical`.
- $C_{\text{stalled form}}$ is of type `BoundChemical`.
- $k$ is a rate constant.

**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 or it reaches a termination site, it is replaced by $C_{\text{stalled form}}$.

**Rate** The rate is given by

$$\lambda = k[C]$$

### 2.3.6 Loading




### Input format

```
LoadingTable <name> [<template> <element_to_load> <occupied_polymerase>,]^{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. 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.

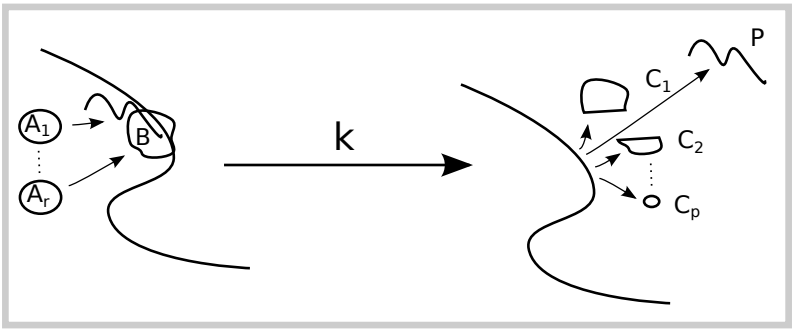
**Rate** The rate is given by

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

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 $t$.

### 2.3.7 Release



**Input format**

```
TransformationTable <name> [<parent_letter> <product_letter>,]^{1..n}
ProductTable <name> <transformation table>
Release <bound chemical> [<chemical> <stoichiometry>]^{0..n} rate <rate> \
  [produces <product table>]
```

**Formula** A `Release` represents detachment of a bound element from a sequence. An arbitrary number of elements can be used as coreactants to trigger the release, and an arbitrary number of products are released. It is defined by

$$B + a_1 A_1 + ... + a_r A_r \xrightarrow{k} \delta_P P + c_1 C_1 + ... + c_p C_p$$

where

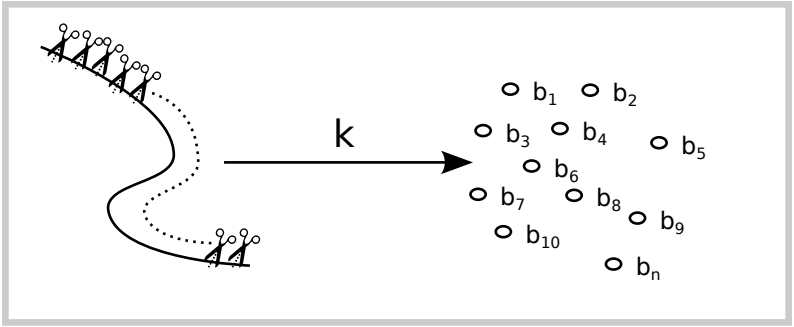
- $B$ is of type `BoundChemical`.
- $P$ is of type `ChemicalSequence`. It is a product that is released if $B$ is a polymerase. In the latter case $\delta_P = 1$ else $\delta_P = 0$.
- $A_i$ are of type `FreeChemical`.
- $C_i$ are of type `FreeChemical`.
- $a_i$ and $c_i$ are stoichiometric coefficient.
- $k$ is a rate constant.

**Action** When the reaction is performed, a random $B$ is chosen and removed from the pool (and detached from its sequence). The $A_i$ and $C_i$ pool are changed according to their stoichiometric coefficients. If a `ProductTable` is defined, it means that the released chemical is a polymerase. A product (a chemical sequence) corresponding to the bound chemical's binding and release points is added. The `ProductTable` is populated by the second input format of `ChemicalSequence`.

**Rate** The rate is given by

$$\lambda = k[B] \prod_{i=1}^{r} [A_i]^{a_i}$$

### 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. 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

$$\lambda = k[CS]$$

## 2.4 Event hierarchy

## 2.5 Rate container hierarchy

## 2.6 Solver hierarchy

# 3 Detailed design

## 3.1 Rate containers

### 3.1.1 Rate container interface, communication with other classes

### 3.1.2 Rate container implementations

See CATI file.