Laboratorio de Inteligencia Artificial

Universidad Politécnica de Madrid

gro 3.0 documentation

Language reference

Code version: alpha 3.0.0

Document version: 1.0.1

AUTOR: Elena Núñez Berrueco

Jan 2024

Contents

List of Figures

List of Tables

Acronyms

DDE delayed differential equation

GFP green fluorescent protein

GUI graphical user interface

JSON JavaScript Object Notation

ODE ordinary differential equation

RGB reed, green, blue

SDE stochastic differential equation

T4SS type IV secretion system

Chapter 1

Language reference: an overview

1.1 The graph-based language

Graph paradigm The design philosophy behind the novel gro language places a premium on user-friendliness, striving to render the simulator accessible to individuals without programming expertise while streamlining experiment specification for all users. While the original CCL language once played a major role, it has since been relegated to a minor role in the new language. The syntax of gro 3.0 draws inspiration from visual scripting, although a graphical interface for experiment specification remains pending. Nevertheless, the language has been entirely aligned with this paradigm, facilitating the prospective development of a graphical interface as a straightforward task.

The foundational concept behind the new language revolves around portraying a network of interacting entities such as genetic elements, cells, and signals as a graph, termed the simulation graph. Within this construct, each entity serves as a node visually depicted as a box housing two primary categories of attributes: (i) user-defined parameter values and (ii) edges to related elements. The `.gro` file is a direct textual representation of this graph, exemplified in Figure 1.1. Each element is stored as a dictionary wherein property names function as keys, containing user-defined parameter values alongside references to other interlinked elements.

The basic structure of these dictionaries adheres to the following format:

```
element_type_keyword([ name := "name of this element"
  , one_parameter := value
  , another_parameter := another_value
  , referenced_element := "name of another element"
]);
```

An independent dictionary for each element The initial implementation of this dictionary-style structure debuted in gro 2, primarily utilized for delineating `Operons`. Initially, elements like `Promoters` and `Proteins` were subsumed under the purview of the `Operon`, their descriptions nested within it. However, within gro 3.0, each element operates independently, facilitating linkage with others without ownership (refer to **??**). This paradigm shift enables multiple `Operons` to share the same `Promoter`, establishing unequivocal identity amidst similar parameter values. `Promoters` and `Proteins` are defined just once and potentially reusable across distinct `Operons`. Moreover, users have the flexibility to describe a collection of `Promoters` and `Molecules` in a separate file, constituting a comprehensive genetic repository. Subsequently, these components can be amalgamated into various `Operons` within diverse experiment files.

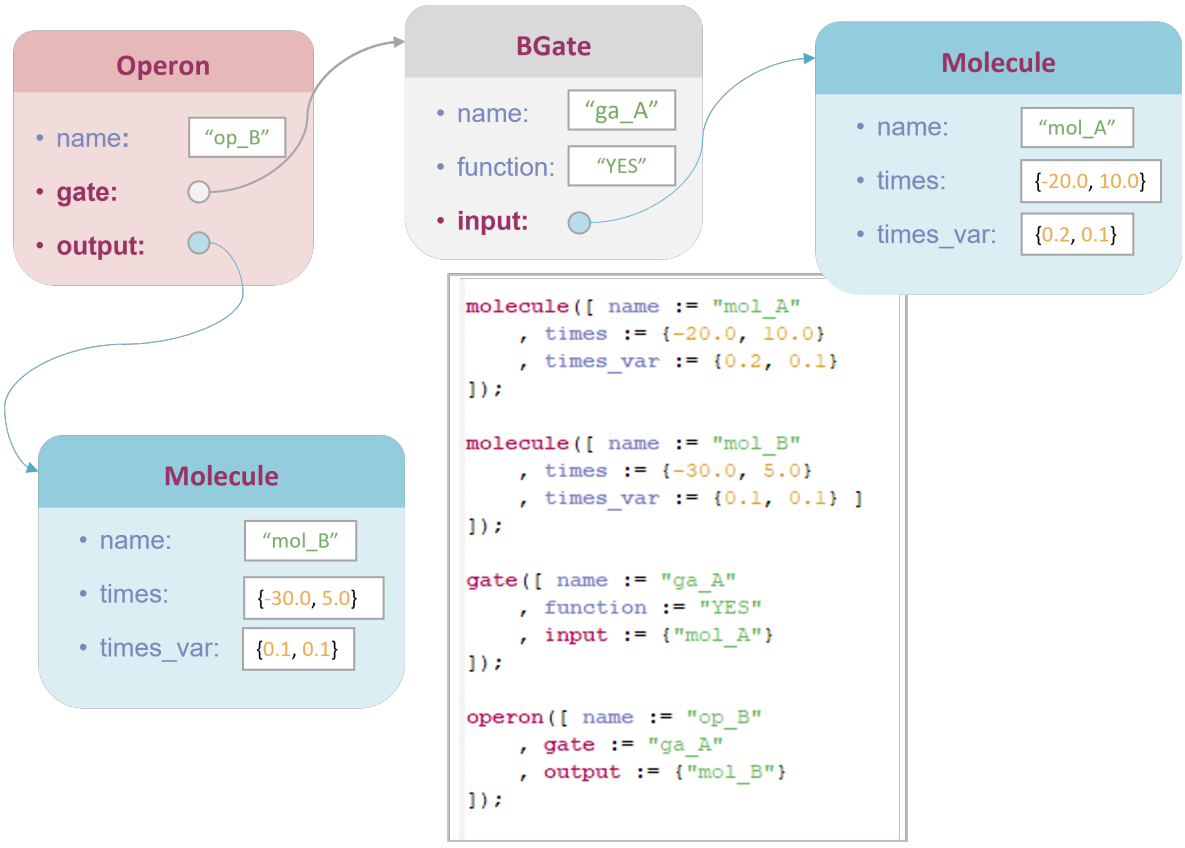

Figure 1.1: **Simulation graph in visual and text formats.**

The order in which dictionaries appear within the `.gro` file doesn't hold any significance; elements can be referenced prior to their description. This flexibility empowers users to modify and organize input data according to their requisites, unencumbered by the order of appearance. This feature significantly simplifies the depiction of intricate systems involving numerous interacting components. Likewise, the specific order of fields within each dictionary does not affect the outcome.

Although akin to JavaScript Object Notation (JSON), the format in use deviates slightly. Retaining elements of the original CCL language proffers certain advantages. Firstly, users can conveniently define pivotal simulation parameters at the beginning of the document, facilitating easy adjustments. This also streamlines batch execution of simulations. Secondly, users possess the capability to conduct fundamental arithmetic operations using these parameters. Additionally, the "include" directive enables the extension of simulations across multiple files and the creation of reusable library `.gro` files.

1.2 Accessory software

gro language pack for Notepad++ For a smoother experience in creating experiment files, users can integrate a language pack into Notepad++. Locate the necessary files within the "Files/gro language for Notepad++" directory. Activating syntax highlighting involves importing the `"gro3lang.xml"` file into Notepad++ as a custom language. Access this by navigating to `"Language -> User defined language -> Define your language..."`, selecting `"Import..."`, and choosing the specified file. Moreover, to automate code snippet generation for dictionaries, resembling LaTeX editors, utilize the Snippets plug-in. Import the `"gro3_snippets.sqlite"` file via the plug-in menu by selecting `"Import library"`. Additionally, find supplementary, easily understandable language reference files within the same folder.

1.3 Details

Naming rules As the name serves as an identifier, each element's name must be unique. If a name is repeated, the previous elements with the same name are overridden, so that only the last element is included in the simulation. The name is mandatory only for elements that are referenced by other elements; otherwise, it is optional. Users provide the name of the element as a string, which can include characters, numbers, symbols, and even spaces. The only restriction is that it should not start with an underscore, as this is reserved for automatically generated elements. Names, like everything else, are case-sensitive.

Although not necessary, it is a good practice to prefix the names with an abbreviation of the element's type to always make it clear what they are, and to reduce the chances of mistakenly repeating a name. The recommended prefixes for each element type are detailed within their respective sections in this document, as well as in the summary tables.

Input checking The validation process involves identifying syntax errors, ensuring mandatory fields are present, verifying references to defined elements of correct types, and validating values in categorical fields. Numerical data is not checked and, following CCL's behaviour, is not automatically converted between floats and integers. Thus, it is crucial that values precisely match the data type of the corresponding field.

Categories of elements Within the gro 2 language, seven elements were defined as dictionaries or similar constructs: the four genetic elements (`Protein`, `Promoter`, `Operon`, `Plasmid`), along with `Action`. Additionally, two elements (`Signal`, `Ecolis`) were inherited from gro 1. In this updated version, the original elements have been

maintained, albeit modified, except for `Action`, which has been substituted by distinct elements tailored for each type of action.

In gro 3.0, there are now over 30 elements categorized into four groups:

1. **Mathematical and logical elements:** This group encompasses random variables, logic gates, and user-defined quantitative conditions (equations and inequations), functions, and differential equations. They form the foundational logic for the other three categories. While most of these elements possess defaults tailored for standard simulations, preventing basic users from direct interaction, they remain accessible for users aiming to implement custom behaviours. Refer to **??** for a summary of these elements.

2. **Biological elements:** Encompassing modified and extended versions of elements from gro 2 alongside numerous new elements catering to processes like plasmid replication, horizontal gene transfer, mutation, and metabolism. Some of these elements replace gro 2 actions. Refer to **??** for a summary.

3. **Medium elements:** Comprising extracellular signals and the grid where they reside and diffuse. These elements are summarized in **??**.

4. **Elements for simulation control:** This group comprises elements facilitating cell and signal placement within the world, as well as generating statistics, output files, plots, and images. Unlike gro 2, where the original gro language was essential for these functions, gro 3.0 incorporates these elements directly for user convenience. Find a summary of these elements in **??**.

States and markers Every cellular element within categories 1 and 2 is associated with a primary state variable. This variable is employed when referenced as input for a logic element like a `Function` or `Gate`. Generally, the meaning of this variable is self-explanatory. For physical elements such as `Molecules` and `Plasmids`, their primary state denotes their presence in the cell (or their copy number in the case of a quantitative representation). Elements representing processes (like metabolic `Flux`) exhibit their activity level (the exchanged amount) as their main state. For logic elements, it signifies the output of their computation. Tables **??** and **??** furnish details about the state for each element.

Certain elements permit the exposure of additional state variables, typically utilized internally. This is achieved through their "marker" fields, enabling users to create an identifier for accessing that value from other elements. The presence and usage of these fields are expounded upon in each element's section.

Types of parameters Elements in the model possess two types of non-relation attributes: numerical and categorical. Categorical attributes encompass a finite set of possible string values, while numerical attributes can be either free parameters or dependent parameters. Free parameters are independent parameters reliant solely on user input and are shared by all cells. In contrast, dependent parameters rely on the state value of other elements and may constitute a custom function involving one or more state values. The value of dependent parameters can differ from cell to cell.

These parameters can either be deterministic or stochastic. Free stochastic parameters are independent probability distributions customizable in terms of their family and re-sampling schedule. To introduce stochasticity to dependent parameters, users should utilize stochastic functions that incorporate normal (or other family) noise into the calculation.

1.4 Complete example

`.gro` file structure and simple example A `.gro` experiment typically adheres to a four-part structure:

- **Header:** Comprising "include" statements, main parameter definitions, and global parameters.
- **Design of the Biocircuits:** Defining cellular components and their relationships.
- **Creation of Cells:** Determining the components each cell type carries and their placement in the world.
- **Visualization:** Specifying how states are visually reported (cellular colour) or through plots and output files.

The specification of a simple Repressilator circuit with digital gene expression is employed as a basic experiment to exemplify these parts.

Header The header section incorporates the essential "include" statements, typical in the CCL language, enabling the utilization of external files as libraries. By default, `"gro3"` is employed. It proves convenient to define crucial parameters, subject to fine-tuning, in this section. The `"global_params"` block is mandatory for all `.gro` files and serves as the space to set general parameters not tied to specific elements. A frequently modified parameter is the random seed used for generating replicates. In Example 1.1, the primary parameters concern the expression times of repressor proteins, encompassing mean and variability.

Example 1.1: Simple Repressilator example, header

```
include gro3

//params
t_act := 10.0;
t_act_var := 1.0;
t_deg := 50.0;
t_deg_var := 5.0;

global_params([ seed := 133087 ]);
```

Design of the biocircuits In this section, each cellular component, whether biological or logical, capable of existing within cells is delineated by assigning values to its parameters and establishing links with other components. In Example 1.2, three digitally represented `Molecules` (corresponding to three repressor proteins) and their respective `Operons` producing them are depicted. For simulating conditional expression, the `Operons` employ logic `Gates` as `Promoters`. These three `Operons` are then integrated into a plasmid with digital representation (`BPlasmid`), as the quantitative dynamics of plasmids are not relevant for this simple example.

Example 1.2: Simple Repressilator example, description of components

```
molecule([ name := "mol_A", times := {-t_deg, t_act },
   times_var := { t_deg_var, t_act_var } ]);
molecule([ name := "mol_B", times := {-t_deg, t_act },
   times_var := { t_deg_var, t_act_var } ]);
molecule([ name := "mol_C", times := {-t_deg, t_act },
   times_var := { t_deg_var, t_act_var } ]);

bgate([ name := "ga_notMolA", input := {"-mol_A"} ]);
bgate([ name := "ga_notMolB", input := {"-mol_B"} ]);
bgate([ name := "ga_notMolC", input := {"-mol_C"} ]);

operon([ name := "op_A", gate := "ga_notMolC", output := {"
   mol_A"} ]);
operon([ name := "op_B", gate := "ga_notMolA", output := {"
   mol_B"} ]);
operon([ name := "op_C", gate := "ga_notMolB", output := {"
   mol_C"} ]);
```

```
plasmid([ name := "p_ABC", operons := { "op_A", "op_B", "
   op_C" } ]);
```

Creation of cells At the initiation of the simulation, distinct `CellTypes` can harbour different subsets of cellular components, and these may be altered as the simulation advances. `CellPlacers` come into play to introduce cells into the simulation at specific times and locations. In the provided simple example (Example 1.3), two `CellTypes` are delineated, both carrying the plasmid with the genetic circuit. However, they contrast in the repressor protein initially present, initiating their oscillation processes at different points. To situate 100 cells at the onset of the simulation with the two foci spatially separated, one `CellPlacer` is employed for each `CellType`.

Example 1.3: Simple Repressilator example, description of cell types

```
cell_type([ name := "cell_A", plasmids := {"p_ABC"},
   molecules := {"mol_A"} ]);
cell_type([ name := "cell_B", plasmids := {"p_ABC"},
   molecules := {"mol_B"} ]);

cell_placer([ name := "cp_A", cell_types := {"cell_A"},
   amount := 100.0, coords := [ x := -100.0 ] ]);
cell_placer([ name := "cp_B", cell_types := {"cell_B"},
   amount := 100.0, coords := [ x := 100.0 ] ]);
```

Visualization In this section, users define how cellular states are visualized, indicating colours and specifying the output data to collect in real-time `Plots` or text `OutFiles`. In the provided example, the presence of each of the three `Molecules` is depicted using distinct predefined primary colours. When more than one molecule is present simultaneously, their colours are amalgamated. Additionally, the fraction of cells expressing each of the three repressors is charted in real-time on the graphical user interface (GUI).

Example 1.4: Simple Repressilator example, visualization

```
cell_colour([ name := "ccol_green", gate := "mol_A", rgb :=
    _green ]);
cell_colour([ name := "ccol_red", gate := "mol_B", rgb :=
   _red ]);
```

```
cell_colour([ name := "ccol_blue", gate := "mol_C", rgb :=
   _blue ]);

plot([ name := "plot_ABC", fields := {"mol_A", "mol_B", "
   mol_C"}, stats := "avg" ]);
```

Full example The full code of the example is listed below.

Example 1.5: Simple Repressilator example, full

```
//---header
include gro3

//params
t_act := 10.0;
t_act_var := 1.0;
t_deg := 50.0;
t_deg_var := 5.0;

global_params([ seed := 133087 ]);


//---Design of the biocircuits
molecule([ name := "mol_A", times := {-t_deg, t_act },
   times_var := { t_deg_var, t_act_var } ]);
molecule([ name := "mol_B", times := {-t_deg, t_act },
   times_var := { t_deg_var, t_act_var } ]);
molecule([ name := "mol_C", times := {-t_deg, t_act },
   times_var := { t_deg_var, t_act_var } ]);

bgate([ name := "ga_notMolA", input := {"-mol_A"} ]);
bgate([ name := "ga_notMolB", input := {"-mol_B"} ]);
bgate([ name := "ga_notMolC", input := {"-mol_C"} ]);

operon([ name := "op_A", gate := "ga_notMolC", output := {"
   mol_A"} ]);
operon([ name := "op_B", gate := "ga_notMolA", output := {"
   mol_B"} ]);
```

```
operon([ name := "op_C", gate := "ga_notMolB", output := {"
    mol_C"} ]);
plasmid([ name := "p_ABC", operons := { "op_A", "op_B", "
    op_C" } ]);


//---Creation of cells
cell_type([ name := "cell_A", plasmids := {"p_ABC"},
    molecules := {"mol_A"} ]);
cell_type([ name := "cell_B", plasmids := {"p_ABC"},
    molecules := {"mol_B"} ]);

cell_placer([ name := "cp_A", cell_types := {"cell_A"},
    amount := 100.0, coords := [ x := -100.0 ] ]);
cell_placer([ name := "cp_B", cell_types := {"cell_B"},
    amount := 100.0, coords := [ x := 100.0 ] ]);


//---Visualization
cell_colour([ name := "ccol_green", gate := "mol_A", rgb :=
     _green ]);
cell_colour([ name := "ccol_red", gate := "mol_B", rgb :=
    _red ]);
cell_colour([ name := "ccol_blue", gate := "mol_C", rgb :=
    _blue ]);

plot([ name := "plot_ABC", fields := {"mol_A", "mol_B", "
    mol_C"}, stats := "avg" ]);
```

Chapter 2

Cellular logic elements

The elements in this group are presented first because they are integral to the logic of most elements in the other three groups. However, the `Boolean Gate` (`BGate`) is the only one expected to be explicitly used by basic users. The other logic elements are typically automatically generated with default values behind the scenes and usually only require modification to achieve non-standard behaviours in simulations. Refer to **??** for a summary of all the elements in this category.

2.1 Stochasticity

Stochastic elements encompass the `Randomness` and a simpler distribution dictionary. The `Randomness` (Subsection 2.1.2) is a sophisticated cellular element linked to others, dictating their stochastic behaviour throughout the simulation to generate heterogeneity. In contrast, the distribution dictionary (Subsection 2.1.1) is a data structure that stores information about a distribution utilized for one-time events at the global level, such as placing new cells in the world.

2.1.1 The distribution dictionary

Keywords: _
Recommended prefixes: _
Linked to (direct): none
Linked to (reverse): `QPlasmid`, `Strain`, `CellPlacer`, `CellPlating`, `SignalPlacer`

Description A probability distribution. This is not a standalone element but a dictionary parameter that appears as a field in many other elements, both at the cellular level and at the simulation level. For convenience, it can be specified as if it were an element. The complete list of fields for the distribution dictionary is provided in Table 2.2.

Fields The dictionary structure consists of two fields: `type` and `params`. `type` signifies the family of the distribution, with currently available options being `"normal"`, `"uniform"`, and `"exponential"`. `params` is utilized to provide a list of real values serving as the parameters of the distribution. The meaning and default values of these parameters depend on the `type`. Table 2.1 summarizes this information.

Table 2.1: Meaning and default values of distribution parameters

TYPE	PARAMETERS	DEFAULT
normal	{ mean, stddev }	{ 0.0, 1.0 }
uniform	{ lbound, ubound }	{ 0.0, 1.0 }
exponential	{ lambda }	{ 1.0 }

Example Example 2.1 demonstrates one application of the distribution dictionary. The element is a `CellPlacer`, and the distribution is utilized to sample the number of cells placed, using a uniform distribution within the range [1, 10] in this case. `"cp_A"`, `"cp_same"`, and `"cp_also_same"` exemplify equivalent methods of specifying the distribution, with one leveraging the underlying variables system based on the CCL language.

Example 2.1: Distribution dictionary applied to a CellPlacer

```
cell_type([ name := "cell_A" ]);

cell_placer([ name := "cp_A"
  , cell_types := {"cell_A"}
  , amount_dist := [ type := "uniform", params := { 1.0, 10
    .0 } ]
]);

myDist := [ type := "uniform"
  , params := { 1.0, 10.0 } ];

cell_placer([ name := "cp_same"
  , cell_types := {"cell_A"}
  , amount_dist := myDist
]);

distribution([ name := "dist_myDist"
  , type := "uniform"
  , params := { 1.0, 10.0 }
]);

cell_placer([ name := "cp_also_same"
  , cell_types := {"cell_A"}
  , amount_dist := "dist_myDist"
]);
```

Table 2.2: Distribution dictionary

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
type	string in {"normal", "uniform", "exponential"}	—	The type of continuous probability distribution	"normal"
params	array of real	_	Parameters of the distribution. If empty, the default parameters for each distribution are used. Normal: {mean, stddev}, uniform: {lbound, ubound}exponential: {lambda}	normal: { 0.0,1.0 }, uniform: { 0.0, 1.0 }, exponential: { 1.0 }

2.1.2 Randomness

Keywords: `randomness`
Recommended prefixes: `rnd_`
Linked to (direct): none
Linked to (reverse): most cellular elements, both logic and biological (`Function`, `Ode`, `Historic`, `CellColour`, `Molecule`, `Regulation`, `Flux`, `OriV`, `CopyControl`, `Partition`, `OriT`, `Pilus`, `Plasmid`, `Strain`) and reporting global elements (`Population Stat`, `Out File`, `Plot`)

Description A `Randomness` is an advanced element designed for fine-tuning the stochastic behaviour of any linked cellular element. This includes specifying the distribution type and determining how random values are sampled and inherited. The complete list of fields for the `Randomness` is provided in Figure 2.1 and Table 2.3.

