

IMSV: bacteria simulator design

Version 0.0

M. Dinh and S. Fischer

June 23, 2016

Contents

1	Introduction	4
2	Global presentation of the components of the simulator	4
2.1	Components of the simulator	4
2.2	Reactant hierarchy	5
2.2.1	UML class diagram	6
2.2.2	Reactant	6
2.2.3	Chemical	6
2.2.4	FreeChemical	6
2.2.5	BoundChemical	7
2.2.6	ChemicalSequence	7
2.2.7	DoubleStrand	8
2.2.8	BindingSiteFamily	9
2.3	Reaction hierarchy	9
2.3.1	UML class diagram	10
2.3.2	Reaction	10
2.3.3	ChemicalReaction	10
2.3.4	SequenceBinding	11
2.3.5	Translocation	12
2.3.6	Loading	13
2.3.7	Release	15
2.3.8	Degradation	16
2.4	Solver loop	17
2.4.1	Solver class	17
2.5	Input/Output handling	17
3	Detailed design	17
3.1	Reactants	17
3.1.1	FreeChemical	17
3.1.2	BoundChemical	17
3.1.3	ChemicalSequence	18
3.1.4	BindingSiteFamily	18
3.2	Reactions	20
3.2.1	ChemicalReaction	20
3.2.2	SequenceBinding	20
3.2.3	Translocation	20
3.2.4	Loading	20
3.2.5	Release	21
3.3	Solver loop	22
3.3.1	Rate container classes	22
A	Tests	23

A.1	Testing philosophy	23
A.1.1	Programming by contract	23
A.1.2	Unit tests	23
A.1.3	Integration tests	23
A.2	Organizing and running tests	23

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.

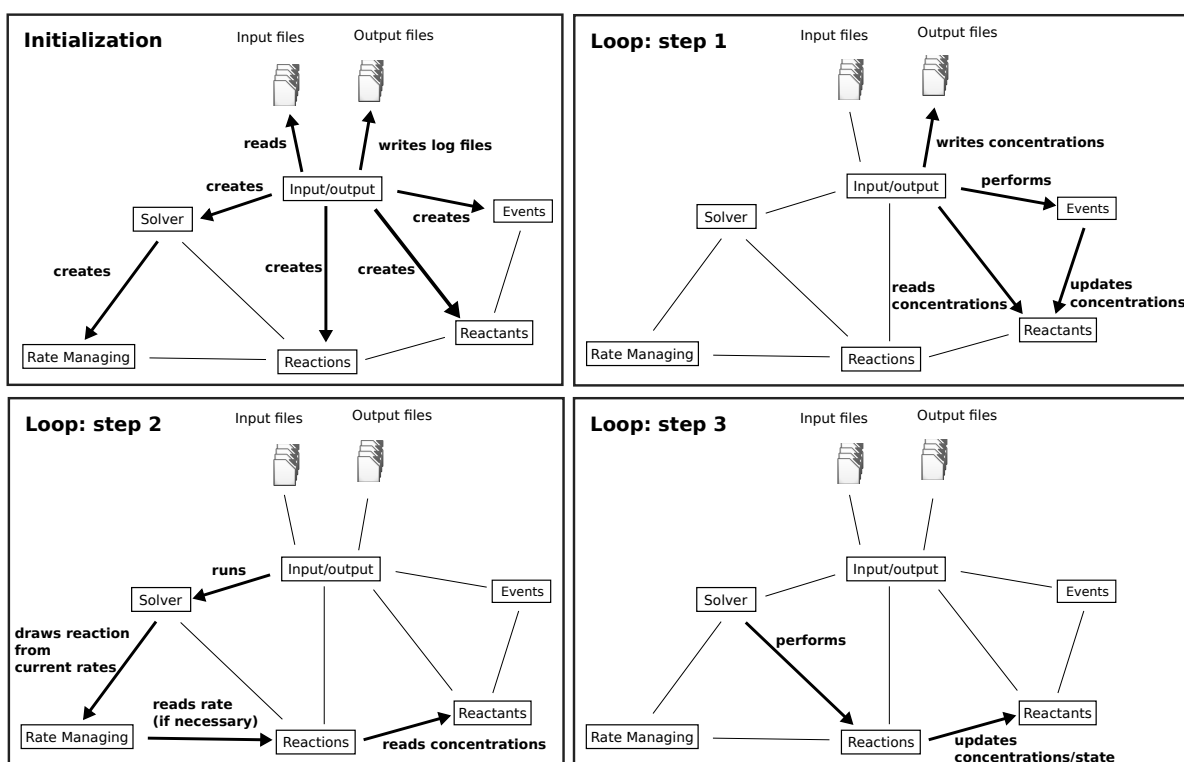
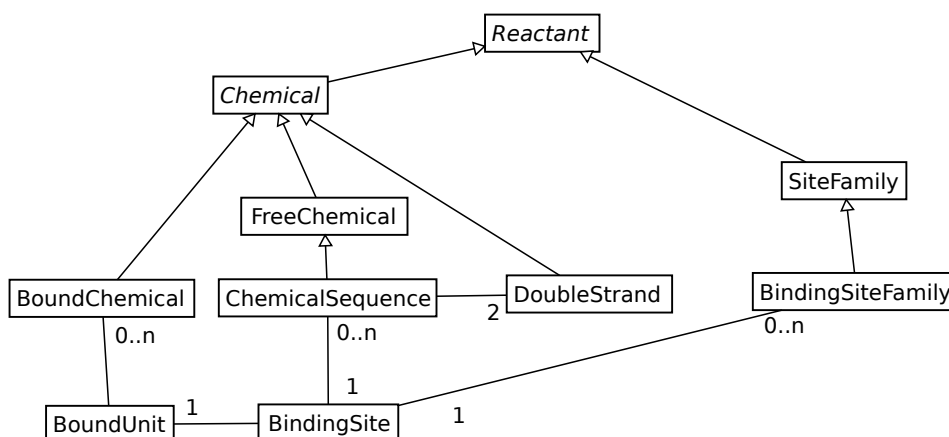