Detailed description A `Randomness` element embodies the stochastic behaviour of another element, comprising a distribution, a re-sampling mechanism, and an inheritance schedule. It finds application in various processes, ranging from gene expression to metabolic fluxes and mutations. This system is not employed for distributions sampled only once during the initialization of new cells (not from cell division), such as the initial volume or the initial copy numbers of plasmids. In such cases, a distribution dictionary (as described in the previous section) is utilized.

The `Randomness` element accounts for population variability arising from factors not explicitly modelled. Essentially, it represents a random variable with inertia, generating values that are not independent of each other. Each new value is a linear combination of the previous one and a new sample. Implementing a `Randomness` instead of a simple random variable helps avoid sudden and unrealistic changes in stochastic processes.

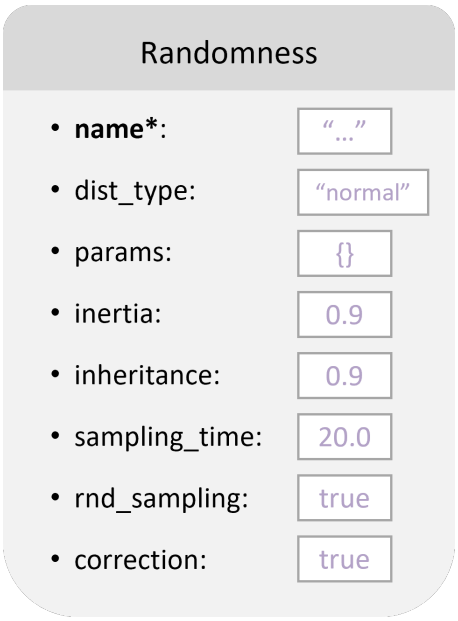

Figure 2.1: **Randomness element**

Moreover, it results in cells within the same lineage having more similar values than unrelated ones. The `Randomness` can be shared with other elements to correlate them. Multiple `Randomness` elements can be combined to consider different components of noise that may be inherited in distinct ways. Furthermore, changes introduced in the parameters of these processes can be leveraged for evolution and/or adaptation experiments.

Probability distribution The `dist_type` field denotes the probability distribution family of the random variable. The default value is `"normal"`, as this distribution is commonly employed to model variability among a large number of individuals. Furthermore, the normal distribution is the only one for which the resulting distribution remains normal with the same parameters. The available alternative distributions are `"uniform"` and `"exponential"`. When used together with the inheritance features of the `Randomness` element, these distributions undergo changes. For instance, a uniform distribution transforms into a triangular distribution. Users are advised to consider this before utilizing any distribution other than the normal.

The field named `params` contains the parameters of the distribution. The number and interpretation of these parameters depend on the type of distribution. For the normal distribution, there are two parameters: mean and standard deviation (`mean, standard deviation`), and for the uniform distribution, there are also two: lower

bound (inclusive) and upper bound (exclusive) (`lower bound, upper bound`). The exponential distribution has a single parameter: the rate (`rate`). By default, the standard versions of the distributions are used: mean 0 and deviation 1 for the normal distribution; lower bound 0 and upper bound 1 for the uniform distribution; and lambda 1 for the exponential. Refer to Table 2.1 for a summary of this information.

Internal vs external scaling Assigning distribution parameters at the `Randomness` level via the `params` field is atypical. The preferred approach is to retain the standard distribution in the `Randomness` and then scale the sampled values in the connected elements (`Molecule`, `Regulation`, `Flux`, etc.). All elements capable of utilizing a `Randomness` feature a field for this purpose. This methodology enables multiple elements to share the same `Randomness`, fostering correlation, even when distinct parameter values are required.

Example 2.2 demonstrates the utilization of a normal `Randomness` for introducing variability in activation and degradation times of a `Molecule` (see section 3.1.1 for details). With a mean of 5.0 and a deviation of 0.1 directly assigned to the `Randomness`, `"mol_bad"` does not exhibit the expected values for the times. This discrepancy arises because a non-standard normal distribution is scaled as if it were standard. `"mol_good"` illustrates how to adjust the time parameters to align with the specified values. While feasible, this is not the recommended usage of `Randomness`. `"mol_typical"` employs an automatically generated `Randomness` with default parameters (i.e., a standard normal). The values supplied to the time parameters are utilized to scale the samples from the standard normal, eliminating the need for additional calculations. The latter is the recommended approach.

Example 2.2: Randomness scaling

```
t_acti := 10.0;
t_deg := 20.0;
t_acti_var := 0.1;
t_deg_var := 0.2;

randomness([ name := "rnd_A"
, params := {5.0, 0.1}
]);

molecule([ name := "mol_bad"
  , rnd := "rnd_A"
  , times := {-t_deg, t_acti}
  , times_var := {t_deg_var, t_acti_var}
```

```
]);

molecule([ name := "mol_good"
  , rnd := "rnd_A"
  , times := {-(t_deg - 5.0), (t_acti - 5.0)}
  , times_var := {t_deg_var / 0.1, t_acti_var / 0.1}
]);

molecule([ name := "mol_typical"
  , times := {-t_deg, t_acti}
  , times_var := {t_deg_var, t_acti_var}
]);
```

Correlations using a shared `Randomness` Example 2.2 illustrates two `Molecules` sharing the same `Randomness`. As no fields have been overridden, it is a standard normal `Randomness`. The variability in activation and degradation times of the two `Molecules` is interconnected through the shared `Randomness`, establishing a correlation. Given that their deviation for degradation time is identical, both `Molecules` will experience precisely the same amount of noise applied to the degradation time. For instance, when the common `Randomness` yields a value of 10.0, the applied deviation from the mean degradation time is $10.0 \cdot 0.2 = 2.0$ minutes. The degradation time would be $20.0 + 2.0 = 22.0$ minutes for `"mol_A"` and $30.0 + 2.0 = 32.0$ minutes for `"mol_B"`. In the case of the activation times, their variability is proportional but not identical, as their standard deviations differ. For `"mol_A"`, the activation time would be $15.0 + 10.0 \cdot 0.05 = 15.5$ minutes; for `"mol_B"`, the activation time would be $20.0 + 10.0 \cdot 0.1 = 21.0$ minutes.

Example 2.3: Shared Randomness

```
randomness([ name := "rnd_common" ]);

molecule([ name := "mol_A"
  , rnd := "rnd_common"
  , times := {-20.0, 15.0}
  , times_var := {0.2, 0.05}
]);

molecule([ name := "mol_B"
  , rnd := "rnd_common"
```

```
  , times := {-30.0, 20.0}
  , times_var := {0.2, 0.1}
]);
```

Inertia and inheritance The `Randomness` is resampled on cell division. The value assigned to each of the daughter cells is calculated by the formula:

$$\begin{aligned} val_{daughter1} &= inheritance \cdot val_{mother} + (1 - inheritance) \cdot newsample_1 \\ val_{daughter2} &= inheritance \cdot val_{mother} + (1 - inheritance) \cdot newsample_2 \end{aligned} \tag{2.1}$$

Example 2.4 showcases a `Randomness` with the inertia and inheritance fields overridden. Suppose the mother cell has a current value of 2.0, resulting in a deviation in the activation time of $2.0 \cdot 0.1 = 0.2$ minutes (activation time is $15.0 + 0.2 = 15.2$ minutes). If new samples of 3.0 and -0.5 are drawn on cell division, then the variability values of the daughter cells would be $0.1 \cdot (0.7 \cdot 2.0 + 0.3 \cdot 3.0) = 0.23$ (activation time of $15.0 + 0.23 = 15.23$ minutes) for one of them and $0.1 \cdot (0.7 \cdot 2.0 + 0.3 \cdot (-0.5)) = 0.125$ (activation time of $15.0 + 0.125 = 15.125$ minutes) for the other. The behaviour of gro 2 is equivalent to an inheritance of 1.0.

The concept of inertia (field `inertia`) is analogous to that of inheritance. The calculation is identical, replacing the inheritance parameter with the inertia parameter, as expressed by the equation:

$$val_{new} = inertia \cdot val_{old} + (1 - inertia) \cdot newsample \tag{2.2}$$

The distinction lies in the fact that inertia pertains to the resampling events that occur spontaneously at any point in the cell cycle.

Sampling time The `sampling_time` field represents the time between sampling events. If `rnd_sampling` is set to false, the time between events is deterministic and always equal to `sampling_time`. By default, `rnd_sampling` is true, signifying that the time is exponentially distributed (a Poisson process), with `sampling_time` as the lambda parameter or average waiting time. In Example 2.4, that average time is 100 minutes. A negative value in this field implies no resampling, i.e., only modifying randomness on cell division. The behaviour of gro 2 is equivalent to an inertia of 0.0.

Example 2.4: Complete Randomness

```
randomness([ name := "rnd_A"
```

```
  , dist_type := "uniform"
  , inertia := 0.5
  , inheritance := 0.7
  , sampling_time := 100.0
  , rnd_sampling := true
]);
```

Correction of the standard deviation The application of inertia and inheritance modifies the distribution, producing that the observed distribution at the population scale does not match the specified one. The `correction` is true by default, indicating that a correction is applied to normal distributions to ensure that the observed distribution aligns with the specified one. If `correction` is set to false, the resulting distribution remains normal, with the same mean, but the standard deviation is reduced. The higher the inheritance and/or inertia, the greater the reduction. The correction only applies to normal distributions. For other distribution families, the resulting distribution is not guaranteed to be of the same family if the inertia and/or the inheritance parameters are not zero.

The provided examples revolve around the usage of `Randomness` for the activation and degradation times of `Molecules`. Examples involving `Randomness` for noise in gene expression, metabolic fluxes, and custom `Functions` can be found in **??**, Subsection 3.2.1, and Subsection 2.2.1, respectively.

2.2 Quantitative calculations

This section encompasses elements utilized to formulate arbitrary mathematical functions over the state of any cellular elements (`Function` Subsection 2.2.1), along with differential equations of diverse types, including ordinary differential equations (ODEs) (`Ode` Subsection 2.2.2), stochastic differential equations (SDEs) (`Ode`, Subsection 2.2.2), and delayed differential equations (DDEs) (`Ode`Subsection 2.2.2, `Historic` Subsection 2.2.3).

2.2.1 Function

Keywords: `function`
Recommended prefixes: `fun_`, `f_`
Linked to (direct): any cellular element, logic or biological, including other `Functions`
Linked to (reverse): Most cellular elements, logic (`Function`, `Ode`, `Historic`, `QGate`,

Table 2.3: Parameters of the Randomness element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
dist_type	string in {"normal", "uniform", "exponential"}	_	The distribution type that the random variable follows	"normal"
params	array of real	_	Parameters of the distribution. Usually not used; scaled in the linked elements instead	{} no params (standard versions of the distributions). Normal: { 0.0, 1.0 }, uniform: { 0.0, 1.0 }, exponential: { 1.0 }
inertia	real in [0, 1]	_	Tendency of the random variable to change smoothly, avoiding big changes on spontaneous resampling. Current value = inertia * old value + (1- inertia) * new sample	0.9
inheritance	real in [0, 1]	_	Tendency of the random variable to change smoothly, avoiding big changes betwwen mother and daughter cells on cell division. Daughter value = inheritance * mother value + (1- inheritance) * new sample	0.9
sampling_time	real	min	Average time between spontaneous resampling events (exponentially distributed). A negative value means no spontaneous resampling (only on cell division)	20.0
rnd_sampling	Boolean	_	If false, the value is sampled every sampling_time minutes, exactly. If true, the sampling times follow an exponential distribution with lambda = sampling_time	true
correction	Boolean	_	If true, normal distributions use a correction to ensure that the resulting distribution (at population level) has the given parameters. If false, the resulting distribution is also normal but with reduced variance due to the inertia and inheritance.	true

`Cell Colour`) and biological (`Molecule`, `Regulation`, `Flux`, `OriV`, `Copy Control`, `Partition`, `OriT`, `Pilus`, `Plasmid`), and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `Function` element represents a mathematical function applied to one or more input elements. The complete list of fields for the `Function` is provided in Figure 2.2 and Table 2.5.

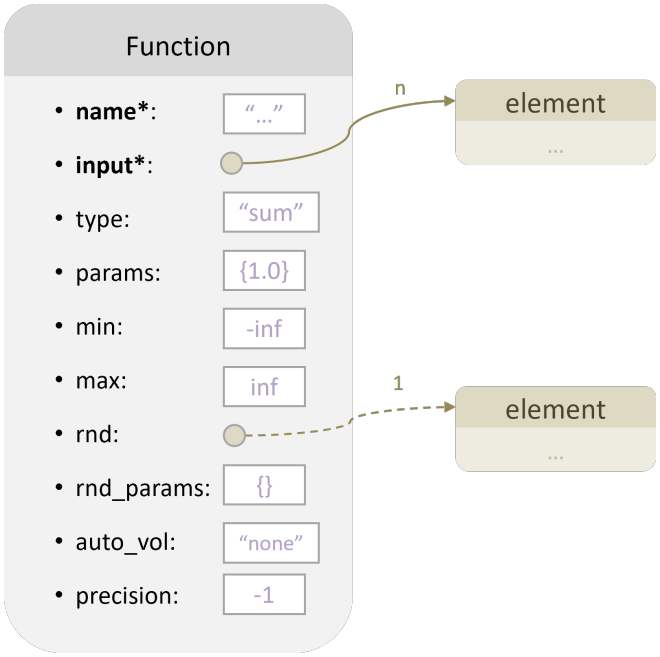

Figure 2.2: **Cell-level Function element**

Input Any logical or biological element serves as a legitimate input for a `Function`, with a preference for those of a quantitative nature. Boolean elements are mapped to 0.0 (for false) and 1.0 (for true). The creation of composite formulas is facilitated by the ability to chain multiple `Functions`. It is imperative to maintain a unidirectional chain, as the presence of any loops would result in undefined behaviour. Assignment of inputs is executed through the obligatory `input` list field, which mandates that each `Function` possess at least one input element.

Mathematical function The mathematical function is delineated by two key fields: `type` and `params`. The `type` field represents the function family, drawn from a set of predefined types explicated below and summarized in Table 2.4. The default function type is the summation or weighted sum (`"sum"`).

`params` constitute real numbers that remain constant throughout the entire simulation. The elucidation of inputs, parameters, and default values is contingent on the specific function. Certain `Function` types allow an unbounded number of inputs, while others accommodate only one or two. The count of parameters is typically aligned with the number of inputs. Any surplus inputs and parameters are disregarded. In cases where an insufficient number of parameter values are specified, the remaining ones are endowed with default values.

Function types The presently available function families include: constant (`"const"`), minimum (`"min"`), maximum (`"max"`), absolute value (`"abs"`), summation or weighted sum (`"sum"`), weighted product (`"product"`), exponential (`"exp"`), logarithm (`"log"`), sigmoid (`"sig"`), Hill function (`"hill"`), combinatorial summation (`"combi_sum"`), saturating exponential (`"sat_exp"`), and exponential product (`"exp_product"`). The subsequent sections expound upon each of these `Function` types, delineating the quantity and meaning of their inputs and parameters, along with their default values. This information is summarised in Table 2.4.

Constant : `"const"` This is the simplest function, merely returning the value of the first input. Any additional inputs or parameters are disregarded. Its use is shown in Example 2.5.

Example 2.5: Constant Function

```
qplasmid([ name := "qp_A" ]);

function([ name := "fun_qpA"
  , type := "const"
  , inputs := {"qp_A"}
]); //"qp_A"
```

Minimum: `"min"` This function computes the minimum among an unbounded number of inputs. It accommodates an optional parameter; if provided, the returned value represents the minimum among all the inputs and that parameter. Its use is exemplified in Example 2.6 and its formula is as follows:

$$output = min(inputs_{0-n}, param_0) \tag{2.3}$$

Maximum: `"max"` This function calculates the maximum among an unbounded number of inputs. It allows for an optional parameter; if provided, the returned value represents the maximum among all the inputs and that parameter. Its use is exemplified in Example 2.6 and its formula is as follows:

$$output = max(inputs_{0-n}, param_0) \tag{2.4}$$

Example 2.6: Minimum and maximum Functions

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_minAB"
  , type := "min"
  , inputs := {"qp_A", "qp_B"}
  , params := {10.0}
]); //min("qp_A", "qp_B", 10.0)

function([ name := "fun_maxAB"
  , type := "max"
  , inputs := {"qp_A", "qp_B"}
  , params := {100.0}
]); //min("qp_A", "qp_B", 100.0)
```

Absolute value: `"abs"` This function computes the absolute value of a single input. Any supplementary inputs or parameters are disregarded. It can be effectively employed in conjunction with a difference operation (refer to the summation function below) to obtain the absolute difference. An illustration of its application is provided in Example 2.7, and its formula is articulated as follows:

$$output = |input_0| \tag{2.5}$$

Example 2.7: Absolute value Function

```
qplasmid([ name := "qp_A" ]);
```

```
qplasmid([ name := "qp_B" ]);

function([ name := "fun_diffAB"
  , type := "sum"
  , inputs := {"qp_A", "qp_B"}
  , params := {1.0, -1.0}
]); //"qp_A" - "qp_B"

function([ name := "fun_absDiffAB"
  , type := "abs"
  , inputs := {"fun_diffAB"}
]); //abs("qp_A" - "qp_B")
```

Summation: `"sum"` This function entails a versatile summation, serving as a weighted sum, first-order polynomial, or linear combination, accommodating an unbounded number of inputs. Each input undergoes multiplication by its designated weight, followed by an aggregation of all products, along with an optional bias term. In the case of n inputs, the first n parameters denote their weights (sequentially), and the $n+1$ parameter represents the bias term. The default value for the bias is 0.0, while the default value for weights is 1.0. Negative weights produce subtraction. An illustration of its application is provided in Example 2.8, and its formula is articulated as follows:

$$output = \sum_{i=0}^{n} param_i * input_i + param_{n+1} \tag{2.6}$$

Example 2.8: Summation Function

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_sumAB"
  , type := "sum"
  , inputs := {"qp_A", "qp_B"}
  , params := {0.4, 0.6}
]); //0.4 * "qp_A" + 0.6 * "qp_B"

function([ name := "fun_withbias"
  , type := "sum"
  , inputs := {"qp_A", "qp_B"}
```

```
  , params := {0.4, 0.6, 2.0}
]); //0.4 * "qp_A" * 0.6 * "qp_B" + 2.0

function([ name := "fun_substactionAB"
  , type := "sum"
  , inputs := {"qp_A", "qp_B"}
  , params := {0.4, -0.6}
]); //0.4 * "qp_A" - 0.6 * "qp_B"
```

Product: `"product"` This function embodies a versatile product, encompassing exponentiated product or polynomial terms, accommodating an unbounded number of inputs. Each input undergoes exponentiation to its designated parameter, followed by the multiplication of all results, along with an optional scale factor. In the case of n inputs, the first n parameters denote their exponents (sequentially), and the $n+1$ parameter signifies the scale factor. The default value for the scale is 1.0, while the default value for exponents is 1.0. Negative exponents lead to division. An illustration of its application is provided in Example 2.9, and its formula is articulated as follows:

$$output = \prod_{i=0}^{n} (input_i^{param_i}) \cdot param_{n+1} \tag{2.7}$$

Example 2.9: Product Function

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_prodAB"
  , type := "product"
  , inputs := {"qp_A", "qp_B"}
  , params := {0.4, 0.6}
]); //"qp_A"^0.4 * "qp_B"^0.6

function([ name := "fun_withscale"
  , type := "product"
  , inputs := {"qp_A", "qp_B"}
  , params := {0.4, 0.6, 1.0}
]); //"qp_A"^0.4 * "qp_B"^0.6 * 2.0

function([ name := "fun_divisionAB"
```

```
  , type := "product"
  , inputs := {"qp_A", "qp_B"}
  , params := {1.0, -1.0}
]); //"qp_A" / "qp_B"
```

Exponential: `"exp"` This function represents an exponential operation with a customizable base. It can be invoked with either two inputs (a), accompanied by a parameter (b), or with just one input (c), with an optional parameter. The default base is e. In instances where more than one input is provided, the parameters are not utilized. An illustration of its application is provided in Example 2.10, and the formulas for the three cases are expressed as follows:

$$\begin{aligned} a) 2 inputs : output &= input_0^{input_1} \\ b) 1 input and 1 param : output &= param_0^{input_0} \\ c) just 1 input : output &= e^{input_0} \end{aligned} \tag{2.8}$$

Example 2.10: Exponential Function

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_expA"
  , type := "exp"
  , inputs := {"qp_A", "qp_B"}
]); //"qp_B" ^ "qp_A"

function([ name := "fun_expB"
  , type := "exp"
  , inputs := {"qp_A"}
  , params := {10.0}
]); //10.0 ^ "qp_A"

function([ name := "fun_expC"
  , type := "exp"
  , inputs := {"qp_A"}
]); // e ^ "qp_A"
```

Logarithm: `"log"` This function entails a logarithmic operation with a customizable base. It can be invoked with either two inputs (a), accompanied by a parameter (b), or with just one input (c), optionally with a parameter. The default base is e. In cases where more than one input is provided, the parameters are not employed. An illustration of its application is provided in Example 2.11, and the formulas for the three cases are expressed as follows:

$$\begin{aligned} a)2inputs : output &= log_{input_1} input_0 \\ b)1inputand1output &= log_{param_0} input_0 \\ c)output &= lninput_0 \end{aligned} \tag{2.9}$$

Example 2.11: Logarithmic Function

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_logA"
  , type := "log"
  , inputs := {"qp_A", "qp_B"}
]); //log base "qp_B" of "qp_A"

function([ name := "fun_logB"
  , type := "log"
  , inputs := {"qp_A"}
  , params := {2.0}
]); //log base 2.0 of "qp_A"

function([ name := "fun_logC"
  , type := "log"
  , inputs := {"qp_A"}
]); //ln of "qp_A"
```

Sigmoid: `"sig"` This is a straightforward sigmoid function that only accepts one input and has no parameters. An example of its application is outlined in Example 2.12, and its formula is expressed as follows:

$$output = \frac{1}{1 + e^{-input_0}} \tag{2.10}$$

Example 2.12: Sigmoid Function