Figure 1: Schematical view of the simulator.

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.

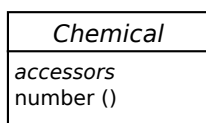
2.2.1 UML class diagram



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

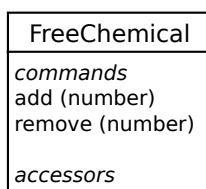


Chemical is an abstract class. It defines all standard chemical entities. **Chemical** represents a *pool* of a given chemical species, meaning that one may access its current number at any time.

2.2.4 FreeChemical

Input format

FreeChemical <name> [<initial quantity>]

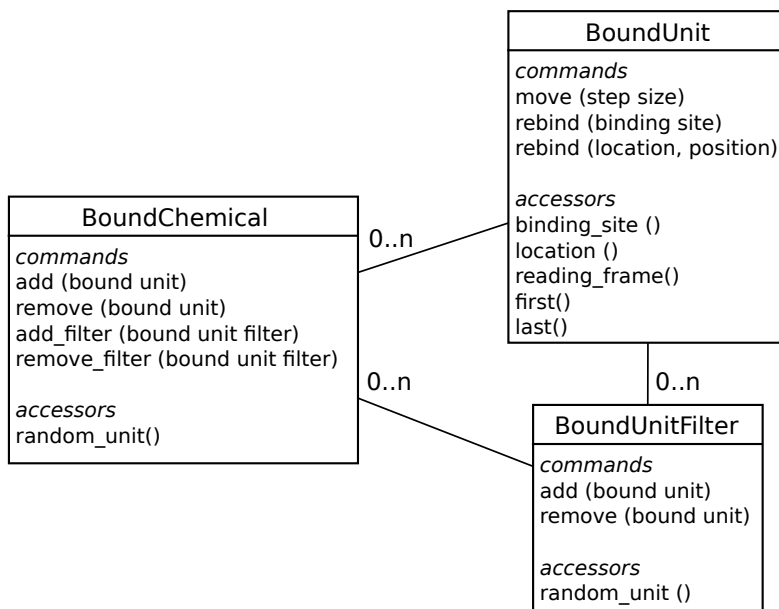


FreeChemical is a subclass of **Chemical** that represents free chemical (*e.g.* molecules diffusing in the cytosol or extracellular medium).