```
qplasmid([ name := "qp_A" ]);

function([ name := "fun_sigA"
  , type := "sigmoid"
  , inputs := {"qp_A"}
]); // 1 / (1 + e^"qp_A")
```

Hill function: `"hill"` The well-established Hill function, commonly employed to model gene expression with cooperative binding, serves as a generalization of the Michaelis-Menten and Monod equations. These specific cases are easily derived by maintaining the exponent parameter (n) as 1. Repression dynamics are achieved by utilizing a negative value for n. This function accepts only one input (the concentration) and entails three parameters: the maximum rate, the half-saturation constant k, and the exponent n, in this specified order. An illustration of its application is provided in Example 2.13, and its formula is expressed as follows:

$$output = \frac{max \cdot input_0^n}{k^n + input_0^n} \tag{2.11}$$

Example 2.13: Hill Function

```
hill_max := 10.0
hill_k := 0.5
hill_n := 2.0

qplasmid([ name := "qp_A" ]);

function([ name := "fun_hillA"
  , type := "hill"
  , inputs := {"qp_A"}
  , params := {hill_max, hill_k, hill_n}
]); // (10.0 * "qp_A"^2) / ( 0.5^2 + "qp_A"^2 )
```

Combinatorial summation: `"combi_sum"` This function executes a polynomial operation, specifically a summation of products. The terms encompass all conceivable combinations ranging from order 1 to n of the inputs, where n denotes the number of

inputs. Within each term, the inputs and their respective parameters are subject to multiplication. A total of $n+1$ parameters are involved: a weight assigned to each input plus a bias term, which is added to the final result. The default values for the weights are 1.0, and the bias is set at 0.0. The overarching formula is elucidated as follows:

$$output = \sum_{i=0}^{n} \prod (input_{i-i+o} \cdot param_{i-i+o}) + param_{n+1} \; for \; o = 1 \; to \; n \tag{2.12}$$

For two inputs, the formula is articulated as follows:

$$output = input_0 \cdot param_0 + input_1 \cdot param_1 + input_0 \cdot param_0 \cdot input_1 \cdot param_1 + param_2 \tag{2.13}$$

This `Function` can be employed as the common denominator for kinetic formulas of regulated gene expression involving multiple transcription factors, with the weight parameters serving as the inverse of the respective kinetic constants, as demonstrated in Example 2.14.

Example 2.14: Combinatorial summation Function

```
ka := 3.0;
kb := 0.5;

qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_combiAB"
  , type := "combi_sum"
  , inputs := {"qp_A", "qp_B"}
  , params := {1.0/ka,1.0/kb}
]); //1 + 1/3 * "qp_A" + 2 * "qp_B" + 2/3 * "qp_A" * "qp_B"

function([ name := "fun_A"
  , type := "sum"
  , inputs := {"qp_A"}
  , params := {1.0/ka}
]); //1/3 * "qp_A"

function([ name := "fun_stateA"
  , type := "product"
```

```
  , inputs := {"fun_A", "fun_combiAB"}
  , params := {1.0, -1.0}
]); //("qp_A" / 3) / "fun_combiAB"
```

Saturating exponential: `"sat_exp"` This function exhibits saturation towards a maximum value, following the pattern of an ascending exponential (akin to cumulative exponential probability). It amalgamates a summation and an exponential operation, with the saturating effect applied to the outcome of the summation. This function finds utility in modelling dosage effects or the impact of neighbouring cells in conjugation rates (see Subsection 3.5.1). The number of inputs is unbounded. A summation `Function` is applied to these inputs (refer to the summation `Function` above), and the resultant output is subsequently processed by the saturating exponential.

In cases where there are n inputs, the number of parameters is $n + 2$, with the first parameter representing the maximum (saturation) value. The subsequent parameters correspond to the weights and bias of the summation. The weight associated with input i is thus denoted by parameter $i + 1$. The default value for all parameters is set at 1.0, with the exception of the last one (bias), which defaults to 0.0. This `Function` is depicted in Figure 5.2, an exemplification of its application is provided in Example 2.15, and its formula is articulated as follows:

$$output = param_0 \cdot (1 - e^{-(\sum input_i * param_{i+1} + param_{n+2})}) \tag{2.14}$$

Example 2.15: Saturating exponential Function

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_satexpAB"
  , type := "sat_exp"
  , inputs := {"qp_A", "qp_B"}
  , params := {3.0,0.5,0.6}
]); //3 * (1 - exp( -( 0.5 * "qp_A" + 0.6 * "qp_B") ) )
```

Product of exponentials: `"exp_product"` Similar to the saturating exponential, this function integrates an additional diminishing exponential factor. It operates as a multiplication of two saturating exponential functions, akin to the one described above, one ascending and one descending. The result is a function that undergoes growth until

attaining a maximum value and subsequently experiences a gradual decline, asymptotically approaching 0. This behaviour is depicted in Figure 5.2. This function finds application in modelling diverse phenomena, including copy number control mechanisms for plasmids (refer to Subsection 3.3.2).

The number of inputs is unbounded (n), and the number of parameters is $n + 3$. The first parameter represents the maximum value, while the last two denote the biases of each of the exponential factors. The remaining parameters pertain to the weights of the summations performed at each factor. The sign of these weights determines the destination of the input: positive values contribute to the growing exponential, whereas negative values contribute to the diminishing one (with their sign reversed). An exemplification of its application is furnished in Example 2.16, and its formula is expressed as follows:

$$\begin{aligned} exp1 &= 1 - e^{-(\sum_{i=0}^{n} input_i * param_{i+1} + param_{n+2})} \; if \; param_i > 0 \\ exp2 &= e^{-(\sum_{i=0}^{n} -input_i * param_{i+1} + param_{n+3})} \; if \; param_i < 0 \\ output &= param_0 \cdot exp_1 \cdot exp_2 \end{aligned} \tag{2.15}$$

Example 2.16: Exponential product Function

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_expprodAB"
  , type := "exp_product"
  , inputs := {"qp_A", "qp_B"}
  , params := {3.0,0.5,-0.6, 1.0, 0.5}
]); // 3 * (1 - exp( -(0.5 * "qp_A" + 1) ) * exp( -(0.6 * "
   qp_B" + 0.5) ) )

function([ name := "fun_expprodA"
  , type := "exp_product"
  , inputs := {"qp_A", "qp_A"}
  , params := {3.0,1.0,-0.1}
]); // 3 * (1 - exp( -1 * "qp_A" ) * exp( -0.1 * "qp_A" ) )
```

Function composition `Functions` can be chained to form composite `Functions`, such as a polynomial of any order. The code in Example 2.17 exemplifies how two

Table 2.4: Types of Function element

KEY	MATH	INPUT NUM	INPUTS	PARAM NUM	PARAMS	DEFAULT
"const"	a	1	a	0	{}	{}
"min"	$min(x\_0 - x\_n, b)$	n	{x_0–x_n}	1 (optional)	{b}	{}
"max"	$max(x\_0 - x\_n, b)$	n	{x_0–x_n}	1 (optional)	{b}	{}
"abs"	$abs(a)$	1	a	0	{}	{}
"sum"	$\sum_{i=0}^{n}(w_i \cdot x_i) + b$	n	{x0-xn}	n+1 (optional)	{w0-wn, b}	{1.0, 1.0, ..., 0.0}
"product"	$\prod_{i=0}^{n}(x_i^{w_i}) \cdot b$	n	{x0-xn}	n+1 (optional)	{w0-wn, b}	{1.0, 1.0, ..., 1.0}
"exp"	b^a	1 or 2	{a} or {a, b}	1 (optional)	{b} used only if the input size is 1	{e} = e as the base
"log"	$log_b a$	1 or 2	{a} or {a, b}	1 (optional)	{b} used only if the input size is 1	{e} = natural logarithm with e as the base
"sigmoid"	$\frac{1}{1+e^{-a}}$	1	{a}	0, all are standard	{}	—
"hill"	$\frac{max \cdot x^n}{k^n + x^n}$	1	{x}	3 (optional)	{max, k, n}	{1.0, 1.0, 1.0}
"combi_sum"	$\sum_{i=0}^{n}\prod(x_{i-i+o} \cdot w_{i-i+o}) + b$ $o = 1\ to\ n$	n	{x_0–x_n}	n+1	{$w\_0 - w\_n, b$}	{1.0, 1.0, ..., 1.0}
"sat_exp"	$max \cdot (1 - e^{-(\sum w_i \cdot x_{i+1} + b)})$	n	{x_0–x_n}	n+2	{max, w_0–w_n, b}	{1.0, 1.0, ..., 0.0}
"exp_product"	$sum1 = \sum x_i * w_{i+1} + b1$ $if\ w_{i+1} > 0$ $sum2 = \sum -x_i * w_{i+1} + b2$ $if\ w_{i+1} < 0$ $max \cdot (1 - e^{-sum1}) \cdot e^{-sum2}$	n	{x_0–x_n}	n+3	{max, w_0–$w\_n, b1, b2$}	{1.0, 1.0, ..., 0.0, 0.0}

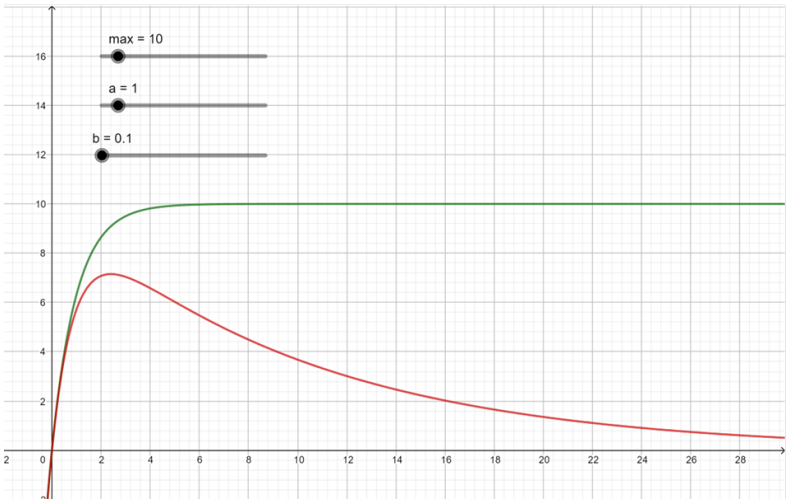

Figure 2.3: **Plots of saturating exponential and exponential product Functions**. The examples employ a single input, x, where max represents the maximum value, and a and b denote the weights of that input. The saturating exponential (depicted in green) is defined as $max \cdot (1 - exp(-a \cdot x))$, while the exponential product (depicted in red) is expressed as $max \cdot (1 - exp(-a \cdot x)) \cdot exp(-b \cdot x)$.

`"product"` `Functions` are incorporated into a `"sum"` `Function`, culminating in the construction of a polynomial of order 2 with two terms.

Example 2.17: Composition of a polynomial function

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_prodA2"
  , type := "product"
  , inputs := {"qp_A"}
  , params := {2.0}
]); //"qp_A"^2

function([ name := "fun_prodAB"
  , type := "product"
  , inputs := {"qp_A", "qp_B"}
  , params := {1.0, 1.0}
]); //"qp_A" * "qp_B"

function([ name := "fun_poly"
```

```
  , type := "sum"
  , inputs := {"fun_prodA2", "fun_prodAB"}
  , params := {1.0, 1.0, 3.0}
]); //"qp_A"^2 + "qp_A" * "qp_B" + 3.0
```

Bounds of the value and precision By default, the function's output is unrestricted. However, two parameters, `min` and `max`, can be employed to establish bounds, capping any values outside of this range. The keyword `min` corresponds to the minimum value or lower bound, while `max` designates the maximum value or upper bound, both included. Additionally, the `precision` parameter enables the rounding of the output to a specified number of decimal places. In certain exceptional cases, this may prove beneficial in mitigating errors introduced by floating-point representations. The default behaviour does not include any rounding.

Automatic concentration The quantitative state of elements is stored in cells as absolute amounts. When concentration is required for a calculation, the element must be divided by the current cell volume. While this can be accomplished using a `"product"` type `Function`, a more convenient approach is available through the `auto_vol` field of the `Function` element.

By default, no automatic scaling by the volume is applied, and the raw amount of the `input` elements is utilized in the calculation. However, when set to `"division"`, each `input` is automatically divided by the cellular volume, and the resultant concentrations are employed instead. It is crucial to emphasize that this feature should only be employed when the division is meaningful for all inputs. Specifically, it should not be used when there are any Boolean inputs or inputs already represented as concentrations.

In Example 2.18, a `Function` is created for calculating the sum of the concentrations of two plasmids in two alternative ways. `"fun_sumAB1"` necessitates additional `Functions` to compute the concentrations of the plasmids, whereas `"fun_sumABsame"` accomplishes this automatically.

Example 2.18: Automatic concentration calculation

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_concA"
  , type := "product"
  , inputs := {"qp_A", "_qm_vol" }
  , params := {1.0, -1.0}
```

```
]);

function([ name := "fun_concB"
  , type := "product"
  , inputs := {"qp_B", "_qm_vol" }
  , params := {1.0, -1.0}
]);

function([ name := "fun_sumAB1"
  , type := "sum"
  , inputs := {"qp_A", "qp_B"}
]);

function([ name := "fun_sumABsame"
  , type := "sum"
  , inputs := {"qp_A", "qp_B"}
  , auto_vol := "division"
]);
```

If `auto_vol` is set to `"product"`, all inputs are multiplied by the volume instead. This functionality can be employed to perform the reverse calculation, converting concentrations into absolute amounts.

Stochastic Functions `Function` elements can be either deterministic or stochastic. The latter employs two fields: `rnd` and `rnd_params`. Modifying either of these fields is sufficient to render a `Function` stochastic, with the other field assigned a default value. If a `Randomness` is assigned to `rnd` without specifying `rnd_params`, as exemplified in `"fun_sumABrnd1"` from Example 2.19, the raw values generated by the `Randomness` are utilised without scaling. Since `"rnd_A"` is a default standard normal distribution, the calculation of `"fun_sumABrnd1"` will incorporate added normal noise with a mean of 0.0 and a standard deviation of 1.0.

Example 2.19: Stochastic Function

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

randomness([ name := "rnd_A" ]);

function([ name := "fun_sumABrnd1"
```

```
  , type := "sum"
  , inputs := {"qp_A", "qp_B"}
  , rnd := "rnd_A"
]);

function([ name := "fun_sumABrnd2"
  , type := "sum"
  , inputs := {"qp_A", "qp_B"}
  , rnd_params := {0.0, 0.1}
]);
```

Conversely, `"fun_sumABrnd2"` is furnished with `rnd_params` but not `rnd`. In this scenario, a default normal `Randomness` (identical to `"rnd_A"`) is automatically generated, and the sampled values are scaled using the provided parameters. Consequently, normal noise with a mean of 0.0 and a standard deviation of 0.1 is introduced into the calculation of `"fun_sumABrnd2"`.

`Functions` can be employed to generate linear combinations of `Randomness` elements, offering an alternative to using individual `Randomness` elements. This approach is useful for producing partial correlations or incorporating different noise components. However, it is important to note that automatic scaling is not performed in such cases, as the appropriate scaling method cannot be deduced automatically. The automatic scaling deduction is only guaranteed when all inputs are normal `Randomness` elements, ensuring that the output is also a normal distribution.

2.2.2 Ordinary differential equation (Ode)

Keywords: `ode`
Recommended prefixes: `ode_`
Linked to (direct): any cellular element, logic or biological, specially `Function` and including other `Odes`
Linked to (reverse): Most cellular logic (`Function`, `Ode`, `Historic`, `QGate`, `Cell Colour`); and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `Ode` element represents an ODE, and it is possible to concatenate multiple instances of it to form a coupled ODE system. The optional stochastic properties enable the calculation of SDEs. When combined with the `Historic` element (refer to Subsection 2.2.3), it facilitates the modelling of DDEs. The complete list of fields for the `Ode` is provided in Figure 2.4 and Table 2.6.

Table 2.5: Parameters of the Function element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
input* / inputs*	array of cellular elements	ref.	List of input elements. Any logic (including other Functions) or biological intracellular element is valid. Those Quantitative in nature are best suited but Boolean ones may be assigned too (0 = 0.0, 1 = 1.0).	_
type	string in {"sum", "product", "exp", "log", "sigmoid", "hill"}	—	Family of mathematical functions: weighted summation, weighted product, exponentials, logarithmic, sigmoid and Hill function (Michaelis-Menten and Monod are concrete cases of it)	"sum"
params	array of real	_	Constant parameters whose meaning depends on the type of function	{} = Default params that depend on the function type
min	real	—	Minimum value. Values higher than this are capped.	-infinite
max	real	—	Maximim value. Values lower than this are capped.	infinite
rnd / randomness	Randomness	ref.	Randomness ruling the stochastic behaviour in case it exists.	_ = deterministic Function
rnd_params	array of real	_	Parameters for scaling the Randomness in case it exists and is not already scaled	{} = the default parameters of the distribution type of the Randomness
auto_vol	string in "division", "product", "none"	—	Automatic scaling by the cellular volume. In "division" mode, every input is divided by the volume to convert amounts into concentrations. In "product" mode, every input is multiplied by the volume to convert concentrations into amounts.	false
precision	integer	decimal places	Number of decimal places to round the output	-1 = no rounding (may introduce float point rounding errors)

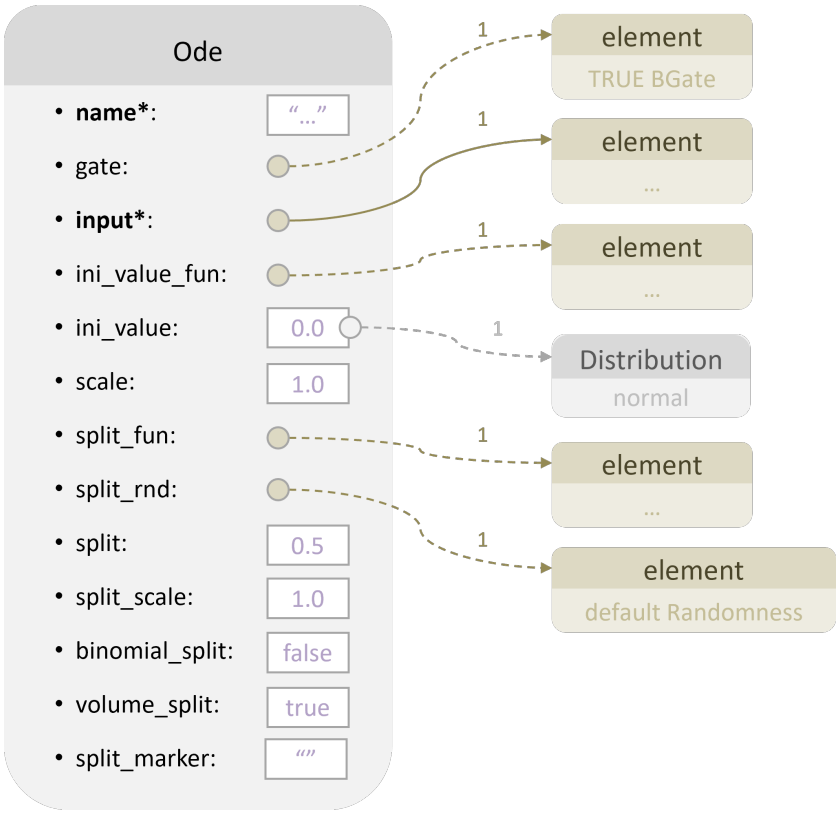

Figure 2.4: **Ordinary differential equation (Ode) element**

Behaviour While the behaviour of cells in gro 3.0 is predominantly driven by the composition of present biological elements, the utilization of differential equations is also feasible. An `Ode` element monitors the magnitude that undergoes modification at each time step, adhering to a synchronous approach.

Input The `input` field is employed to establish the rate, i.e., the quantity that will be added to the tracked amount during each time step. While a `Function` is the typical input for an `Ode`, any element can be assigned, including another `Ode` or even itself. Loops are permissible, and in such instances, the previous values of the `Ode` are employed. The state of the specified input element is scaled by the time step size and then added to the current value.

Example 2.20 illustrates how to employ `Ode` elements to simulate unregulated transcription in a quantitative manner. An `Ode` element is designated for each of the two tracked quantitative values (RNA and protein). The system's behaviour is governed by the following differential equations. For simplicity, the RNA production rate has

been made proportional to the copy number of the `Operon` element. Users have the flexibility to use more sophisticated custom `Functions` for such calculations. It is noteworthy that the copy number used is the absolute amount and not the concentration. Introducing an additional `Function` would be necessary to compute the concentration, as the `auto_conc` option should not be employed to avoid affecting the other input, `"ode_rna"`, which is already a concentration. The resulting ODE system is as follows:

$$\begin{aligned} drna/dt &= 0.2 \cdot \text{"}op_A\text{"} - 0.01rna \\ dprot/dt &= 0.01 \cdot rna - 0.001prot \end{aligned} \tag{2.16}$$

Example 2.20: Unregulated gene expression with Odes

```
operon([ name := "op_A" ]);

function([ name := "fun_rna"
  , inputs := { "op_A", "ode_rna" }
  , params := {0.2, -0.01 }
]); //drna/dt = 0.2 * "op_A" - 0.01 * rna

ode([ name := "ode_rna", input := "fun_rna" ]);

function([ name := "fun_prot"
  , inputs := { "ode_rna", "ode_prot" }
  , params := { 0.01, -0.001}
]); //dprot/dt = 0.01 * rna - 0.001 * prot

ode([ name := "ode_prot", input := "fun_prot" ]);
```

Example 2.21 outlines how to delineate regulated gene expression with one activator transcription factor using `Odes`. To simplify the model, the RNA stage has been omitted, and the rates have been made independent of the copy number. The system calculates the following differential equations:

$$\begin{aligned} dTF/dt &= 0.01 - 0.001 \cdot TF \\ dprot/dt &= Hill(max = 0.03, k = 5.4, n = 1) - 0.001 \cdot prot \end{aligned} \tag{2.17}$$

Example 2.21: Regulated gene expression with Odes

```
function([ name := "fun_tf"
```

```
  , inputs := {"ode_tf"}
  , params := { -0.001, 0.01 }
]); //dtf/dt = 0.01 - 0.01 * tf

ode([ name := "ode_tf", input := "fun_tf" ]);

function([ name := "fun_protHill"
  , type := "hill"
  , inputs := {"ode_tf"}
  , params := { 0.03, 5.4 }
]);

function([ name := "fun_prot"
  , inputs := { "fun_protHill", "ode_tf" }
  , params := { 1.0, -0.001 }
  , rnd_params := { 0.0, 0.005 }
]); //dprot/dt = Hill(max=0.03, k=5.4, n=1) - 0.001 * prot

ode([ name := "ode_prot", input := "fun_prot" ]);
```

Moreover, random noise, adhering to a normal distribution with a mean of 0.0 and a standard deviation of 0.005, is introduced to the rate. Notably, this noise is not sampled at each time step; instead, the same value is reused until the associated `Randomness` (see Subsection 2.1.2) element is resampled.

Conditional Odes When the `gate` field is assigned a `Gate` (or another element), the `Ode` is only updated when the condition/presence specified in that gate evaluates to true; otherwise, the value remains unaltered.

Scale The `scale` parameter serves to automatically multiply the input at the `input` field before addition. Its default value is 1.0, implying that the input remains unaltered. In most scenarios, this parameter is not essential as scaling can be accomplished within a `Function` designated as `input`. However, it may prove convenient when directly utilizing other types of elements as input. The `Odes` are assumed to be specified in minutes, so that the change during each time step corresponds to the input's value multiplied by the time step size.

Initial value The `ini_value` parameter is utilised to establish the initial value of the `Ode`, typically defaulting to 0.0. It can be assigned either a deterministic value

or a random distribution from which to sample the initial value. When an array of real numbers is provided, it is interpreted as the mean and standard deviation of a normal distribution. If a different distribution family is desired, it can be specified using the `ini_value_dist` keyword, either as a reference or an inlined dictionary (see Subsection 2.1.1).

Alternatively, a custom `Function` can be designated as the initial value via the `ini_value_fun` field, allowing any cellular element. This value takes precedence over `ini_value` or `ini_value_dist`, rendering them irrelevant. To make the initial value both dependent on other elements and stochastic, a stochastic `Function` (refer to Figure 2.2) must be assigned to this field.

Split on cell division The `Odes` are assumed to be expressed in concentration units as defined by the user, rather than in amount. Accordingly, their value is initially converted into an amount by multiplying it by the cellular volume. Subsequently, this amount is (randomly) distributed between the daughter cells and then converted back to a concentration by dividing it by the cellular volume. The distribution of the amount adheres to a user-defined value or distribution. By default, this is a fraction of 0.5, signifying a perfectly deterministic and equal split of the amount. Consequently, in this scenario, the daughter cells possess precisely the same value for the `Ode` (concentration) as the mother cell.

Users can introduce asymmetry to the partition by assigning a different value to the `split` parameter. Alternatively, if an array is provided, these values are interpreted as parameters of a random distribution from which to sample the fraction. The default distribution is normal, but an alternative distribution can be employed by linking a custom `Randomness` (or another element) to the `split_rnd` field.

An alternative approach involves utilizing a binomial distribution to explicitly distribute each molecule. This method is particularly well-suited for molecules present in low amounts, as commonly encountered in models of plasmid loss. To opt for this mode, set `binomial_split` to true, and the sampled split fraction will be interpreted as the p parameter of the binomial distribution.

In both scenarios, the split is executed in terms of amount. The conversion from concentration is performed by multiplying by the volume in cubic micrometers, assuming that the concentration units of the `Ode` are $molecules/\mu m^3$. If an alternative unit, such as molarity, is employed, the conversion factor from that unit to $molecules/\mu m^3$ must be supplied via `split_scale`. The `volume_split` option is set to true by default, ensuring that the difference in cell volume among the daughter cells is factored into the calculation.

In Example 2.22, a `Function` involving the concentration of `"op_A"` is employed as the initial value for the `Ode` representing the concentration of RNA. Uniform noise in

the range [-0.1, 1.0] is introduced to this value. The split fraction on cell division is sampled from a normal distribution with a mean of 0.5 and a standard deviation of 0.1.

Example 2.22: Initial value and partition in Odes

```
operon([ name := "op_A" ]);

function([ name := "fun_rna"
  , inputs := { "op_A", "ode_rna" }
  , params := {0.2, -0.01 }
]); //drna/dt = 0.2 * "op_A" - 0.01 * rna

function([ name := "fun_ini"
  , inputs := { "op_A" }
  , params := { 2.4 }
  , auto_vol := "division"
]);

ode([ name := "ode_rna"
  , input := "fun_rna"
  , gate := "_ga_true"
  , ini_value_fun := "fun_ini"
  , ini_value_dist := [ type := "uniform", params := {-0.1,
    0.1} ]
  , split_params = {0.5, 0.1}
]);
```

2.2.3 Historic and delayed differential equations

Keywords: `historic`
Recommended prefixes: `h_`
Linked to (direct): any cellular element, logic or biological, specially `Functions`
Linked to (reverse): Most cellular elements, logic (`Function`, `Ode`, `QGate`, `Cell Colour`) and biological (`Molecule`, `Regulation`, `Flux`, `OriV`, `Copy Control`, `Partition`, `OriT`, `Pilus`, `Plasmid`), and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description This element is employed to store past states of any other element. Its primary application is in computing DDEs, using it in combination with `Ode` (see

Table 2.6: Parameters of the Ode element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
input* / delta_input	any cellular element	ref.	Any logic (including other Odes) or biological element is valid. Those Quantitative in nature are best suited but Boolean ones may be assigned too (0 = 0.0, 1 = 1.0).	—
gate	any cellular element, usually Gate	ref.	Condition that must evaluate to true/present for the Ode to be updated	"_ga_true"
ini_value_fun	any cellular element, usually Function	ref.	Custom Function or any other element to use its state for the initial value. Shadows "ini_value".	—
ini_value / ini_value_params / ini_value_dist	real, array of 2 real, Distribution or dictionary	—	Initial value. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution. If a custom function is set too, they are added (this one acts as noise).	0.0 deterministic
scale	real	_	Multiplies the value of "input" for fast scaling without the need of a Function	1.0
split_fun	any cellular element, usually Function	ref.	Custom function used to distribute the value on cell division in the case of continuous partition. Shadows "split". Interperted as the p param if "binomial_split".	—
split_rnd	any cellular element, usually Randomness	ref.	Provider of randomness for the split fraction	default normal Randomness
split / split_params	real or array of 2 real in [0.0, 1.0]	fraction or prob.	Fraction used to distribute the value on cell division in the case of continuous partition. Either a single deterministic one or params for a distribution defined by "split_rnd". Interperted as the p param if "binomial_split".	0.5 deterministic
custom_split	Boolean	_	Whether the partition of the value on cell division is made by using a binomial distribution (false) or a custom continuous distribution (true)	true
split_scale	real	_	Conversion factor from the used custom concentration unit to molecules/um	1.0
binomial_split	Boolean	_	Whether the partition of the value on cell division is made by using a binomial distribution (true) or a custom continuous distribution (false)	false
volume_split	Boolean	_	Whether to consider the value as a concentration and take into account the relative volumes of daughter cellson split	true
split_marker	string	_	User-given name to access the value of a stochastic "split" param	—

Subsection 2.2.2). The complete list of fields for the `Historic` is provided in Figure 2.5 and Table 2.7.

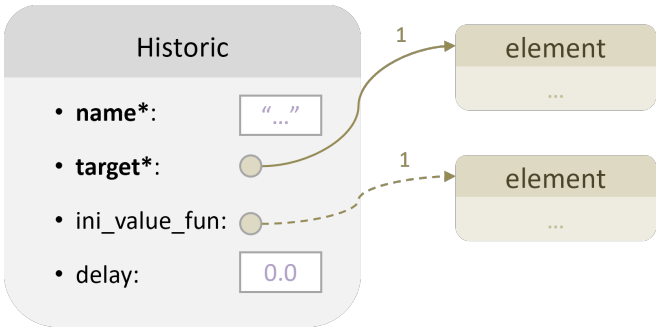

Figure 2.5: **Historic element**

Target The target element is assigned at the `target` field. This element is the one whose historic states are intended to be stored. Any cellular element is valid, encompassing both logic and biological elements, whether digital or quantitative. Using other `Historic` elements as target is not forbidden but discouraged.

Delay The delay represents the difference between the current time and the time in the past when the state of the target element was stored. Essentially, it indicates how long ago the state was recorded, measured in minutes. It is not necessary for the delay to be a multiple of the step size, as `Historic` elements can be updated asynchronously at any time. However, it is important to note that if element types that are updated synchronously (such as `Ode` or `CellColour`) are used as targets, they will exhibit values corresponding to the last synchronous update at the fixed time step size. The delay is explicitly set and specified by the `delay` parameter, measured in minutes, with a default value of 0.1, aligning with the default time step size.

Example 2.23 illustrates how to store past values of `Functions` used as derivatives for `Odes`. The derivative for `"ode_rna"` is a weighted sum of the current value of `"fun_rna0"` and two past states of it, one from 0.1 minutes ago and the other from 0.2 minutes ago. These past values are stored using `Historic` elements.

Example 2.23: Delayed Ode using Historic derivatives

```
operon([ name := "op_A" ]);
```

```
function([ name := "fun_opAconc"
  , type := "const"
  , inputs := {"op_A"}
  , auto_vol := "division"
]);

function([ name := "fun_rna0"
  , inputs := {"fun_opAconc", "ode_rna"}
  , params := {0.2, -0.01 }
]); //drna/dt = 0.2 * "op_A" - 0.01 * rna

historic([ name := "h_rna1"
  , target := "fun_rna0"
  , delay := 0.1
]); //"fun_rna0" in time t-0.1

historic([ name := "h_rna2"
  , target := "fun_rna0"
  , delay := 0.2
]); //"fun_rna0" in time t-0.2

function([ name := "fun_rna"
  , inputs := {"fun_rna0", "h_rna1", "h_rna2"}
  , params := {0.5, 0.25, 0.25}
]); //drna/dt = 0.2 * "op_A" - 0.01 * rna

ode([ name := "ode_rna", input := "fun_rna" ]);
```

Example 2.24 is similar to the previous example. The distinction lies in the utilization of previous states of the `Ode` (integral) itself in the derivative `Function`.

Example 2.24: Delayed Ode using Historic states

```
operon([ name := "op_A" ]);

function([ name := "fun_opAconc"
  , type := "const"
  , inputs := {"op_A"}
  , auto_vol := "division"
```

```
]);

function([ name := "fun_rna0"
  , inputs := {"fun_opAconc", "ode_rna"}
  , params := {0.2, -0.01 }
]); //drna/dt = 0.2 * "op_A" - 0.01 * rna

historic([ name := "h_rna1"
  , target := "ode_rna"
  , delay := 0.1
]); //"ode_rna" in time t-0.1

historic([ name := "h_rna2"
  , target := "ode_rna"
  , delay := 0.2
]); //"ode_rna" in time t-0.2

function([ name := "fun_rna"
  , inputs := {"fun_rna0", "h_rna1", "h_rna2"}
  , params := {0.9, 0.05, 0.05}
]); //drna/dt = 0.2 * "op_A" - 0.01 * rna

ode([ name := "ode_rna", input := "fun_rna" ]);
```

Initial value Newly created cells necessitate an initial state for the `Historic`. The default behaviour is to set the initial value as the current (initial) state of the target element. Alternatively, a custom initial value can be assigned to the `ini_value_fun` field, accepting any quantitative cellular element. The initial state of this element will then also be the initial state of the `Historic`. Direct use of a numerical value is not permitted, but the same outcome can be achieved by employing a constant `Function` (see Figure 2.2) as the linked element. Leveraging a `Randomness` (see Figure 2.1) or a stochastic `Function` will yield initial values sampled from a probability distribution.

Example 2.25 elucidates how to employ custom `Functions` for the delay and the initial value. The delay will vary in size, contingent on the value of a dynamically changing `Function`. The initial value is a stochastic `Function`, introducing an element of randomness to it.

Example 2.25: Custom delay and initial value in Historic

```
operon([ name := "op_A" ]);

function([ name := "fun_opAconc"
  , type := "const"
  , inputs := {"op_A"}
  , auto_vol := "division"
]);

function([ name := "fun_rna0"
  , inputs := {"fun_opAconc", "ode_rna"}
  , params := {0.2, -0.01 }
]); //drna/dt = 0.2 * "op_A" - 0.01 * rna

function([ name := "fun_delay"
  , inputs := {"fun_rna0"}
  , params := {0.1}
]); //drna/dt = 0.1 * "fun_rna0"

function([ name := "fun_ini"
  , type := "const"
  , inputs := {"fun_opAconc"}
  , rnd_params := {0.0, 1.0}
]);

historic([ name := "h_rna1"
  , target := "fun_rna0"
  , delay := "fun_delay"
  , ini_value := "fun_ini"
]);
```

2.3 Conditions (gate elements)

This section covers the `Gate` elements, which denote conditions that evaluate to true or false. They are employed to regulate the activity of conditional cellular elements, constituting the majority of logical and biological elements. The `Quantitative Gate` or `QGate` (Subsection 2.3.1) is a condition assessed over a quantitative input, i.e., an inequation. Meanwhile, the `Boolean Gate` or `BGate` (Subsection 2.3.2) simulates a logic gate operating on digital inputs.

Table 2.7: Parameters of the Historic element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
target*	any quantitative cellular element	ref.	The element whose history is to be tracked.	_
ini_value / ini_value_elem	any quantitative cellular element	ref.	Initial value in newly created cells. Can be made stochastic by assigning an stochastic Function.	_ = the current value of the target
delay	positive real	_	Deterministic size of the delay	0.0

2.3.1 Quantitative Gate (QGate)

Keywords: `qgate` `q_gate`
Recommended prefixes: `qga_`
Linked to (direct): any cellular element with quantitative state, logic or biological
Linked to (reverse): Some logic cellular elements (`Ode`, `BGate`, `Cell Colour`) and most biological ones (`Regulation`, `Flux`, `OriV`, `Copy Control`, `Partition`, `OriT`, `Pilus`, `Strain`), and global reporting elements (`Population Stat`, `Out File`, `Plot`, as well as a filter for `CellPlating`)

Description The `Quantitative Gate` or `QGate` is a quantitative condition, represented by an equation or inequation evaluated over any cellular input element. The comprehensive list of fields for the `QGate` is presented in Figure 2.6 and Table 2.8.

Behaviour The evaluation performed by the `QGate` element can be expressed as follows:

$$< input >< operator >< value >? \tag{2.18}$$

The three involved fields are user-defined. `<input>` is the state of the cellular element under evaluation. `<value>` is a numeric parameter used for comparison. `<operator>` is a comparison operator, and its selection determines whether the comparison is an equation or an inequation.

The output of the `QGate` is Boolean: true if the state of the input element satisfies the condition and false otherwise. While technically feasible to chain `QGates` provided

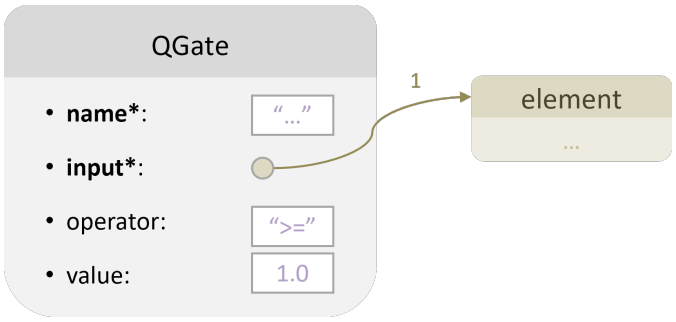

Figure 2.6: **Quantitative Gate (QGate) element**

that no loops are created, it is not a typical use of the element. Loops encompassing only `Gate` elements, whether all `QGates`, all `BGates`, or a combination, are not permissible.

Input The input of a `QGate` is designated through the `input` field, which is compulsory. It can be any cellular element, but elements of quantitative nature are particularly suitable. While the `QGate` element accepts a single input, an arbitrary function of multiple elements can be provided by assigning a `Function` element (Subsection 2.2.1) as input. This allows for the comparison of values from multiple cellular elements, as demonstrated in Example 2.28.

In Example 2.26, the `QGate` evaluates to true when the number of copies of `"qp_A"` plasmid is greater than or equal to 1.0.

Example 2.26: Simplest QGate

```
qplasmid[ name := "qp_A" ]);

qgate([ name := "qga_A"
  , input := "qp_A"
]);
```

Comparison operator and value The state of the given element (on the left) is compared to a user-defined value (on the right) using a user-defined comparison operator. The operator is assigned using the `operator` field, with "greater than or equal" being the default.

The default value is 1.0 and can be adjusted with the `value` parameter. Equations (equal and not equal operators) should only be used with inputs of integer (or Boolean) nature, such as `Operon` or `BPlasmid`. Utilizing them with real-valued inputs, like a

`Function`, is risky due to the representation error of real numbers. A pair of inequations is better suited when such error is expected.

In Example 2.27, the `QGate` evaluates to true when the number of copies of `"qp_A"` plasmid is exactly 3.0. Note that the `value` parameter expects a real value in all cases.

Example 2.27: QGate with custom operator and value

```
qplasmid[ name := "qp_A" ]);

qgate([ name := "qga_A"
  , input := "qp_A"
  , operator := "="
  , value := 3.0
]);
```

The `QGate` in Example 2.28 evaluates to true when the number of copies of `"qp_A"` plasmid is greater than or equal to that of `"qp_B"` (i.e., `"qp_A"` minus `"qp_B"` is greater than or equal to 0.0).

Example 2.28: Function and QGate chain

```
qplasmid[ name := "qp_A" ]);
qplasmid[ name := "qp_B" ]);

function([ name := "fun_ABdiff"
  , input := {"qp_A", "qp_B"}
  , params := {1.0, -1.0}
]);

qgate([ name := "qga_moreA"
  , input := "fun_ABdiff"
  , value := 0.0
]);
```

Table 2.8: Parameters of the QGate element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
input* / elem*	any cellular element	ref.	The cellular element to compute with as the left side of the (in)equation	_
operator / type	string in {">", ">=", "<", "<=", "=", "!="}	_	Comparison operator used to compare both sides of the (in)equation. "=" and "!=" should not be used with real inputs, only with those of integer nature, like the amount of DNA elements in amount and not concentration.	">="
value	real	_	The righ hand value to compare the input with	1.0

2.3.2 Boolean Gate (BGate)

Keywords: `bgate b_gate gate`
Recommended prefixes: `bga_ ga_`
Linked to (direct): any cellular element, logic or biological, specially those with digital states, including other `Gates`
Linked to (reverse): All the conditional cellular elements, some logic ones (`Ode`, `BGate`, `Cell Colour`) and most biological ones (`Regulation`, `Flux`, `OriV`, `Copy Control`, `Partition`, `OriT`, `Pilus`, `Strain`), and global reporting elements (`Population Stat`, `Out File`, `Plot`, as well as a filter for `CellPlating`)

Description The `Boolean Gate` or `BGate` is a condition in the shape of a logic gate over inputs with digital values producing digital output (true or false). The comprehensive list of fields for the `BGate` is presented in Figure 2.7 and Table 4.2.

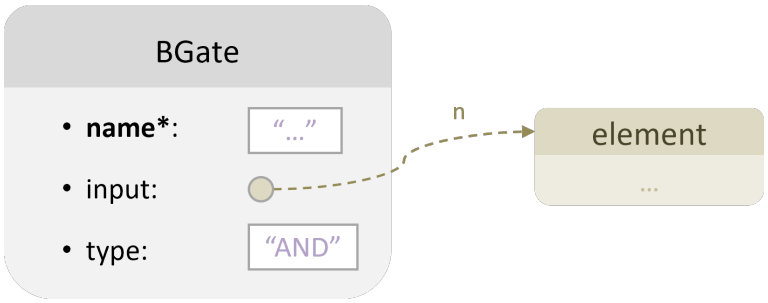

Figure 2.7: **Boolean Gate (BGate) element**

Relation with gro 2' elements The `BGate` is used to check conditions on the presence/absence of `Molecules` and other elements inside a concrete cell. Matches the `Promoter` from gro 2, except for that the input can be not only `Molecules`, but most of the biological elements plus other `Gates`. It also serves as a generalization of the protein-based condition of the gro 2 actions.

Input The `input` constitutes an array comprising cellular elements, encompassing both logical and biological entities. By default, this field remains unpopulated. While entities of a Boolean nature (e.g., `Gate`, `Molecule`, `BPlasmid`, `Strain`, `CellType`) are deemed more suitable, some quantitative elements may also be assigned and interpreted as follows: true if their value surpasses 0.0, and false otherwise. It is imperative to note that this behaviour is exclusively assured for integer-based elements denoting tangible biological components, namely: `Regulation`, `Operon`, `OriT`, `OriV`, `CopyControl`, `Partition`, `QPlasmid`. The `input` field mandates a list input, even when there is a single input.

The logic function A `BGate` encompasses solely two fields: `type` and `input`. The `type` field accommodates all the logic gates from the realm of electronics. "TRUE" (or "OPEN") and "FALSE" (or "CLOSED") necessitate no input; "YES" and "NOT" demand a singular input, while the remaining types ("AND", "OR", "XOR", "NAND", "NOR", and "XNOR") can accommodate an unbounded number of inputs. It is essential to highlight that function names must be written in upper-case. Although not all logic functions are necessary (as equivalent behaviour can be achieved through different gate types or combinations of them), the full assortment contributes to conceptual clarity. The default value for `type` is "AND". In instances where no inputs are provided, an AND gate emulates a TRUE gate. When a solitary input is present, it operates akin to a YES gate (or a NOT gate if the input is negated).