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 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.).

2.2.6 ChemicalSequence

Input format

ChemicalSequence <name> sequence <sequence> [<initial quantity>]

TransformationTable <name> [<parent_letter> <product_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

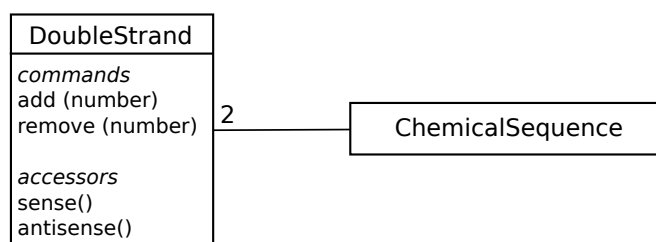
ChemicalSequence
<i>commands</i> add (number) remove (number) bind_unit (first, last) unbind_unit (first, last) add_termination_site (termination site) watch_site (binding site) set_appariated_sequence (chemical sequence) start_strand (position) extend_strand (strand_id, position)
<i>accessors</i> number_sites (first, last) number_available_sites (first, last) partial_strands () is_out_of_bounds (first, last) is_termination_site (position, TS families) length () sequence () sequence (first, last) relative (absolute position) appariated_sequence () complementary (position)

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.

2.2.7 DoubleStrand

Input format

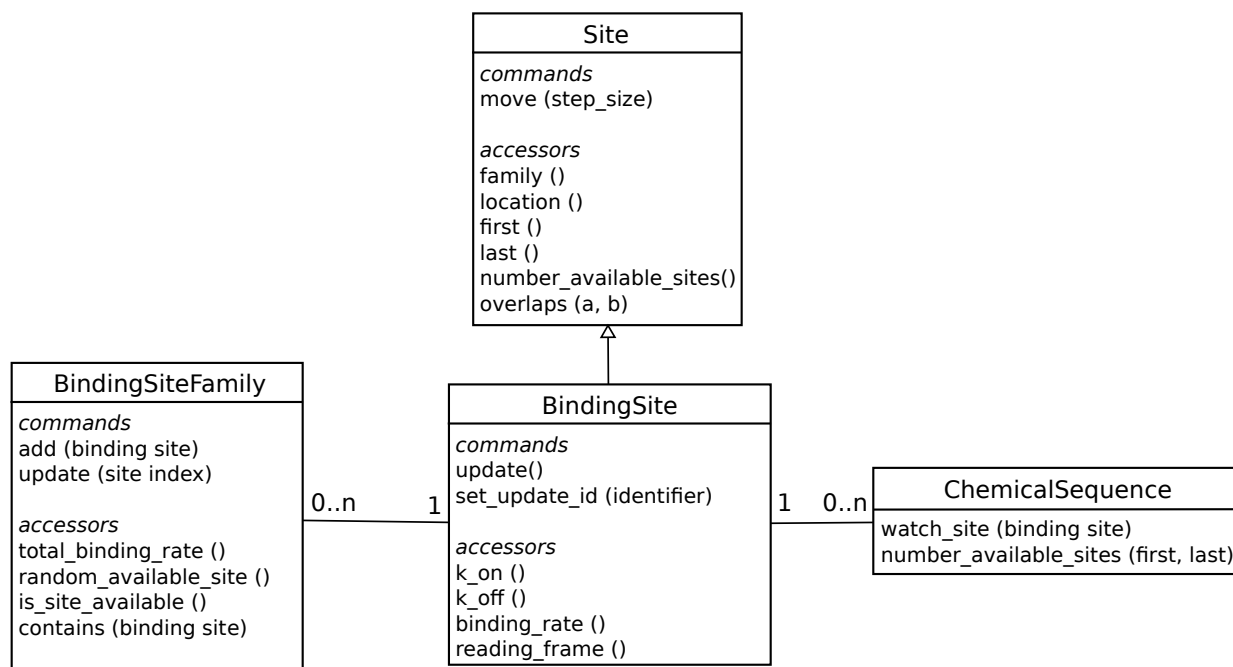
```
TransformationTable <name> [<letter> <complementary_letter>,<initial quantity>]^{1..n}
DoubleStrandSequence <name> <name_sense_sequence> <sense_sequence> \
    <name_antisense_sequence> <transformation_table> [<initial quantity>]
```