For instance, `"bga_andA"` and `"bga_yesA"`, as illustrated in Example 2.29, are functionally equivalent. Additionally, `"bga_orAnotB"` exemplifies how input negation can be achieved by prefixing the input name with "-". This specific configuration computes `"mol_A"` OR NOT `"mol_B"`.

Example 2.29: Simple BGates

```
molecule([ name := "mol_A" ]);
molecule([ name := "mol_B" ]);

bgate([ name := "bga_andA"
  , input := {"mol_A"}
]);
```

```
bgate([ name := "bga_yesA"
  , function := "YES"
  , input := "mol_A"
]);

bgate([ name := "bga_orAnotB"
  , function := "OR"
  , input := {"mol_A", "-mol_B"}
]);
```

Example 2.30 illustrates diverse approaches for verifying identical conditions. In the gro 2 context, where `Molecules` (`Proteins`) exclusively serve as input for `Gates`, encompassing both `Promoters` and action conditions, accomplishing seemingly straightforward tasks such as delineating distinct cell groups with different colors necessitated extensive code. The conventional method involved introducing a dummy `Protein` along with an `Operon` and a `Plasmid` (for its production) for each cell group, exemplified by `"bga_molA"`.

Example 2.30: Different element types as BGate inputs

```
molecule([ name := "mol_A", times := {0.0, 0.0} ]);
operon([ name := "op_A", output := "mol_A" ]);
bplasmid([ name := "bp_A", operons := "op_A" ]);
cell_type([ name := "cell_A", bplasmids := {"bp_A"} ]);

bgate([ name := "ga_cellA", input := {"cell_A"} ]);
bgate([ name := "ga_opA", input := {"op_A"} ]);

bgate([ name := "ga_redundant"
  , input := { "cell_A", "bp_A" }
]);
bgate([ name := "ga_false"
  , input := { "bp_A", "-op_A" }
]);
```

In gro 3.0, the `BGate` for cell type determination can be configured directly using a `CellType` element as input, as demonstrated by `"ga_cellA"` from Example 2.30. Similarly, `"ga_opA"` proves to be equivalent in this context, as all the `"cell_A"` cells possess a `BPlasmid` housing the specified `Operon`. Consequently, the subsequent gates

are redundant and yield false outcomes, respectively. It is crucial to note that this equivalence is contingent upon the instantaneous and constitutive production of the `Molecule`, coupled with the stipulation that the `BPlasmid` is neither lost nor modified throughout the simulation.

Predefined BGates The `"TRUE"` and `"FALSE"` `BGates` come pre-defined and can be readily employed by referencing the names outlined in Table 2.9.

Table 2.9: Predetermined individual-level BGates

NAME	FUNCTION	INPUT
"_ga_true"	"TRUE"	{}
"_ga_false"	"FALSE"	{}

Composing logic expressions Utilizing other `BGates` as inputs permits the creation of composite Boolean formulas. However, it is imperative to establish a unidirectional chaining without introducing loops, which encompasses both direct and indirect loops. The latter refers to loops exclusively composed of logic elements of any nature, leading to undefined behaviour. Example 2.31 elucidates instances of prohibited loops.

Example 2.31: Forbidden loops including BGates

```
bgate([ name := "bga_A", input := {"bga_A"} ]);

bgate([ name := "bga_B", input := {"bga_C"} ]);
bgate([ name := "bga_C", input := {"bga_B"} ]);

qgate([ name := "qga_D", input := "bga_D" ]);
bgate([ name := "bga_D", input := {"qga_D"} ]);

function([ name := "fun_E", input := "bga_E" ]);
qgate([ name := "qga_E", input := "fun_E" ]);
bgate([ name := "bga_E", input := {"qga_E"} ]);
```

`"bga_exactly2"` in Example 2.32 evaluates to true if a minimum of two out of the three `Molecules` are present. Three additional `BGates` are employed to construct this

behaviour. In the gro 2 framework, users would traditionally need to generate three intermediate `Operons` and `Proteins`, resulting in a more intricate code and marginally slower simulation.

Example 2.32: Composite BGates

```
molecule([ name := "mol_A" ]);
molecule([ name := "mol_B" ]);
molecule([ name := "mol_C" ]);

bgate([ name := "bga_andAB"
  , input := {"mol_A", "mol_B"}
]);

bgate([ name := "bga_andBC"
  , input := {"mol_B", "mol_C"}
]);

bgate([ name := "bga_andCA"
  , input := {"mol_C", "mol_A"}
]);

bgate([ name := "bga_atLeast2"
  , function := "OR"
  , input := {"bga_andAB", "bga_andBC", "bga_andCA"}
]);

function([ name := "fun_same"
  , input := {"mol_A", "mol_B", "mol_C"}
]);

qgate([ name := "qga_same", input := "fun_same", value := 1
   .9 ]);
```

Alternatively, `"qga_same"` presents another approach to achieve the same condition through the quantitative pathway. It is noteworthy that the value 1.9 is employed instead of 2.0, a precautionary measure to mitigate potential representation and rounding errors associated with real numbers.

Output usage `BGates` play a crucial role in determining the behaviour of numerous cellular elements, such as `Ode`, `CellColour`, `Regulation`, `Flux`, `Pilus`, `OriT`, `OriV`, `Copy Control`, `Partition`, and `Mutation Process`. This extensive utilization stems from the capability of most elements to exhibit conditionally regulated behaviour. Furthermore, `BGates` serve as cellular filters within various global simulation components, such as `Cell Plating` or `Out File`.

Table 2.10: Parameters of the BGate element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
input / inputs	array of Elements	ref.	List of input elements. Any logic (including other Gates) or biological intracellular element is valid. Those Boolean in nature are best suited but quantitative ones may be assigned too (0 = 0, any other number = 1). Use "-" preceding the name to negate an input.	{} = no inputs. Any gate with no inputs will behave as TRUE or FALSE depending on the function.
type	string in {"TRUE", "OPEN", "FALSE", "CLOSED", "YES", "NOT", "AND", "OR", "XOR", "NAND", "NOR", "XNOR"}	_	Boolean logic function that the gate performs over the inputs	"AND": equivalent to YES when a single input given. Equivalent to TRUE when no inputs given

2.4 Visual elements

Visual elements are employed to graphically depict the internal states of cells to users. Presently, the sole element within this category is `Cell Colour`.

2.4.1 Cell Colour

Keywords: `cell_colour cell_color`
Recommended prefixes: `ccol_`
Linked to (direct): any digital cellular element as condition and any quantitative one as a target, `Randomness` is stochastic
Linked to (reverse): global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description This element dictates the colouration of cells, shown in the the graphical user interface (GUI) and potentially stored to files. Its functionality extends to simulating fluorescent proteins with a degree of realism or simply conveying states of interest in the cells. The comprehensive list of fields for the `Cell Colour` is presented in Figure 2.8 and Table 2.12.

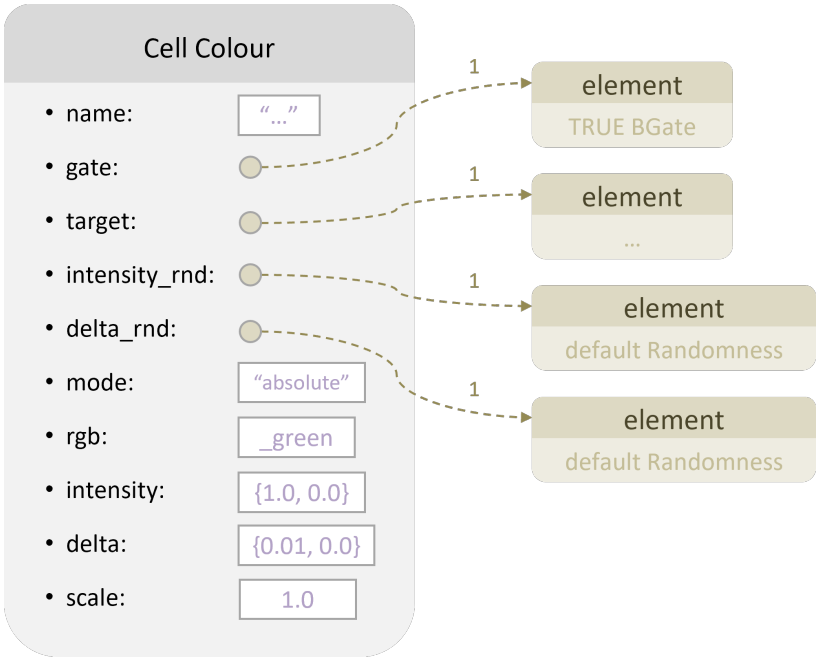

Figure 2.8: **CellColour element**

Conditional colouring Similar to numerous other cellular elements, the functionality of a `Cell Colour` element can be contingent upon the state of its `gate` field. The default element associated with this field is the predefined TRUE `BGate`, `"_ga_true"` (refer to Subsection 2.3.2), ensuring universal application of colouring to all cells in the simulation. However, specifying a custom `Gate` or any other element implies that the colour will exclusively manifest in cells where the provided `gate` element is present, is active, or evaluates to true.

In Example 2.33, the cells of the type `"cell_A"` are assigned a green colour, while those identified as `"cell_B"` are coloured in red.

Example 2.33: Simple conditional CellColours

```
cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B" ]);

cell_colour([ name := "ccol_A"
  , gate := "cell_A"
  , rgb := _green
]);

cell_colour([ name := "ccol_B"
  , gate := "cell_B"
  , rgb := _red
]);
```

Colours The colour to be applied is specified in the `"rgb"` field, requiring an array of three real numbers within the range [0.0, 1.0], representing the three reed, green, blue (RGB) channels. The default value is 0.0, 1.0, 0.0, signifying green. While users retain the flexibility to define custom colours, the gro 3.0 library files encompass descriptions of commonly used primary and secondary colours for convenience, accessible as ccl variables. The catalogue of predefined colours is detailed in Table 2.11. It is important to note that, as ccl variables predefined by gro 3.0, the colour names commence with an underscore, and no quotation marks are used.

In scenarios where more than one `Cell Colour` element is applicable (i.e., its `gate` evaluates to true) to a cell, their colours are combined using additive theory. If the condition ceases to hold, the contribution of that particular `Cell Colour` element is automatically deducted from the total colour, eliminating the need for the manual intervention that was required in gro 2.

In Example 2.34, the cells of `"cell_A"` type are assigned a green colour, while

"cell_B" cells are coloured red, same as in the preceding example. Remarkably, "cell_C", meeting the conditions of both `Cell Colours` due to the presence of the two `Plasmids`, is coloured yellow: green (0,1,0) + red (1,0,0) results in yellow (1,1,0).

Example 2.34: CellColours addition

```
bplasmid([ name := "bp_A" ]);
bplasmid([ name := "bp_B" ]);
cell_type([ name := "cell_A", bplasmids := {"bp_A"} ]);
cell_type([ name := "cell_B", bplasmids := {"bp_B"} ]);
cell_type([ name := "cell_C", bplasmids := {"bp_A", "bp_B"}
    ]);

cell_colour([ name := "ccol_green"
  , gate := "bp_A"
  , rgb := _green
]);

cell_colour([ name := "cell_B"
  , gate := "bp_B"
  , rgb := _red
]);
```

Table 2.11: Predetermined colours

NAME	RGB
_green	{ 0.0, 1.0, 0.0 }
_red	{ 1.0, 0.0, 0.0 }
_blue	{ 0.0, 0.0, 1.0 }
_yellow	{ 1.0, 1.0, 0.0 }
_magenta	{ 1.0, 0.0, 1.0 }
_cyan	{ 0.0, 1.0, 1.0 }
_white	{ 1.0, 1.0, 1.0 }
_black	{ 0.0, 0.0, 0.0 }

Intensity The `"intensity"` parameter determines the colour intensity, accepting a real number within the range [0.0, 1.0], which serves as a multiplier for the `"rgb"` field. The default intensity is 1.0, signifying no modification to the `"rgb"` field. Given that the predefined colours are saturated (utilizing the maximum value, 1.0), the default behaviour entails the use of the brightest possible colours. Users can achieve muted variations of the predefined colours by reducing the intensity. Both `Cell Colour` elements in Example 2.35 are functionally equivalent.

Example 2.35: CellColours with low intensity

```
cell_colour([ name := "ccol_A"
  , rgb := {0.0, 0.0, 0.4}
]);

cell_colour([ name := "ccol_same"
  , rgb := _blue
  , intensity := 0.4
]);
```

Stochasticity in colour intensity can be introduced by modifying at least one of two following fields: `intensity_rnd` and `intensity`. This advanced feature is employed to emulate heterogeneity in colour intensity across a population of cells expressing, for example, a fluorescent protein. It is noteworthy that the average user primarily interested in reporting internal states may not necessitate this feature.

Assigning a custom `Randomness` to the `intensity_rnd` field enables intensity sampling from the `Randomness`'s distribution in accordance with its specified sampling schedule. `intensity_params` is utilized to furnish parameters scaling the samples from the `Randomness`. This field expects a list of real numbers, with interpretation contingent upon the distribution family of the `Randomness` element. In scenarios where appropriate parameters are already designated within a custom `Randomness`, this field becomes superfluous. It finds particular utility with default or shared `Randomness` elements. If `intensity` is assigned but `intensity_rnd` is not, `intensity` is construed as the scaling parameters for a normal distribution, and a default normal `Randomness` is automatically generated. To ensure sampled values reside within the [0.0, 1.0] interval, negative values are set to 0, and values exceeding 1 are capped at 1.

The two `Cell Colour` elements in Example 2.36 are functionally identical: the intensity follows a normal distribution with a mean of 0.5 and a variance of 0.1. In one instance, a `Randomness` element with the specified parameters is assigned. In the other instance, a `Randomness` following the standard normal distribution is automatically generated, and its samples are subsequently scaled by the `Cell Colour` element.

Example 2.36: CellColour with stochastic intensity

```
randomness([ name := "rnd_A", params := {0.5, 0.1} ]);

cell_colour([ name := "ccol_A"
  , rgb := _green
  , intensity_rnd := "rnd_A"
]);

cell_colour([ name := "ccol_same"
  , rgb := _green
  , intensity_params := {0.5, 0.1}
]);
```

Modes The previously described behaviour corresponds to the default mode of `Cell Colour`: `"absolute"`, which can be set using the `mode` field. An alternative mode, `"delta"`, can be specified through the `mode` field. In `"delta"` mode, colour is incrementally added every time step, in contrast to the instantaneous colour changes observed in `"absolute"` mode. This feature can contribute to realism. The intensity parameters remain applicable in `"delta"` mode, representing the maximum intensity attainable.

A new parameter, `delta`, defines the rate at which colour is added. This is expressed on a per-minute basis; the actual amount of colour added per time step is the product of this value and the step size. When the `gate` condition is not satisfied, colour is subtracted at the same `delta` rate until it reaches 0.0. Similar to intensity, the delta can also be made stochastic. This involves two fields: `delta_rnd` and `delta_params`, behaving similarly to `intensity_rnd` and `intensity_params`, previously described. Both the delta and the intensity can be made stochastic simultaneously.

In Example 2.37, the `Cell Colour` elements incrementally modify the green colour intensity of the cells. `"ccol_dark"` does so when the `Operon` constitutively producing green fluorescent protein (GFP) is present. With the maximum intensity set to 0.4, the cells can only attain a dark green shade (0.4 out of the saturating intensity of 1.0) during the activation time of GFP, taking 10 minutes to reach the maximum value. This approach simulates the gradual appearance of GFP glow during synthesis. The `delta` is set to 0.04 colour points per minute, requiring 10 minutes (the activation time) to reach this maximum value. When `"mol_gfp"` is present, the remaining 0.6 of green colour can be added by the other `Cell Colour` in another 10 minutes. It is important to note that this method should be employed for cosmetic purposes only, as gene expression dynamics are non-linear, and the precise amount of protein is not accurately represented using this approach.

Example 2.37: CellColour in delta mode

```
molecule([ name := "mol_gfp", times := {-20.0, 10.0} ]);
operon([ name := "op_gfp", output := {"mol_gfp"} ]);

cell_colour([ name := "ccol_dark"
  , gate := "op_gfp"
  , mode := "delta"
  , rgb := _green
  , intensity := 0.4
  , delta := 0.04
]);

cell_colour([ name := "ccol_bright"
  , gate := "mol_gfp"
  , mode := "delta"
  , rgb := _green
  , intensity := 0.6
  , delta := 0.06
]);
```

Target reporting The `target` field is employed to report quantitative states, accepting any cellular element, whether logic or biological, of quantitative nature. When set, the intensity of the colour will align with the quantitative state from the specified element, superseding the `intensity` parameter. Considering that the range of values for the targeted element may differ from the [0.0, 1.0] range applicable to intensity, a `scale` parameter is introduced for scaling. The input value is automatically multiplied by `scale`, set to 1.0 by default (indicating no scaling). Should more intricate scaling be necessary, users are advised to utilize a `Function` element (see Subsection 2.2.1) for that purpose.

In Example 2.38, colour intensity is employed to report the quantitative state (copy number) of a QPlasmid. With a `scale` set to 0.01, the intensity of 1.0 (maximum) would correspond to a copy number of 100, reaching colour saturation. Additionally, normal noise with a mean of 0 and deviation of 0.01 is added.

Example 2.38: CellColour reporting QPlasmid copy number

```
qplasmid([ name := "qp_A" ]);

cell_colour([ name := "ccol_A"
```

```
  , target := "qp_A"
  , rgb := _green
  , intensity_params := {0.0, 0.01}
  , scale := 0.01
]);
```

This approach remains compatible with both colour modes, `"absolute"` and `"delta"`. In `"delta"` mode, the target element represents the maximum intensity, akin to the behaviour of the `intensity` parameter when no element is designated as a target. If the intensity is made stochastic, the provided distribution is interpreted as added noise, resulting in the summation of the state of the target and the sampled intensity.

Table 2.12: Parameters of the CellColour element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name	string	—	ID name for referencing the objet. Not neccesary as it is final.	—
gate	Gate	ref.	The condition that must hold for the cell to be colored	"_ga_true" = TRUE BGate
target / elem	Any cellular element	ref.	For reporting element states with colour. If given, the maximum colour value is set to the quantitative state of this element	""
intensity_rnd	Randomness	ref.	Randomness than rules the stochasticity in the maximum value of the colour. The maximum value acts as a multiplier for the "rgb" param.	"" = automatically created normal only if a stddev is provided by "intensity"; deterministic otherwise
delta_rnd	Randomness	ref.	Randomness than rules the stochasticity in the delta or rate of increase/decrease of the colour value.	"" = automatically created normal only if a stddev is provided by "delta"; deterministic otherwise
mode	string in {"absolute", "delta"}	—	In "absolute" mode, the colour matches the "intensity" if the "gate" evaluates to true and 0.0 otherwise. In "delta" mode, the colour increases at a rate given by the "delta" param until reaching the "intensity" when the "gate" evaluates to true and it decreases at "delta" rate until reaching 0.0 otherwise."	"absolute"
rgb / colour /color	array of 3 reals in [0.0, 1.0]	RGB in [0.0, 1.0]	The colour to apply. Either the three RGB values or the name of a predefined colour	_green = {0.0, 1.0, 0.0}
scale	real	—	A multiplier that is only used in the case of a "target" to scale its value	1.0 = not modified
intensity / intensity_params	positive real	RGB in [0.0, 1.0]	Maximum value or intensity of the colour. It acts as a multiplier of "rgb". May be a single deterministic value or a probability distribution.	{ 1.0, 0.0 } = deterministic value of 1.0
delta / delta_params	real, typically positive	"RGB in [0.0, 1.0] / min	Rate of increase/decrease of the colour value. Only used in "delta" mode. May be a single deterministic value or a probability distribution.	{ 0.01, 0.0 } = deterministic value of 0.01

Table 2.13: Summary of all the cell logic elements

NAME	KEYWORD	TYPE	DESCRIPTION	PREFIX	LINKS
Randomness	randomness	stochasticity	A random variable with inertia and inneritance properties. Represents variability among cells.	rnd_	Subsection 2.1.2, Figure 2.1, Table 2.3
Function	function	quantitative	Any mathematical function with cellular elements as input. May be stochastic.	fun_ / f_	Subsection 2.2.1, Figure 2.2, Table 2.5
Ode	ode	quantitative	Ordinary diferential equation	ode_	Subsection 2.2.2, Figure 2.4, Table 2.6
Historic	historic	quantitative	Historic record of the states of other element to use with delayed differential equations	h_	Subsection 2.2.3, Figure 2.5, Table 2.7
QGate	qgate / q_gate	condition	Comparison gate with cellular quantitative inputs and Boolean output	qga_	Subsection 2.3.1, Figure 2.6, Table 2.8
BGate	gate / bgate / b_gate	condition	Logic gate with Boolean cellular elements as input and Boolean output	ga_ / bga_	Subsection 2.3.2, Figure 2.7, Table 4.2
CellColour	cell_colour / cell_color	cell logic	Glow or coloring of a cell, due to the expression of fluorescent proteins or other colored molecules.	ccol_	Subsection 2.4.1, Figure 2.8, Table 2.12

Chapter 3

Biological elements

3.1 Gene expression

3.1.1 Molecule

Keywords: `molecule protein`
Recommended prefixes: `mol_ prot_`
State: presence (digital)
Linked to (direct): `Randomness` or equivalent if stochastic, `Function` or equivalent for custom expression times
Linked to (reverse): `Operon` (source), `Flux`, `CellType`, digital logic elements (`BGate`, `Cell Colour`), any conditional logic or biological element as condition, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `Molecule` represents any non-DNA intracellular molecule with digital representation, typically a protein. It is generated by `Operons` (Subsection 3.1.2), involving the necessary synthesis/degradation time delays. Refer to Figure 3.1 and Table 3.1 for the comprehensive list of fields associated with the `Molecule`.