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

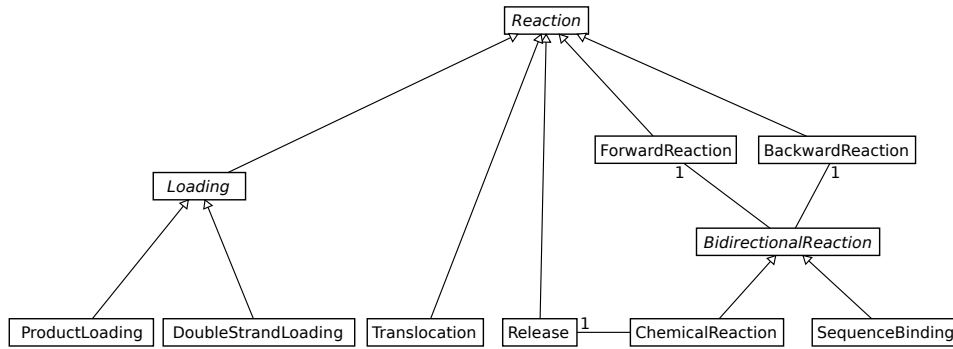
```

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

```

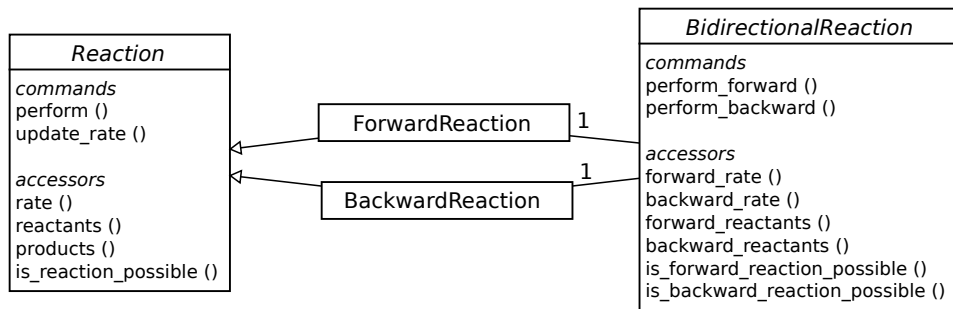
2.3 Reaction hierarchy

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



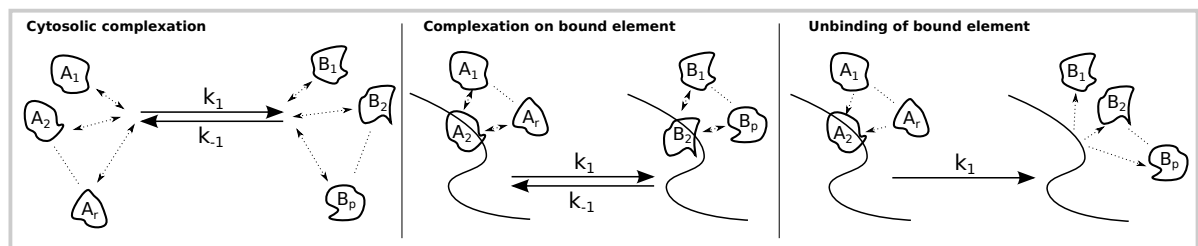
2.3.1 UML class diagram

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.

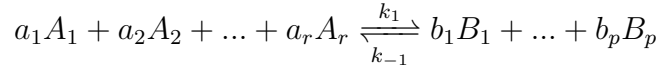
2.3.3 ChemicalReaction



Input format

ChemicalReaction [<chemical> <stoichiometry>]~{1..n} rates <k₁> <k₋₁>

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



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.

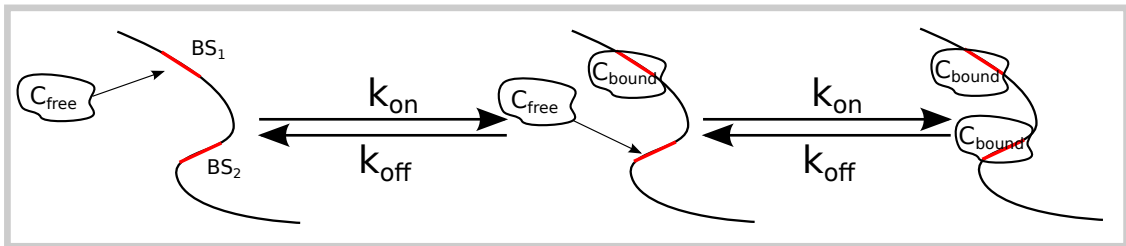
Action When the reaction is performed, the number of chemicals involved is changed according to their stoichiometric coefficient. If `BoundChemical` are involved on each side, the simulator will assume that the bound chemical that is consumed is replaced by the bound chemical on the other side of the 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

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

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

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



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.

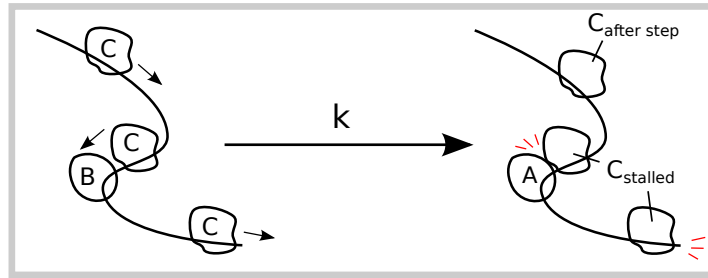
Rate The rates are given by

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

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

- $(k_{on})_s$ 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.

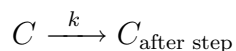
2.3.5 Translocation



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



or



where

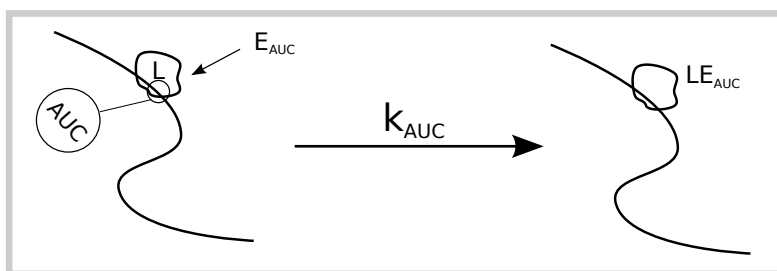
- C is of type BoundChemical.
- $C_{\text{after step}}$ is of type BoundChemical.
- $C_{\text{stalled form}}$ is of type BoundChemical.
- k is a rate constant.

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

$$\lambda = k[C]$$

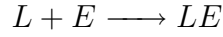
2.3.6 Loading



Input format

```
LoadingTable <name> \  
  [<template> <element_to_load> <occupied_polymerase> <rate>,<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. It is defined by



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.

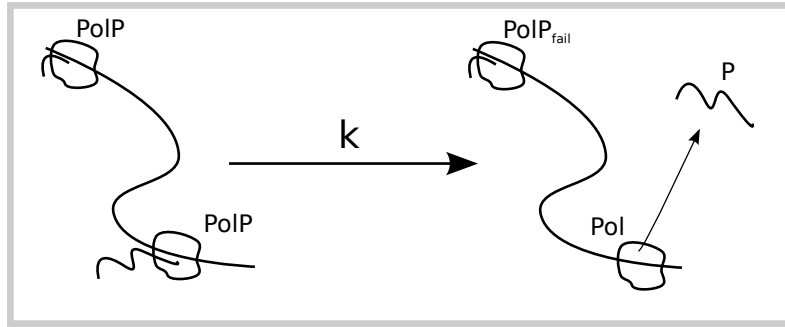
Action Each instance of L reads a specific template. Using its `LoadingTable`, we know which E it tries to load, which LE is yielded if loading occurs and the loading rate associated with the template. When the reaction is performed, a random L is chosen according to loading rates. An element to load 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

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

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 .

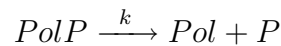


2.3.7 Release

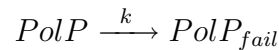
Input format