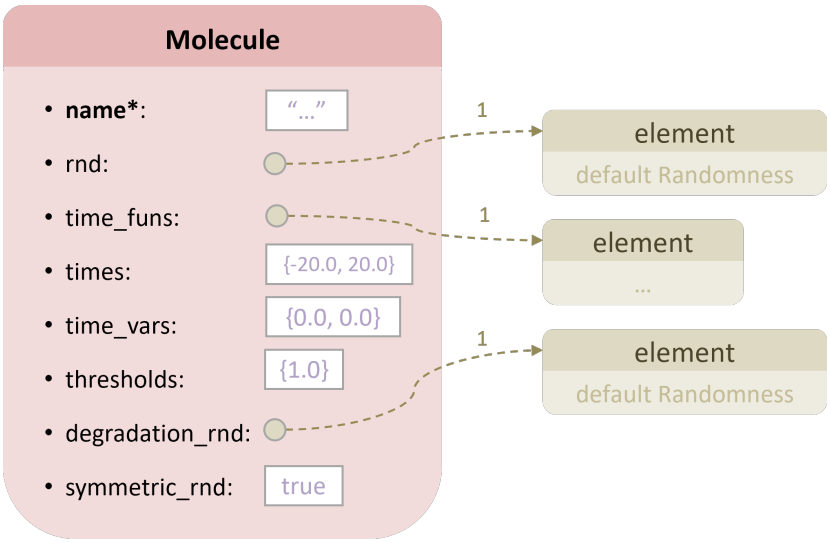

Figure 3.1: `Molecule` **element**

Relationship with gro 2 The `Molecule` serves as the equivalent element that supersedes the gro 2 `Protein`, primarily designed to represent intracellular proteins

within a digital model of gene expression. To emphasize this equivalence, the `protein` keyword is also accepted as a synonym for `molecule`. However, users have the flexibility to employ it to represent RNA, metabolites, and other small molecules as needed.

Connections and state The `Molecule` element is primarily connected to the `Operon` (Subsection 3.1.2) and `Flux` (Subsection 3.2.1) elements, which act as sources producing these molecules. Additionally, it is associated with the `Cell Type element` (Subsection 3.6.4), where it specifies which `Molecules` are already present in newly created cells.

Given that the `Molecule` can be stochastic in terms of its synthesis and degradation times, `Randomnesses` (Subsection 2.1.2) can be linked to it to control this behaviour. The duration of these periods might depend on the state of other cellular elements, making various quantitative cellular elements, both logical and biological, linkable to the `Molecule`. `Functions` (Subsection 2.2.1) are particularly useful for describing durations as custom functions of one or more input elements.

As a Boolean element, `Molecules` possess a digital state, either present (1) or absent (0). This state value can be accessed by the name of the `Molecule` and employed as a Boolean condition for any conditional element. Due to its digital nature, it is well-suited as an input to `BGates` (Subsection 2.3.2). However, it is also possible to use it as a continuous input (present equals 1.0 and absent equals 0.0) for `QGates` Subsection 2.3.1), `Functions`, `Odes`, and `Historics` (Section 2.2). The state of the `Molecule` can be reported through `Cell Colours` (Subsection 2.4.1), `Population Stats`, `Out Files`, and `Plots` (**??**).

Mean activation/degradation times The production and degradation of `Molecules` adhere to a digital model, governed by activation and degradation times. These times are specified in the `times` field, which expects a list of real numbers. In its simplest form, the array comprises two numbers: degradation time followed by activation time (both in minutes). The default values for this field (utilized in the absence of user-defined values) are an activation time of 20.0 minutes and degradation of also 20.0 minutes, which are reasonable defaults for these parameters. To distinguish degradation from activation times, the former must be preceded by a "-" sign, indicating its negative (degradation) nature, as illustrated in Example 3.2.

Example 3.1: Simple `Molecule` with deterministic expression times

```
molecule([ name := "mol_simple"
  , times := { -25.0, 10.0 }
]);
```

Variability By default, expression times exhibit deterministic behaviour (no variability among cells). To introduce heterogeneity, the built-in method involves incorporating normal noise, although users can devise their custom distributions (as detailed later). Users need to specify standard deviations through the `times_var` field (representing expression times variability). This field, an array, aligns with equivalent positions in `times`, creating normal distributions from which times are drawn, with `times` indicating the mean and `times_var` the standard deviation. In the simplest scenario, users provide an array of two values: variability in degradation time followed by variability in activation time. Note that the "-" indicator for degradation is unnecessary here, as the direction is inferred from `times`.

In Example 3.2, the `Molecule` has an average activation time of 9.0 minutes with a standard deviation of 0.1 minutes, and a degradation time of 25.0 minutes on average with a standard deviation of 0.2 minutes.

Example 3.2: Simple `Molecule` with stochastic expression times

```
molecule([ name := "mol_var"
  , times := { -25.0, 9.0 }
  , times_var := { 0.2, 0.1 }
]);
```

Linking `Randomness` elements When expression times are made stochastic, and no `Randomness` element is provided (refer to Subsection 2.1.2), a default normal one is automatically generated to govern the variability in expression times. Alternatively, users have the option to assign a custom `Randomness` through the `rnd` field. The default approach involves using a single `Randomness` element to govern the stochastic behaviour of all associated expression times, covering both synthesis and degradation times. Consequently, the times sampled from different distributions within the same `Molecule` are entirely correlated. For degradation, the symmetric counterparts of the sampled random numbers (derived from the `Randomness`) are utilized instead of the raw ones. This ensures that random numbers generating long synthesis times also yield short degradation times and vice versa. To disable this feature, users can set the `symmetric_rnd` field, defaulting to `true`, to `false`.

Alternatively, users can opt to employ two distinct `Randomness` elements, one for synthesis times and another for degradation times, to eliminate correlation. Assigning a `Randomness` to the `degradation_rnd` field ensures its use exclusively for degradation times, while the one assigned to `rnd` (or the default if none is assigned) continues to govern synthesis times. When the `degradation_rnd` field retains its default setting, the `Randomness` specified at `rnd` (or the default if none is assigned) is applied to both cases.

In more intricate scenarios, users can completely eliminate correlation, making each time distribution independent, by utilizing the `times_fun` field, as elucidated later.

Non-normal expression times The default expectation assumes normal distributions for expression times. In such cases, the `times` and `times_var` fields are interpreted as previously described, representing means and standard deviations. If an alternative distribution family is employed (by assigning a non-normal `Randomness` to `rnd` and/or `degradation_rnd`), the `times` and `times_var` would be re-interpreted accordingly. In the case of a uniform distribution, `times` would represent the lower bounds, and `times_var` would signify the upper bounds. For an exponential distribution, `times` would denote the lambda parameters, and `times_var` would be ignored. When providing a non-`Randomness` element (such as a `Function`), the automatic scaling only functions seamlessly if it is a linear combination of normal `Randomness` (i.e., also a normal). In any other scenario, it is imperative for the user to perform the scaling at the `Functions` instead.

Correlated expression times When multiple `Molecules` are assigned the same `Randomness` element, as depicted in Example 3.3, they "share their variability," implying complete correlation. This signifies that the same quantity of normal noise is incorporated into their expression times: precisely the same value when they possess identical standard deviations or scaled proportionally otherwise. In the given example, the amount of noise introduced to the second `Molecule` is halved due to its corresponding standard deviation parameter. The `Randomness` element is explicitly defined in this example solely to facilitate the shared variability between the two `Molecules`, with all its fields left at their default settings.

Example 3.3: Two `Molecules` sharing a `Randomness`

```
randomness([ name := "rnd_shared" ]);

molecule([ name := "mol_A"
  , randomness := "rnd_shared"
  , times := { -22.0, 10.0 }
  , times_var := { 0.2, 0.1 }
]);

molecule([ name := "mol_B"
  , randomness := "rnd_shared"
  , times := { -10.0, 5.0 }
  , times_var := { 0.1, 0.05 }
```

```
]);
```

Multiple activation/degradation times Given that both the activation and degradation times hinge on the dynamics of the sources of the molecule, each `Molecule` can feature an unbounded number of alternative degradation and activation times. These times should be presented in a sorted manner, ranging from "less production" to "more production," adhering to the following rules:

- All degradation times precede the activation times and are denoted with a "-" sign.
- Degradation times are arranged in ascending order (not considering the "-"), as "more production" corresponds to longer degradation times.
- Activation times are arranged in descending order, as "more production" translates to shorter activation times.

The user has the flexibility to assign a distinct standard deviation for each of the times. The array of mean times takes precedence over the variability array, its size determines the actual number of times loaded into the `Molecule`. If `times_var` exceeds the length of `times`, the rightmost values are disregarded. Conversely, if `times_var` is shorter than `times`, it is padded with zeros on the right, indicating that the rightmost times will be deterministic.

Thresholds The `thresholds` field anticipates a list of real numbers, deployed to determine the expression direction (synthesis or degradation) and its duration, based on the activation levels from the associated sources (`Operons` and `Fluxes`, see Subsection 3.1.2 and Subsection 3.2.1, respectively) of the `Molecule`.

When there is only one degradation and one activation time, the threshold between them defaults to 0.5. This implies that the `Molecule` is produced when the combined activation provided by the `Operons` and `Fluxes` reaches at least 0.5, and degradation occurs otherwise. With n expression times defined, there must be $n-1$ thresholds. The default array is an incrementing series of $n-1$ elements starting at 0.5 and growing in increments of 1.0. For instance, the default array of thresholds for a `Molecule` with 5 times would be `{0.5, 1.5, 2.5, 3.5}`. Users can override this array at `thresholds`. Similar to the behaviour of variabilities, if this array exceeds the length of `times`, the rightmost values are disregarded. Conversely, if it is shorter, it is augmented with the requisite number of values, incrementing by 1.0, at the array's end.

Example 3.4 exemplifies the scenario of multiple expression times. This `Molecule` possesses two potential degradation times and three activation times, each assigned

to named variables for clarity. The commented arrays indicate how the user-provided ones are automatically completed. In this instance, the `Molecule` is produced within 5 minutes when its activation reaches or exceeds 13; within 10 minutes when it is at least 12 but less than 13; and, on average, within 20 minutes with a standard deviation of 0.05 when the activation is at least 10 but less than 12. Degradation occurs, on average, within 30 minutes with a standard deviation of 0.1 when the activation is at least 1 but less than 10. Alternatively, degradation occurs, on average, within 20 minutes with a standard deviation of 0.2 when the activation falls below 1.

Example 3.4: Multiple activation and degradation times in a `Molecule`

```
t_actiVeryFast := 5.0;
t_actiFast := 10.0;
t_actiSlow := 20.0;
t_degFast := 20.0;
t_degSlow := 30.0;

molecule([ name := "mol_A"
  , times := { -t_degFast, -t_degSlow, t_actiSlow,
     t_actiFast, t_actiVeryFast }
  , times_var := { 0.2, 0.1, 0.05 }
  //{0.2, 0.1, 0.05, 0.0, 0.0}
  , thresholds := { 1.0, 10.0, 12.0 }
  //{1.0, 10.0, 12.0, 13.0}
]);
```

Utilizing multiple activation and degradation times is an advanced feature primarily employed when the following aspects are integral to the model:

- Distinct `Operons` and/or `Fluxes` generate the same `Molecule` at varying rates.
- Different `Operons` and/or `Fluxes` collaborate to expedite the production of a shared `Molecule` compared to each working independently.
- The variability in expression times adheres to a multimodal distribution, approximated by a mixture of Gaussians.

Custom functions for expression times Instead of independent values or distributions, expression times can be made dependent on the states of other elements by employing custom `Functions` or other elements. These are assigned to the `time_funs` field. Each non-null (not `""`) entry in this array supersedes the values of `times` and

`times_var` at the corresponding index. Both may be utilized simultaneously, with values from `times` and `times_var` applied to indexes lacking a valid `time_funs`. Custom functions do not utilize the common `rnd` (or `degradation_rnd`) but can be made stochastic, utilizing the inheritable randomness system by utilizing stochastic `Functions`. To indicate the direction is degradation, the name of the element is prefixed with `"-"`, mirroring the convention for `times`. Custom `Functions` allow the creation of intricate behaviours, such as using different `Randomness` for each time or bypassing the built-in time selection system to implement a custom one based on specific conditions rather than activations.

Use of `Molecule` elements The subsequent sections (Subsection 3.1.2 and Subsection 3.1.3) provide further examples of the utilization of `Molecules` in the context of gene expression. The outlined cases will be expounded upon in more detail throughout these sections.

3.1.2 Operon

Keywords: `operon`
Recommended prefixes: `op_`
State: presence (quantitative)
Linked to (direct): `Regulation`, `Molecule`
Linked to (reverse): `Plasmid`, logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `Operon` functions as a source for one or more `Molecules`, with the capacity for constitutive or conditional production. While typically representing a bacterial operon, it has the flexibility to embody the regulated production of various `Molecules`, including RNA or metabolites. Refer to Figure 3.2 and Table 3.2 for a comprehensive list of fields associated with the `Operon` element.

Connections and state The `Operon` functions as a regulatory unit facilitating the synthesis of a specific set of `Molecules` under certain conditions, governed by a shared `Regulation`. Integral to the `Operon` structure are its constituents: the n `Molecules` (**??**) it generates and the associated `Regulation` (**??**), which serves as a promoter or other control point. Allocation of `Operons` occurs within `Plasmids` (**??**, **??**), molecular carriers residing within cells.

As quantifiable entities, `Operons` exhibit both a binary state, denoted by presence (1) or absence (0), and a continuous state reflecting their actual copy number. When

Table 3.1: Parameters of the Molecule element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
rnd / randomness	any cellular element, usually Random-ness	ref.	Governs the stochasticity in the expression times.	default normal Randomness
time_funs / times_funs	array of any cellular element, usually Function	ref.	Custom functions for the expression times. They shadow the "times" and "time_vars" at the same index. Prefix with "-" to represent a degradation time.	{}
times / times_mean / time_means / times_means	array of real	min	Mean synthesis/degradation times. Degradation times are preceded by a "-". Typically sorted in ascending order. If "rnd" not normal, its first param.	{-20.0, 20.0} = degradation in 20 min and synthesis in 20 min
times_var / time_vars / times_vars	array of positive real	min	Stdedv of normal variability in synthesis/degradation times.If it has less elements than mean_times, it is completed with as many 0.0 as needed to match lengths; if it has more elements, the rightmost ones are ignored. If "rnd" not normal, its second param.	{0.0, 0.0} = no variability
thresholds	array of real	—	Thresholds used for choosing a synthesis/degradation time from the list depending on the activation provided by sources. If "times" has length n, this one must have size n-1. Must be sorted in ascending order.	{0.5} when mean_times is also default or of length 2. If longer, n-1 thresholds with a distance of 1.0 are generated: {0.5, 1.5, 2.5, ..., n-1}
symmetric_rnd	Boolean	_	If true, the symmetric of "rnd" is used for degradation direction. Only if "rnd" is of Randomness type	true
degradation_rnd	any cellular element, usually Random-ness	ref.	Used to provide a different randomness source for the degradation times. Typically used when "rnd" is not of Randomness type and symmetric values cannot be calculated automatically.	_

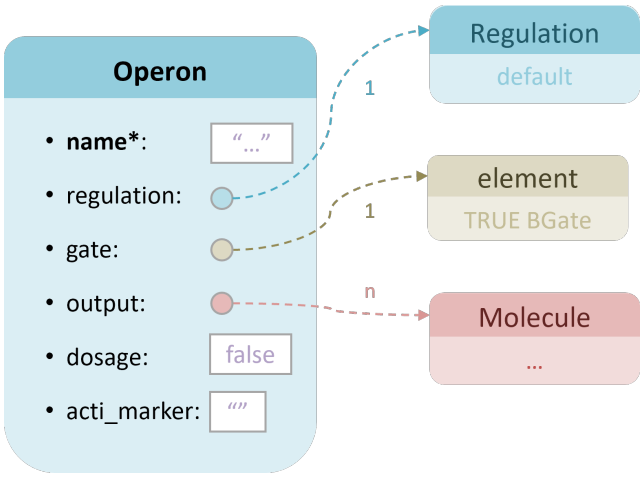

Figure 3.2: `Operon` **element.**

situated within `QPlasmids` (??), the continuous state signifies the total copies of the `Operon` in a cell. Conversely, when employing `BPlasmids` (??), it represents the count of distinct `BPlasmids` harbouring the `Operon` within the cell. Both of these states can be accessed via the `Operon`'s name. The choice between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for `BGates` (??); or as a continuous input for logic elements (`QGates`, ??), `Functions`, `Odes`, and `Historics` (Section 2.2). The `Operon`'s status can be reported through `Cell Colours` (??), `Population Stats`, `Out Files`, and `Plots` (??).

Output The `output` field constitutes an array of `Molecule` elements that the `Operon` generates. This field is optional, with the default being an empty array, resulting in an `Operon` that produces no relevant elements. Example 3.5 showcases a basic `Operon` that constitutively produces a single `Molecule`. It is important to note that even with a single output, the use of list brackets is necessary. Once the `Operon` is present within a cell, the `Molecule` will emerge in 10 minutes.

Example 3.5: Simple `Operon` for constitutive gene expression

```
molecule([ name := "mol_A", times := {-20.0, 10.0} ]);
operon([ name := "op_A", output := {"mol_A"} ]);
```

Regulated gene expression The `gate` field accepts a `Gate`, which can be either Boolean (BGate, ??) or quantitative (`QGate`, ??). If a `Gate` is specified, the output

`Molecules` will be generated only when the `Gate` evaluates to true. By default, the assigned `Gate` is `"_ga_true"` (??), ensuring constitutive expression.

For the production of output `Molecules`, the `Gate` must spend a duration equal to or exceeding their activation times (??) in the `TRUE` state. Conversely, for output degradation, the `Gate` must spend a time equal to or greater than their degradation times in the `FALSE` state. Given variations in expression times among different output `Molecules`, it is plausible for some outputs to be produced or degraded while others remain unaffected. In Example 3.6, the `Operon` yields two `Molecules` contingent upon the presence of its transcription factor, `"mol_tf"`. `"mol_B"` manifests within a brief 5-minute span, whereas `"mol_A"` necessitates a 10-minute interval for production.

Example 3.6: Conditional `Operon` for regulated gene expression

```
molecule([ name := "mol_tf" ]);
molecule([ name := "mol_A", times := {-20.0, 10.0} ]);
molecule([ name := "mol_B", times := {-30.0, 5.0} ]);

operon([ name := "op_AB"
  , gate := "mol_tf"
  , output := {"mol_A", "mol_B"}
]);
```

Channing `Operons` Multiple `Operons` can be sequentially linked to construct gene regulatory networks. The formation of loops, of any conceivable length, is permissible, exemplified by the Repressilator oscillating biocircuit in Example 3.7.

Example 3.7: Repressilator with `Operons`

```
molecule([ name := "mol_A" ]);
molecule([ name := "mol_B" ]);
molecule([ name := "mol_C" ]);
bgate([ name := "bga_notA", input := {"-mol_A"} ]);
bgate([ name := "bga_notB", input := {"-mol_B"} ]);
bgate([ name := "bga_notC", input := {"-mol_C"} ]);

operon([ name := "op_A"
  , gate := "bga_notC"
  , output := {"mol_A"}
]);
```

```
operon([ name := "op_B"
  , gate := "bga_notA"
  , output := {"mol_B"}
]);

operon([ name := "op_C"
  , gate := "bga_notB"
  , output := {"mol_C"}
]);
```

Quantitative conditions are also valid, as illustrated in Example 3.8, where the `Operon` will only produce `"mol_A"` when its own copy number is at least 2. Please note that this condition refers to the absolute amount and not the concentration.

Example 3.8: `Operon` with quantitative condition

```
molecule([ name := "mol_A" ]);
qgate([ name := "qga_copyNum", input := {"op_A"}, value :=
   2.0 ]);

operon([ name := "op_A"
  , gate := "qga_copyNum"
  , output := {"mol_A"}
]);
```

Advanced features through `Regulation` elements Instead of a simple `Gate`, a more intricate element, the `Regulation`, can be designated using the `regulation` field. By default, no `Regulation` is assigned, and a default one is automatically generated using the `Gate` specified at `gate`. This default `Regulation` produces an activation of 1 when in the `ON` state and 0 when in the `OFF` state. The subsequent section (**??**) will elaborate on custom `Regulations` and their application to `Operons`.

`Operons`' activation Unlike gro 2, the `Operon` and the generation of output `Molecules` are not directly interconnected. Even in cases of constitutive expression, an intermediate evaluation occurs: verifying whether the `Operon`'s contribution to synthesis is sufficient to elevate the concentration above the predefined threshold, thus confirming the presence of the `Molecule`.

By default (when the default `Regulation` is assigned), each `Operon` holds an activation of 1 for the production of its outputs when active (fulfilling the `gate` condition) and 0 otherwise. This value is arbitrary and lacks relevance in absolute terms, representing no biologically measurable magnitude. Its significance lies solely in relation to the `thresholds` of the `Molecule` elements (**??**). With the default threshold at 0.5, when the `Operon` is active, synthesis prevails ($(1 > 0.5)$), while inactivity results in degradation ($(0 < 0.5)$). Adjusting this threshold at the `Molecule` level can modify default behaviour. In Example 3.9, the threshold is set to 1.5, rendering `"op_A"` incapable of producing the `Molecule` ($(1 < 1.5)$), ensuring the selection of the degradation direction independently of the `Operon`'s state.

Example 3.9: Weak `Operon` not producing the output

```
molecule([ name := "mol_A"
  , times := {-20.0, 10.0}
  , thresholds := {1.5}
]);

operon([ name := "op_A", output := {"mol_A"} ]);
```

Dosage effect This functionality facilitates the integration of the effects from multiple `Operons` producing the same `Molecule`. In Example 3.10, two constitutively active `Operons` generate the same `Molecule`. When only one is present, the `Molecule` is produced in 15 minutes ($1 > 0.5$ but $1 < 1.5$). If both are present, the `Molecule` is produced in just 10 minutes due to the summation of their activations ($2 > 1.5$). In the absence of any `Operon`, the `Molecule` undergoes degradation ($0 < 0.5$). Modifying the default activation of each `Operon` is achievable by assigning `Regulation` elements to them, as explained in the next section (Subsection 3.1.3).

Example 3.10: Combined effect of two `Operons`

```
molecule([ name := "mol_shared"
  , times := { -20.0, 15.0, 10.0 }
  , thresholds := { 0.5, 1.5 }
]);

operon([ name := "op_A", output := {"mol_shared"} ]);
operon([ name := "op_B", output := {"mol_shared"} ]);
```

The described behaviour for two distinct `Operons` can also be achieved with two copies of the same `Operon`. Multiple copies of a single `Operon` may coexist in a cell, either placed in a `QPlasmid` or distributed across several `Plasmids` of any type. In the context of `BPlasmids`, each instance counts as one copy and so do the `Operons` it carries, irrespective of the "present" state representing a specific number of copies.

In Example 3.11, the exhibited behaviour aligns with that previously seen in Example 3.10. To achieve this, the `dosage` field is employed. By default, the number of copies is disregarded, and an `Operon` contributes the same activation regardless of dosage. Setting `dosage` to true scales the activation by the number of copies. If the associated `Regulation` element accounts for dosage, the `Operon` automatically accommodates this consideration, even when the `dosage` option is set to false.

Example 3.11: Dosage effect with `BPlasmids`

```
molecule([ name := "mol_A"
  , times := { -20.0, 15.0, 10.0 }
  , thresholds := { 0.5, 1.5 }
]);

operon([ name := "op_A"
  , output := {"mol_A"}
  , dosage := true
]);

bplasmid([ name := "bp_A1", operons := {"op_A"} ]);
bplasmid([ name := "bp_A2", operons := {"op_A"} ]);
```

In scenarios where dosage is considered, the activation generated by the regulation is multiplied by the number of copies of the `Operon`. Consequently, `"op_A"` from Example 3.11 contributes 2 to the production when both `BPlasmids` are present, resulting in a synthesis time of just 10 minutes. Tailoring custom dosage effects is accomplished through modelling at the `Regulation` elements, as detailed in Subsection 3.1.3.

Tracking an `Operon`'s activation To expose and facilitate the utilization of the activation provided by an `Operon` as an input for other elements, users can assign a name to the `acti_marker` field. This user-defined name serves as a reference to access the total activation, accounting for all copies in the case of dosage effects.

Table 3.2: Parameters of the Operon element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
regulation	Regulation	ref.	Regulation (promoter) that rules the expression of the output molecules	Automatic ExactRegulation using default values and "gate"
gate	any cellular element, usually Gate	ref.	Gate used to auomatically create a default ExactRegulation in case the noise and dosage fucntionality is not used	"_ga_true" = default TRUE Gate
output	array of Molecule	ref.	List of Molecules that the operon produces as output. The activation of the Regulation is transmitted to them.	{} = no output
dosage / dosage_effect	Boolean	—	If true, the unitary activation of the Regulation is multiplied by the dosage of the operon. Automatically set to true if the linked "regulation" has a dosage effect.	false
acti_marker	string	_	Used to expose and access the activation per copy	_

3.1.3 Regulation

Keywords: `regulation promoter`
Recommended prefixes: `reg_ prom_`
State: presence (quantitative)
Linked to (direct): `Gate` or equivalent as condition, `Randomness` or equivalent if stochastic, `Function` or equivalent for dosage effect
Linked to (reverse): `Operon`, logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description A `Regulation` augments a `Gate` (**??**), converting its Boolean output into a continuous one. It is employed within an `Operon` (Subsection 3.1.2) to ascertain its strength. Refer to **??** and **??** for a comprehensive list of fields associated with the `Regulation` element.

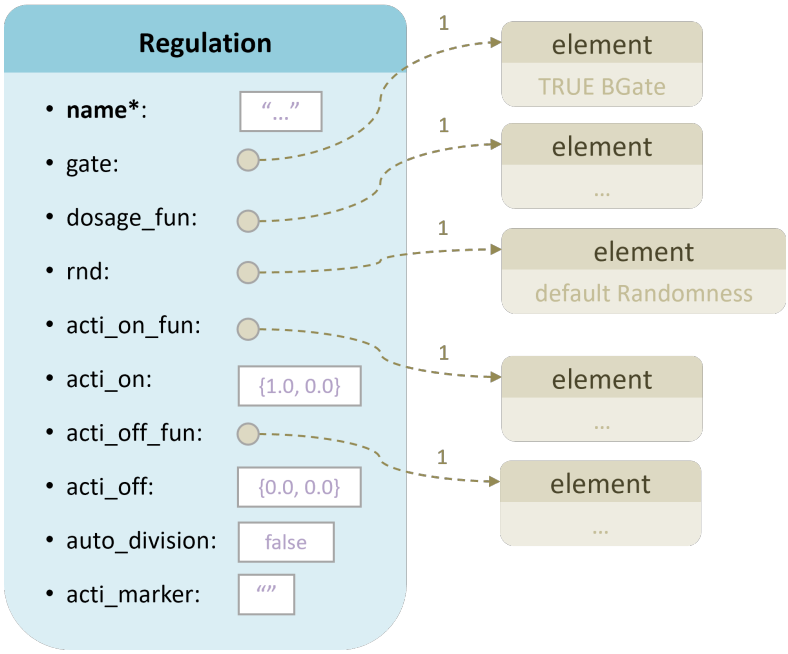

Figure 3.3: `Regulation` **element**

The extended Gate A Regulation serves as a sophisticated element, often representing a promoter. Functioning as an extension of a Gate, it is instrumental in modelling diverse control mechanisms, encompassing promoters, riboswitches, translational regulation, or post-translational modifications. Notably, in the gro 2 framework, the equivalent to Promoter is the BGate (Subsection 2.3.2), being Regulation introduced as an extension. In gro 3.0, the terms regulation and promoter are synonymous.

The gate field within a Regulation refers to the Gate that the Regulation extends. This Gate can be either a BGate (Subsection 2.3.2) or a QGate (Subsection 2.3.1). In cases where no specific Gate is referenced, the predefined TRUE BGate (see Subsection 2.3.2) is automatically assigned.

The primary purpose of incorporating a Regulation element lies in the adjustment of an Operon's strength, thereby regulating the extent of activation it produces. Acting as an intermediary converter, the Regulation transforms the digital output of the Gate into a continuous activation value. This resultant activation value is then conveyed to the Operon, influencing the production of its Molecules. In instances where there is no necessity to modify the strength of the Operon, it is more direct to assign the Gate directly to the Operon.

Connections and state The primary element linked to the Regulation is the Gate it extends, representing the regulatory condition and accepting either a BGate (Subsection 2.3.2) or a QGate (Subsection 2.3.1). Furthermore, Regulations generating stochastic activation (i.e., exhibiting noise) incorporate a Randomness element (Subsection 2.1.2) to govern that aspect. In cases where the strength or activation dynamically relies on the state of other elements, these elements are connected to the Regulation. Function (Subsection 2.2.1) elements are commonly utilized as inputs to create custom functions involving the influencing input elements. Additionally, a Regulation may be linked to itself or other Regulation elements to delineate dosage effects and interference.

Regulations are consistently embedded within Operon elements (Subsection 3.1.2), fulfilling the role of promoters or other regulatory points. As DNA components, Regulations are indirectly housed within Plasmids (??, ??) through their encompassing Operons. The sharing of a single Regulation among multiple Operons is allowed and can give rise to interference effects, as elucidated later.

As quantifiable entities an similarly to Operons, Regulations exhibit both a binary state, denoted by presence (1) or absence (0), and a continuous state reflecting their actual copy number. When situated within QPlasmids (??), the continuous state signifies the total copies of the Regulation in a cell. Conversely, when employing BPlasmids (??), it represents the count of distinct BPlasmids harbouring the Regulation within the cell. Both of these states can be accessed via the Regulation's name. The choice between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for BGates (??); or as a continuous input for logic elements (QGates,

??), Functions, Odes, and Historics (Section 2.2). The Regulation's status can be reported through Cell Colours (??), Population Stats, Out Files, and Plots (??).

Setting the outputted activation The control of activation output involves two key parameters: acti_on and acti_off. Specifically, acti_on represents the activation value produced when the designated gate (at gate) evaluates to true, while acti_off signifies the activation output when the assigned gate evaluates to false. Regulations with the gate field left as default (TRUE BGate) invariably yield acti_on.

Example 3.12 illustrates two Operons sharing a common Molecule product. The "op_weak" generates default activation values (1 and 0), whereas the activation value produced by "op_strong", operating constitutively in this case, has been doubled through the implementation of an Regulation intermediate.

Example 3.12: Operons with different strengths

```
molecule([ name := "mol_shared"
  , times := {-20.0, 20.0, 10.0, 7.0}
  , thresholds := {0.5, 1.5, 2.5}
]);

operon([ name := "op_weak", output := {"mol_shared"} ]);

regulation([ name := "reg_strong", acti_on := 2.0 ]);

operon([ name := "op_strong"
  , regulation := "reg_strong"
  , output := {"mol_shared"}
]);
```

When only "op_weak" is active, "mol_shared" is synthesized within 20 minutes ($1 > 0.5 and 1 < 1.5$) (see Subsection 3.1.1 for activation times). In the exclusive presence of "op_strong", the production of "mol_shared" is achieved in 10 minutes ($2 > 1.5 and 2 < 2.5$). Concurrent activation of both operons results in the production of the Molecule within a shorter duration of 7 minutes ($3 > 2.5$). Conversely, absence of both operons leads to the degradation of the molecule within 20 minutes ($0 < 0.5$).

In Example 3.13, a leaky promoter is depicted, representing an Operon capable of producing a significantly elevated basal level of its Molecule product even in its inactive state. This behaviour is simulated by adjusting the activation output when the operon is off (i.e., when its assigned Gate evaluates to true) using the acti_off parameter within an intermediate Regulation.

Example 3.13: Leaky Promoter

```
molecule([ name := "mol_tf" ]);

molecule([ name := "mol_low"
  , times := {-5.0, 100.0, 1.0}
  , thresholds := {0.05, 0.5}
]);

operon([ name := "op_nice"
  , gate := "mol_tf"
  , output := {"mol_low"}
]);

regulation([ name := "reg_leaky", acti_off := 0.1 ]);

operon([ name := "op_leaky"
  , regulation := "reg_leaky"
  , output := {"mol_low"}
]);
```

Upon activation of either or both of the `Operons`, the production of `"mol_low"`, representing a low yet relevant concentration of a specific protein, is achieved within a brief timeframe of 1 minute ($1 > 0.5 or 2 > 0.5$). When only `"op_nice"` is present and inactive, degradation occurs within 5 minutes ($0 < 0.05$). However, in the presence of `"op_leaky"`, even in its inactive state, `"mol_low"` is still produced, albeit at a prolonged duration of 100 minutes ($0.1 > 0.05 and 0.1 < 0.5$).

Stochastic activation Both parameters (`acti_on` and `acti_off`) can take the form of a single independent value (deterministic), a probability distribution, or a custom `Function` contingent on the state of other elements (Subsection 2.2.1). When an array of values is assigned to `acti_on` and/or `acti_off` instead of a single value, it is interpreted as the parameters of a probability distribution to draw activation from. By default, a normal `Randomness` is automatically created and employed with the default re-sampling schedule, shared by both parameters. Alternatively, a custom `Randomness` (Subsection 2.1.2) can be designated at the `rnd` field to utilise a different distribution family, thereby imparting a distinct significance to the parameters array or modifying the re-sampling schedule.

It is essential to observe that when utilized in conjunction with the smoothing features of the `Randomness` (`inertia` and/or `inheritance` other than 0.0 or 1.0, see

Subsection 2.1.2), distributions other than the normal undergo modifications. Despite the apparent simplicity of the uniform distribution, its amalgamation with these features results in its transformation into a triangular distribution.

Dynamic activation An additional option is to substitute `acti_on` (`acti_off`) with `acti_on_fun` (`acti_off_fun`), enabling the use of a custom `Function` (Subsection 2.2.1) or the state of any other element type as the activation value, thus making the `Regulation`'s activation dynamic. Leveraging stochastic `Functions` facilitates the incorporation of a different `Randomness` for each of the two parameters. Deploying the "function version" (`acti_on_fun`, `acti_off_fun`) eclipses the simple parameter field (`acti_on`, `acti_off`) and the application of `rnd` for that parameter.

Simulating expression noise Stochastic activation is employed to simulate expression noise, encompassing both positive and negative noise. In Example 3.14, the `acti_on` parameter indicates a normal distribution (in the absence of a custom `Randomness` assigned to the `rnd` field) with a mean of 1.0 and a standard deviation of 0.0, manifesting as a deterministic production of a value of 1. Consequently, when the associated `Regulation` is activated, the `Molecule` will be produced constitutively ($1 > 0.5$).

On the other hand, `acti_off` generates a normal distribution with a mean of 0.0 and a standard deviation of 0.3. In instances where the `Regulation` is deactivated, the `Molecule` is mostly not produced; however, there exists a probability of 0.04779 for this distribution to yield values greater than 0.5. Consequently, there is a probability of 0.04779 associated with the occurrence of positive noise.

Example 3.14: Positive noise with a stochastic `Regulation`

```
molecule([ name := "mol_A"
  , times := {-20.0, 20.0}
  , thresholds := {0.5}
]);

regulation([ name := "reg_noisy"
  , acti_on := {1.0, 0.0}
  , acti_off := {0.0, 0.3}
]);

operon([ name := "op_noisy"
  , regulation := "reg_noisy"
  , output := {"mol_A"}
]);
```

In Example 3.15, an augmentation of negative noise is introduced alongside the positive noise depicted in the preceding example, with a probability of 0.00621. Within this probability, the normal distribution with a mean of 0.0 and a deviation of 0.2 is capable of generating values falling below the 0.5 threshold of the `Molecule` (see Subsection 3.1.1 for information on `Molecule`'s thresholds).

Example 3.15: Positive and negative noise with a stochastic `Regulation`

```
molecule([ name := "mol_A"
  , times := {-20.0, 20.0}
  , thresholds := {0.5}
]);

regulation([ name := "reg_noisy"
  , acti_on := {1.0, 0.2}
  , acti_off := {0.0, 0.3}
]);

operon([ name := "op_noisy"
  , regulation := "reg_noisy"
  , output := {"mol_A"}
]);
```

Multimodal variability In **??**, the `acti_on` is represented by a normal distribution with a mean of 10.0 and a standard deviation of 0.1. When the `Regulation` is active, it samples values from this distribution. Notably, as the mean aligns with the second threshold of the `Molecule`, half of the samples lead to rapid synthesis, while the other half result in slow synthesis. Contrasting with the previous examples, in this scenario, there is no alteration in the direction (synthesis vs degradation); rather, there is a shift in the synthesis time distribution. This distinction from the variability introduced in previous instances is essential; using only typical variability yields unimodal distributions, while this feature generates multimodal distributions.

In the provided example (**??**), the resultant distribution is a Gaussian mixture comprising two Gaussians: one with a mean of 20 and a deviation of 0.5, and the other with a mean of 10 and a deviation of 0.1. The weights assigned to these Gaussians are 0.5 each. This approach proves valuable in simulating heterogeneous populations, encompassing diverse subpopulations with notable variation in activation.

Example 3.16: Weak noise with a stochastic `Regulation`

```
molecule([ name := "mol_A"
  , times := {-20.0, 20.0, 10.0}
  , times_var := {0.2, 0.5, 0.1}
  , thresholds := {0.5, 10.0}
]);

regulation([ name := "reg_noisy"
  , acti_on := {10.0, 0.001}
  , acti_off := {0.0, 0.0}
]);

operon([ name := "op_noisy"
  , regulation := "reg_noisy"
  , output := {"mol_A"}
]);
```

Dosage effect The activation generated by a `Regulation` element is consistently construed as the activation contributed by each copy. In cases where the `Operon` carrying the `Regulation` is copy number sensitive (refer to Subsection 3.1.2), the activation transmitted to its associated `Molecules` is the product of the value outputted by the `Regulation` and the copy number of the `Operon`. Should the user-defined activation be the cumulative activation of all copies, there are two approaches: either include the division by the copy number in the activation's formula or utilize the `auto_division` option. By default, this option is set to `false`. When enabled, the activation is automatically divided by the number of `Regulation` copies, yielding the activation per copy.

As described, the default dosage effect is linear, but users have the flexibility to modify it, introducing non-linear saturating effects. The `dosage_fun` field allows for the incorporation of a custom `Function` (or another element) to act as the scaler for activation. The raw activation generated by the `Regulation` element undergoes multiplication by the state of the assigned element. Given that this field is specifically tailored to introduce non-linearities associated with the dosage effect, it is anticipated that users will assign a `Function` whose inputs, either directly or indirectly, involve the `Regulation` itself, considering its state is the copy number. Although achieving the same effect by introducing the copy number directly at `acti_on_fun` and `acti_off_fun` is feasible, the dedicated `dosage_fun` field offers convenience for clarity. Custom quantitative dosage effects are commonly employed in conjunction with `Operons` situated within `QPlasmids` (Subsection 3.6.2).

In Example 3.17, the `Operon` features a bespoke `Regulation` element incorporating

a saturating exponential function as the `dosage_fun`. The inherent raw activation when activated is set to the default value of 1.0.

Example 3.17: `Operon` with quantitative copy number and saturating `Regulation`

```
molecule([ name := "mol_A"
  , times := {-20.0, 15.0}
  , thresholds := {1.0}
]);

function([ name := "fun_total"
  , input := {"reg_A"}
  , type := "sat_exp"
  , params := {1.0, 0.1}
  , auto_vol := "division"
]);

function([ name := "fun_individual"
  , input := {"fun_total", "reg_A"}
  , type := "product"
  , params := {1.0, -1.0}
]);

regulation([ name := "reg_A", dosage_fun := "fun_individual
   " ]);

regulation([ name := "reg_same"
  , dosage_fun := "fun_total"
  , auto_division := true
]);

operon([ name := "op_A"
  , regulation := "reg_A"
  , output := {"mol_A"}
  , dosage := "conc"
]);

qplasmid([ name := "qp_A", operons := {"op_A"} ]);
```

Therefore, the aggregate activation produced is calculated as follows:

$$activation = 1.0 \cdot 1.0 \cdot (1 - e^{-0.1 \cdot [R]}) \tag{3.1}$$

And the activation per `Regulation` copy is:

$$activation = \frac{1.0 \cdot 1.0 \cdot (1 - e^{-0.1 \cdot [R]})}{[R]} \tag{3.2}$$

The division operation is explicitly implemented in `"fun_individual"`, utilised by `"reg_A"`. An alternative and more straightforward approach, achieving the same outcome, is illustrated by `"reg_same"`. This configuration directly employs `"fun_total"` along with the `auto_division` option. It is noteworthy that the `dosage` option of `"op_A"` is automatically enabled upon associating a `Regulation` with a custom dosage `Function` (as mentioned in Subsection 3.1.2); manual intervention for activation is unnecessary.

This dosage effect introduces a non-linear behaviour that saturates at sufficiently high copy numbers. The function encapsulates the total activation contributed by all copies of this `Regulation`. In this context, the activation values specified in `acti_on` and `acti_off` represent the maximum activation achieved at saturation, rather than the individual activation in the presence of a single copy.

Multiple `Operons` sharing a `Regulation` In Example 3.18, two distinct `Operons` generate different `Molecules` while sharing a common `Regulation` element (with an associated saturating dosage `Function` as the custom `dosage_fun`). Both operons, having the same promoter, contribute to the same pool or `Regulation`. The presence of each `Operon` exerts a negative influence on the other, given their shared regulatory element. Consequently, the total activation amplifies with the increasing number of copies; however, the activation per concentration unit diminishes, as in the previous examples.

Example 3.18: Dosage effect in `Operons` sharing a `Regulation`

```
molecule([ name := "mol_A" ]);
molecule([ name := "mol_B"]);

function([ name := "fun_total"
  , input := {"reg_A"}
  , type := "sat_exp"
  , params := {1.0, 0.1}
  , auto_vol := "division"
]);

regulation([ name := "reg_shared"
```

```
  , dosage_fun := "fun_total"
  , auto_division := true
]);

operon([ name := "op_A"
  , regulation := "reg_shared"
  , output := {"mol_A"}
]);

operon([ name := "op_B"
  , regulation := "reg_shared"
  , output := {"mol_B"}
]);
```

In scenarios where multiple `Operons` share the same `Regulation`, the total activation is distributed proportionally. For instance, if `"op_A"` comprises 5 copies and `"op_B"` features 10 copies, approximately 2/3 of the total activation will be allocated to `"op_B"`, and the remaining 1/3 to `"op_A"`. It is essential to note that when dealing with different `Operons`, even if they share the same parameters, assigning a distinct `Regulation` to each of them is imperative to prevent interference. Interference can be avoided entirely when a `Gate` is assigned instead of a `Regulation` (refer to Subsection 3.1.2).

Accessing the activation value The activation generated by a `Regulation` is an internal value that users are not inherently required to access. To expose and facilitate its utilization as an input for any other element, a name must be assigned to the `acti_marker` field. This user-specified name will subsequently serve as the identifier to retrieve the value. It is pertinent to note that the value corresponds to a single copy of the `Regulation` in the case of dosage effect usage.

3.2 Metabolism

The metabolic cellular elements serve as the connection between the intracellular cellular system and the extracellular environment, governed by the medium elements (**??**). Currently, there is a single element in this category, the `Flux` (Subsection 3.2.1). This element is employed for the exchange of molecules such as nutrients, metabolites, molecular signals, and toxins between the cell and its surrounding medium, playing a crucial role in intercellular communication and ecological interactions.