```
TransformationTable <name> [<parent_letter> <product_letter>,<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



or



where

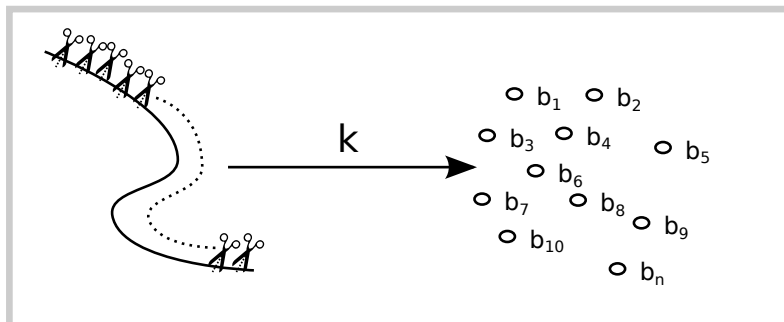
- $PolP$ is a `BoundChemical` representing a polymerase-product complex.
- P is of type `ChemicalSequence`. It is a product that is released by $PolP$ defined in a `ProductTable` associated with reaction.
- Pol is a `BoundChemical` representing an empty polymerase.
- $PolP_{fail}$ is a `BoundChemical` 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

$$\lambda = k[PolP]$$

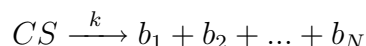
2.3.8 Degradation



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



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 Solver loop

2.4.1 Solver class

2.5 Input/Output handling

3 Detailed design

3.1 Reactants

3.1.1 FreeChemical

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

3.1.2 BoundChemical

`BoundChemical` represents molecules of the same chemical species, but there are specificities for each unit of a `BoundChemical`, as all units are bound at different locations of different `ChemicalSequence` (Fig. 2). 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. 2).

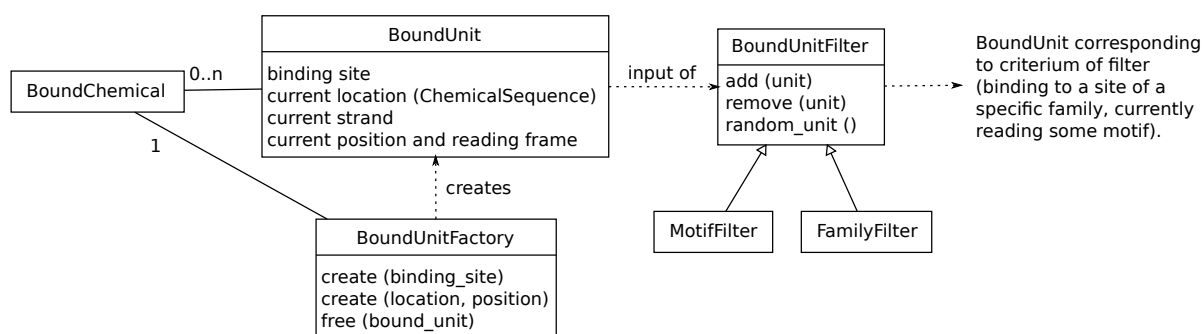


Figure 2: `BoundChemical` are in fact a pool of individual `BoundUnit` created using a `BoundUnitFactory`. A `BoundUnit` is characterized by the `ChemicalSequence` it bound to and its current position. Reaction then use `BoundUnitFilter` to sort `BoundUnit` according to some criterium of reference (*e.g.* Loading reactions sort `BoundUnit` according to the motif they read).

`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`.

3.1.3 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.

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. 3).

3.1.4 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 substrate associated with each binding site. In order to update the rate properly when availability of sites changes, an *observer pattern* is used (Fig. 4).

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.

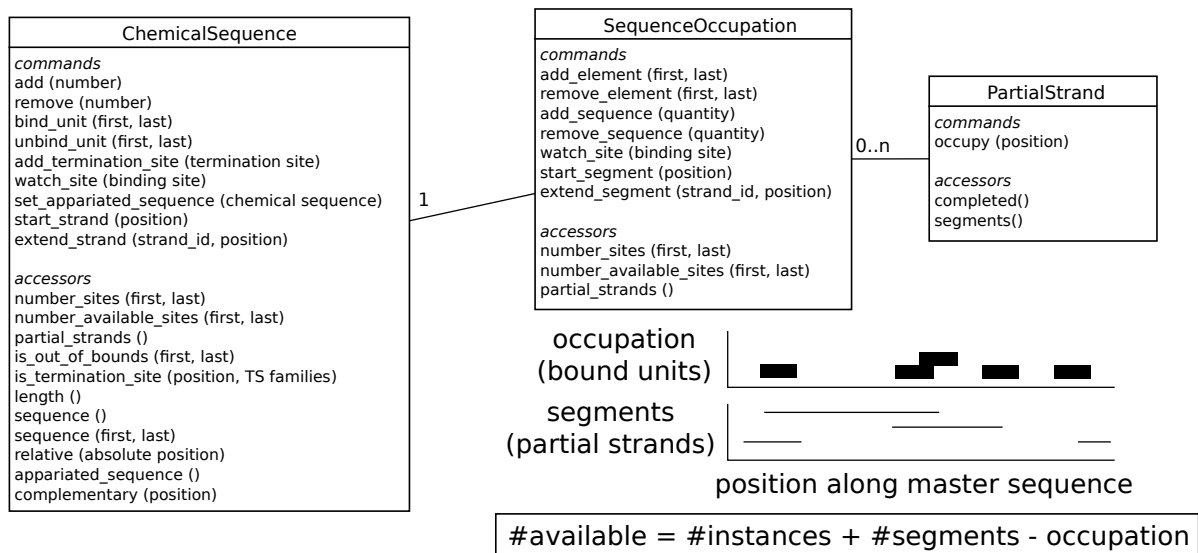


Figure 3: **ChemicalSequence** represents a pool of polymeres that can be elongated and on which **BoundUnits** bind through **BindingSites**. For binding to occur, availability of **BindingSite** is assessed using a utility class **SequenceOccupation** that records the number of instances of the polymer, the position of **BoundUnits** and elongation of **PartialStrands**.

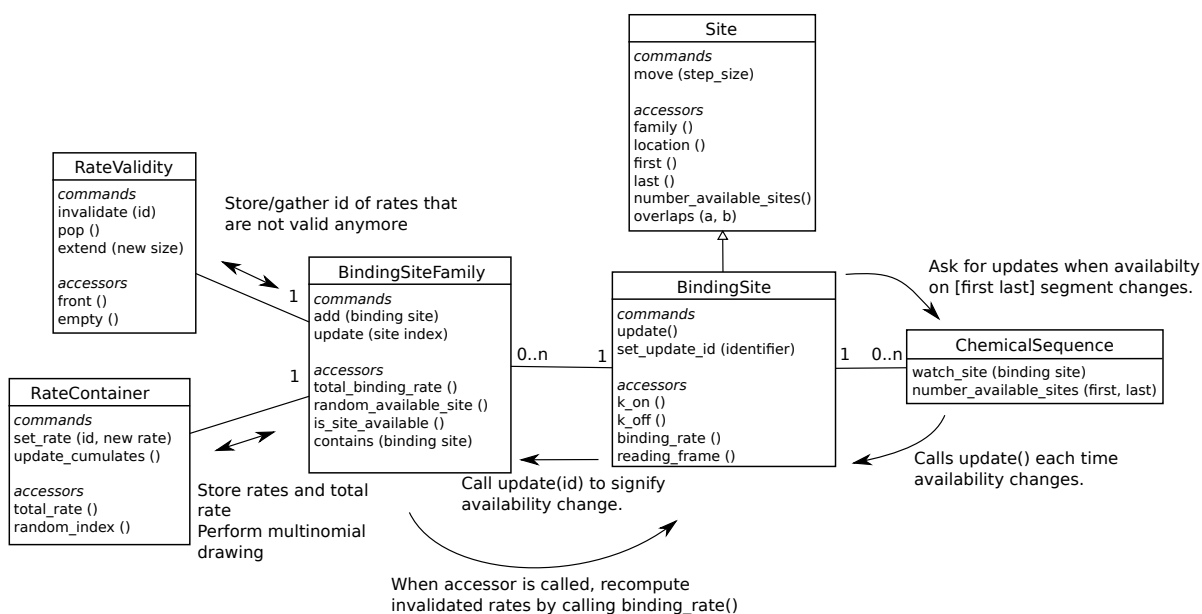


Figure 4: Schematical view of the Observer pattern used to keep availability of binding sites up to date for **SequenceBinding** reactions.

3.2 Reactions

3.2.1 ChemicalReaction

Nothing particular.

3.2.2 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 occurred 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.

3.2.3 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.
- `Translocation` enters stalled form if a `BoundUnit` reaches a termination site *after the translocation was completed*.

3.2.4 Loading

Handling each polymerase individually The main challenge with `Loading` is to maintain the substrates associated with each motif up to date. It needs to maintain a list of all `BoundUnits` reading a specifying 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. 5).

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

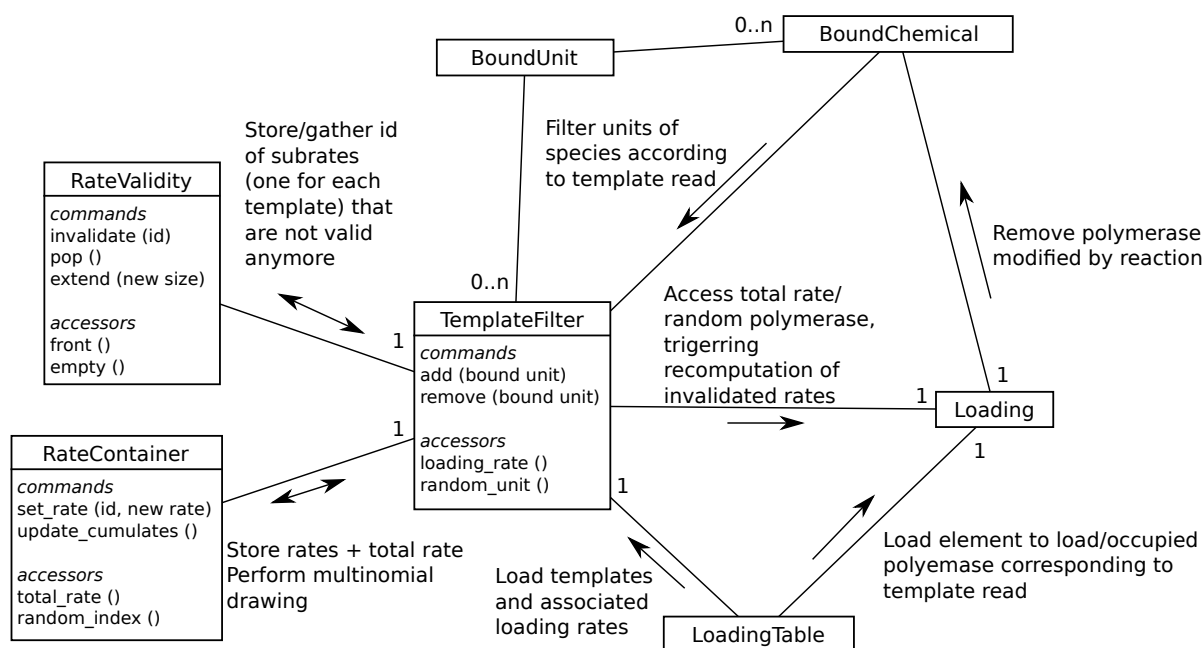


Figure 5: Schematical view of the pattern used to keep substrates associated with each template up to date in a **Loading** reaction.

synthesized. In this case, the **DoubleStrandLoading** fails and the polymerase is replaced by the polymerase in its stalled form.

3.2.5 Release

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

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

3.3 Solver loop

3.3.1 Rate container classes

Implementation See CATI file.

A Tests

A.1 Testing philosophy

Tests are usually divided in several categories. Because of the size of the project, the program includes three types of tests: *programming by contract*, *unit tests*, *integration tests* (Tab. 1). They are designed to make the program fail as rapidly as possible and help find the origin of the problem.

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

A.1.1 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 interact correctly with the outside world, generally other classes.

Preconditions Preconditions.

A.1.2 Unit tests

A.1.3 Integration tests

A.2 Organizing and running tests