Table 3.3: Parameters of the Regulation element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
gate	any cellular element, usually Gate	ref.	The logic Gate that the Regulation extends	"_ga_true" = default TRUE Gate
dosage_fun	any cellular element, usually Function	ref.	Custom fucntion that multiplies the raw activation to scale it by dosage.	_ = no dosage scaling
rnd / randomness	any cellular element, usually Random-ness	ref.	Randomness that rules the stochasticity in the activation output	default normal Randomness
acti_on_fun	any cellular element, usually Function	ref.	Custom function for the activation when the "gate" is true. Shadows "acti_on" and "rnd".	—
acti_on / acti_on_params	real or array of real	—	Amount of activation outputted when the "gate" evaluates to true. Either a single deterministic value or the params for the distribution given by "rnd".	{1.0, 0.0} mean and stddev
acti_off_fun	any cellular element, usually Function	ref.	Custom function for the activation when the "gate" is false. Shadows "acti_off" and "rnd".	_
acti_off / acti_off_fun	real or array of real	—	Amount of activation outputted when the "gate" evaluates to false. Either a single deterministic value or the params for the distribution given by "rnd".	{0.0, 0.0} mean and stddev
auto_division	Boolean	_	Must be set to true when the prodived "dosage_fun" produces the total activation of all the copies, to get the activation per copy.	false
acti_marker	string	_	Used to expose and access the activation per copy	_

3.2.1 Flux

Keywords: `flux`
Recommended prefixes: `flux_`
State: flux (exchange speed)
Linked to (direct): `Molecule`, `Signal`, `Randomness` or equivalent if stochastic, `Function` or equivalent if custom, `Gate` or equivalent as condition
Linked to (reverse): logic elements (`Function`, `Ode`, `Historic`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `Flux` element serves as a representation of quantitative metabolic flux in a cell. It functions as the element responsible for exchanging molecular signals with the environment and responding to them. Refer to Figure 3.4 and Table 3.4 for the comprehensive list of fields associated with the `Flux` element.

Connections and state The `Flux` element is typically linked to logic elements, such as `Gates` (**??**) if it is conditional, `Randomness` (Subsection 2.1.2) if the exchanged amount and/or the required concentration threshold are stochastic, and `Function` (Subsection 2.2.1) to represent a custom dynamic exchanged amount or rate (although any element type can be directly linked). The sole biological element connected to the `Flux` is `Molecule` (Subsection 3.1.1). This connection allows `Fluxes` to function as sources, similar to `Operons` (Subsection 3.1.2).

`Fluxes` exhibit both a binary state, denoted by activity (1) or inactivity (0), and a continuous state reflecting the exchanged amount of `Signal`. The sign of the continuous state indicates the direction of the flux: negative for absorption while positive for emission. The digital state is considered 1 whenever the exchanged amount is non-zero, irrespective of its direction, and 0 otherwise. Both of these states can be accessed via the `Flux`'s name. The selection between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for `BGates` (**??**); or as a continuous input for logic elements (`QGates`, **??**), `Functions`, `Odes`, and `Historics` (Section 2.2). The `Flux`'s status can be reported through `Cell Colours` (**??**), `Population Stats`, `Out Files`, and `Plots` (**??**).

Exchanged `Signal` and growth rate The `signal` field is employed to designate the `Signal` (**??**) that is either emitted or absorbed by the `Flux` element. By default, this field is set to `"_biomass"`, a special keyword utilized to denote the growth rate (refer to the details below).

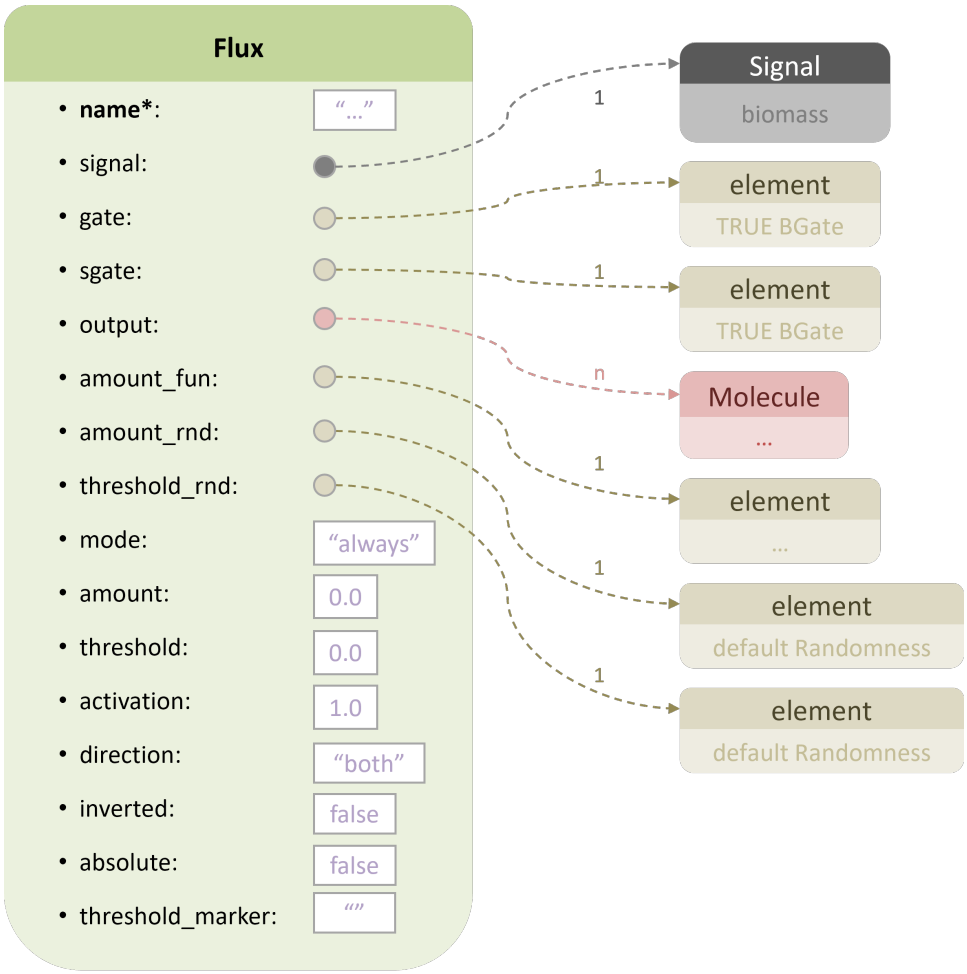

Figure 3.4: `Flux` **element**

Exchanged amount The `amount` field determines the magnitude of the flux, with positive values representing emission and negative values indicating absorption. The flux can exhibit either determinism or stochasticity, introducing variability within the population. A single assigned value implies determinism, while an array of real numbers is construed as the scaling parameters for a probability distribution, defaulting to a normal distribution.

The distribution family can be modified by assigning a custom `Randomness` (Subsection 2.1.2) to the `"amount_rnd"` field. By default, a standard normal `Randomness` is automatically generated, interpreting the `amount` parameters as the mean and standard deviation. If a single value is assigned or the second parameter is set to 0.0, the amount is rendered deterministic, and no `Randomness` is automatically created.

By default, the `amount` is 0.0, indicating no exchange of `Signal`. An empty list (or a single value instead of a list) signifies no scaling of the distribution if an `"amount_rnd"` is explicitly assigned. In the absence of such assignment, it implies determinism, and the `Randomness` is not automatically generated.

Direction The `direction` field serves to constrain the flux direction, allowing for the specification of absorption-only or emission-only behaviour. This proves particularly useful when the amount is stochastic and has the potential to take incorrect signs. By default, the `direction` is set to `"both"`, signifying the allowance for a change in direction. The field accepts either string keywords or signed integers; any positive number is equivalent to `"emission"`, negative numbers denote `"absorption"`, and 0 implies `"both"`.

Units and scaling by cellular volume The `amount` is expressed on a per-minute basis, with the value multiplied by the time step size to determine the amount exchanged during each time step. By default, the sampled flux values are considered relative to cell volume: custom amount units per μm^3 per minute. The total amount exchanged per minute is computed as the product of the sampled value and the current cell volume. To bypass this scaling step, the `absolute` parameter, `false` by default, must be set to `true`. In the case of negative (absorption) fluxes, their amount is automatically restricted to the available quantity of `Signal` in the medium surrounding the cell, ensuring that cells cannot absorb more `Signal` than what is present.

Example on the exchanged amount Example 3.19 presents a pair of `Fluxes` operating on the same `Signal`. `"flux_Aout"` serves as an emission flux, while `"flux_Ain"` functions as an absorption flux. Given that `"flux_Aout"` is stochastic, the `direction` functionality is employed to ensure that it consistently remains an emission flux.

Example 3.19: Simple absorption and emission Fluxes

```
signal([ name := "s_A", colour := _magenta ]);
randomness([ name := "rnd_custom", inertia := 0.0 ]);

flux([ name := "flux_Aout"
  , signal := "s_A"
  , rnd := "rnd_custom"
  , amount := {10.0, 0.1}
  , direction := "emission"
]);

flux([ name := "flux_Ain"
  , signal := "s_A"
  , amount := -1.0
  , absolute := true
]);
```

The emitted amount per μm^3 of cell volume for `"flux_Aout"` follows a normal distribution with a mean of 10 and a standard deviation of 0.1, devoid of inertia. The actual emitted amount (per minute) is determined by multiplying this value by the current cellular volume. In contrast, `"flux_Ain"` is deterministic, with the absorbed amount consistently set to 1 (or the available amount if it is less than 1). Additionally, `"flux_Ain"` operates in an absolute manner, independent of the cellular volume.

Dynamic flux The amount exchanged can be made a dynamic value contingent on the state of any quantitative cellular element, typically `Functions` (Subsection 2.2.1) and `Odes` (Subsection 2.2.2). This capability allows the `Flux` to depend on other `Fluxes`, the concentration of any `Signal` in the surrounding medium, or quantitative amounts of intracellular molecules calculated through `Odes`. If specified, `amount_fun` supersedes the `amount` parameter, and the stochasticity must be set at the `amount_fun` level, employing a stochastic `Function` (see Subsection 2.2.1 for reference) instead of the built-in randomness scaling system.

The `inverted` option, deactivated by default, automatically reverses the sign of the calculated amount at `amount_fun`. This feature proves convenient in automatically establishing the direction of the flux without necessitating explicit sign inversion within a custom `Function`.

Example 3.20 illustrates the utilization of a Hill `Function` with an order of 1 (equivalent to Michaelis-Menten kinetics) (see Subsection 2.2.1) to model active transport. In this model, the rate is contingent on the concentration of the `Signal` in the medium,

with a maximum rate of 10 and a Hill constant of 50. The `inverted` feature is employed to automatically invert the sign of the amount provided by the Hill `Function`, considering this is an absorption `Flux` (i.e., negative).

It is crucial to observe the use of `"_conc_s_A"` instead of `"s_A"` when accessing the concentration of the `Signal` named `"s_A"`. When the name of a `Signal` (`"s_A"`) is employed as an input for a `Function` or other element, it signifies the total flux or exchange of that `Signal` occurring within the cell at that specific moment (the derivative with respect to time). To retrieve the concentration in the surrounding medium instead, the prefix `"conc"` must be appended.

Example 3.20: Custom absorption rate

```
signal([ name := "s_A", colour := _magenta ]);

function([ name := "fun_Ain1"
  , input := {"_conc_s_A"}
  , type := "hill"
  , params := {10.0, 50.0, 1.0}
  , rnd_params := {0.0, 1.0}
]);

function([ name := "fun_Ain2"
  , input := {"_conc_s_A"}
  , type := "hill"
  , params := {10.0, 50.0, 1.0}
]);

flux([ name := "flux_Ain"
  , signal := "s_A"
  , amount_fun := "fun_Ain1"
  , inverted := true
]);

flux([ name := "flux_same"
  , signal := "s_A"
  , amount_fun := "fun_Ain2"
  , amount := {0.0, 1.0}
  , inverted := true
]);
```

The exchanged amount in this scenario is stochastic, conforming to a normal distribution with a mean equivalent to the magnitude outputted by the Hill function (the mean parameter is set to 0, indicating no added bias) and a standard deviation of 1.0. This stochastic behaviour can be achieved through two equivalent approaches: either utilizing `rnd_params` in the `Function`, as demonstrated in `"flux_Ain"`, or incorporating `amount` directly in the `Flux`, exemplified by `"flux_same"`.

Interconnected `Fluxes` In Example 3.21, two `Fluxes` are interconnected in a chain-like fashion so that the amount of `"s_B"` emitted by `"flux_Bout"` becomes contingent on the amount of `"s_A"` absorbed by `"flux_Ain"`. Specifically, the emission value (derivative) of `"flux_Bout"` is set at 0.5 times that of `"flux_Ain"`, which, in turn, is influenced by the concentration of `"s_A"` in the medium. The automated inversion of the sign is unnecessary here since it is already implemented in `"fun_Bout"`. This establishes a direct link between two `Fluxes`, where the dependent `Flux` updates as soon as `"flux_Ain"` undergoes a change. If a delayed link is desired, an intermediary `Molecule` or `Ode` can be employed, as demonstrated in subsequent examples.

Example 3.21: `Flux` chaining

```
signal([ name := "s_A", colour := _magenta ]);
signal([ name := "s_B", colour := _cyan ]);

function([ name := "fun_Ain"
  , input := {"_conc_s_A"}
  , type := "hill"
  , params := {10.0, 50.0, 1.0}
  , rnd_params := {0.0, 1.0}
]);

flux([ name := "flux_Ain"
  , signal := "s_A"
  , amount_fun := "fun_Ain"
  , inverted := true
]);

function([ name := "fun_Bout"
  , input := {"flux_Ain"}
  , type := "sum"
  , params := {-0.5}
]);
```

```
flux([ name := "flux_Bout"
  , signal := "s_B"
  , amount_fun := "fun_Bout"
]);
```

Modes Two alternative modes, designated by the `mode` field, govern the behaviour of the `Flux`. The default mode is `"always"`, signifying that the `Flux` is executed every time step throughout the entire lifespan of a cell. Alternatively, the `"lysis"` mode stipulates that the flux is executed only once, specifically upon cell death. The `"lysis"` mode is employed to simulate the release of cellular components into the medium upon cell lysis, while the `"always"` mode accommodates simulations of any flux transpiring while cells are alive, encompassing those affecting the growth rate (see later in this section). See Example 3.26 for an example on cell lysis.

Conditional fluxes Similar to most biological elements in gro 3.0, `Fluxes` can be made conditional by assigning a `Gate`. The `Flux` element will be only executed if the `Gate` evaluates to true. Conditions may involve intracellular elements, such as `Molecules` required for the production or exchange of the `Signal`, as well as the concentration of specific `Signals` in the surrounding environment falling within a specified range.

For convenience, two separate fields are designated for `Gates`: `gate` and `sgate` (signals gate). The `AND` logic operation is automatically applied to both fields, ensuring that the `Flux` executes its function only when both conditions are `true`. By default, both `gate` and `sgate` are set to the predefined `"_ga_true"` BGate (see Subsection 2.3.2). The recommended usage is to assign `Gates` evaluating intracellular conditions to `gate`, and `Gates` assessing the concentration of `Signals` in the environment to `sgate`. Failure to adhere to this convention may result in undefined behaviour. The extracellular concentration of `Signals` is accessed by adding the `"conc"` prefix to the `Signal`'s name (see Subsection 4.0.2).

Example 3.22 presents two `Cell Types`, one emitting a `Signal` and the other absorbing it. Both `Fluxes` involve the same quantity, differing only in sign, and both exhibit stochastic behaviour. The `Cell Type` serves as the condition, dictating which cells emit the `Signal` and which ones absorb it.

Example 3.22: Conditional `Fluxes`

```
signal([ name := "s_A", colour := _magenta ]);
cell_type([ name := "cell_source" ]);
cell_type([ name := "cell_sink" ]);
```

```
flux([ name := "flux_Aout"
  , signal := "s_A"
  , gate := "cell_source"
  , amount := {10.0, 0.1}
]);

flux([ name := "flux_Ain"
  , signal := "s_A"
  , gate := "cell_sink"
  , amount := {-10.0, 0.1}
]);
```

`Signal` threshold When the external condition involves a simple threshold for the concentration of a `Signal` in the medium, it can be conveniently specified using the `threshold` parameter instead of `sgate`. This threshold can be made stochastic by providing a list of real numbers. By default, the threshold is deterministic and set to 0.0, indicating no threshold. If a list is assigned, these values are considered as the scaling parameters of a random distribution, being the mean and standard deviation of a normal distribution by default. In this case, the default normal `Randomness` is automatically generated. To utilize a different distribution family or sampling schedule, users can assign a custom `Randomness` (Subsection 2.1.2) through the `threshold_rnd` field. The usage of the `threshold` field in illustrated in Example 3.23.

To access the current value of a stochastic threshold, users can utilize the `threshold_marker` field. By assigning a string name to this field, users can subsequently employ it as an input for other elements.

The same functionality can be achieved through the general approach of employing a `QGate` at `sgate` and an intermediary stochastic `Function` between the `Signal` and the `QGate`. However, utilizing the `threshold` parameter leads to cleaner code. Both `sgate` and `threshold` can be employed in the same `Flux` element, with the element effectively performing an `AND` operation over the two `Gates` and the threshold.

Link to gene expression Metabolism and gene expression can be interconnected in various ways. The simplest method involves utilizing the built-in connection feature through the `output` and `activation` fields of the `Flux` element. `output` anticipates a list of `Molecules`, akin to the same field in `Operon` (Subsection 3.1.2). The `Flux` contributes to the production of the designated `Molecules` in a manner similar to an `Operon`. The contributed activation is established by the `activation` parameter, defaulting to 1.0. Unlike the `Operon`, the `Flux` cannot be assigned a custom `Regulation`. The outputted

activation is consistently deterministic, and dosage effects would be nonsensical in this context. For a more intricate connection between a `Flux` and a `Molecule`, one can establish a chain of `Operons` downstream from the `output` of the `Flux`.

In Example 3.23, the connection to gene expression is exemplified along with the usage of a simple threshold. `"flux_sensor"` becomes active when the concentration of `"s_A"` exceeds 100, with a standard deviation of 1.0. While the amount is left as default (zero), signifying no signal absorption or emission, the active `Flux` (`"flux_sensor"`) contributes to the synthesis of `"mol_A"`. `"mol_A"` materializes if the `Flux` remains active for at least 5 minutes and undergoes degradation when inactive for the same duration. Serving as a delayed marker of the `Flux` state, `"mol_A"` can be employed to further interconnect additional `Flux` elements. The behaviour mirrors that of `"op_A"`. The state of `"flux_sensor"` is true when active (i.e., when its associated `Gates` and threshold are satisfied), even if the exchanged amount is zero in this instance.

Example 3.23: Linking a `Flux` to gene expression

```
signal([ name := "s_A", colour := _magenta ]);
molecule([ name := "mol_A", times := {-5.0, 5.0} ]);

flux([ name := "flux_sensor"
  , signal := "s_A"
  , output := {"mol_A"}
  , threshold := {100.0, 1.0}
]);

//equivalent
operon([ name := "op_A", gate := "flux_sensor", output := {
   "mol_A"} ]);
```

The `output` and `activation` features of the `Flux` element offer a simpler means of linking metabolic `Fluxes` and gene expression. Nevertheless, `Operons` provide greater flexibility through the use of custom `Regulations`. In Example 3.24, the linkage of a `Flux` to gene expression is demonstrated, incorporating an intermediate custom `Regulation` to introduce additional stochasticity.

Example 3.24: Linking a `Flux` to custom `Operons`

```
signal([ name := "s_A", colour := _magenta ]);
molecule([ name := "mol_A", times := { -5.0, 5.0 } ]);
```

```
flux([ name := "flux_sensor", signal := "s_A", threshold :=
    100.0 ]);

noisy_regulation([ name := "nreg_A"
  , gate := "flux_sensor"
  , acti_on := {1.0, 0.1}
  , acti_off := {0.0, 0.0}
]);

operon([ name := "op_A", regulation := "nreg_A", output :=
   {"mol_A"} ]);
```

Linking `Fluxes` and `Odes` In Example 3.25, an `Ode` is employed to monitor the internal concentration of the absorbed `Signal`, `"s_A"`. In this scenario, the transport is passive, relying on its sign and magnitude to derive from the concentration disparity between the cell and the environment. `"fun_A"` calculates the exchanged amount as the difference multiplied by the transport constant. While `"flux_A"` only produces changes in the medium, the concentration within the cell is traced by `"ode_A"`. `"fun_Aconc"` transforms the magnitude of `"flux_A"` into the derivative needed by the `Ode`, inverting the sign and dividing by the cellular volume to obtain the concentration (as the `Flux` is in amount units). Although using `"fun_A"` as the input for `"fun_Aconc"` would yield the same result, it is recommended to use the `Flux` itself due to the potential for conditional `Fluxes`.

Example 3.25: Linking a `Flux` to Odes

```
k_transport := 0.01;

signal([ name := "s_A", colour := _magenta ]);

function([ name := "fun_A"
  , input := {"_conc_s_A", "ode_A"}
  , type := "sum"
  , params := {-k_transport, k_transport}
]);

flux([ name := "flux_A"
  , signal := "s_A"
  , amount := "fun_A"
```

```
]);

function([ name := "fun_Aconc"
  , input := {"flux_A"}
  , type := "sum"
  , params := {-1.0}
  , auto_vol := "division"
]);

ode([ name := "ode_A",  input := "fun_Aconc" ]);
```

Link to growth rate In addition to exchanging `Signals` with the medium, `Flux` elements can modify the growth rate. Biomass `Fluxes` represent a special case of `Flux` that lacks a linked `Signal` (their `Signal` is designated as `"_biomass"`, the default value for the `signal` field). The default functionality involves adding the raw amount of all biomass `Fluxes` directly to the base growth rate defined by the `Strain`, while it is possible to specify a different custom behaviour at the `Strain` level (refer to Subsection 3.6.3).

In Example 3.26, cell lysis is triggered by a toxin in the medium, leading to the release of cellular components into the medium. `"flux_sensor"` detects the presence of the toxin, becoming active when the toxin concentration exceeds 10, with a standard deviation of 1. Although the same behaviour can be achieved with a `Function`, using a `Flux` is considered better practice. `"flux_death"` is a biomass `Flux` that modifies the growth rate, activated when `"flux_sensor"` is active, reducing the growth rate by 1. Given the base growth rate of the default `Strain` (`"_str_wt"`) is around 0.035 min^{-1}, the resulting growth rate will always be negative. In `"_str_wt"`, as it is the default behaviour (see Subsection 3.6.3), negative growth rates imply cell death. Consequently, cells exposed to toxin concentrations sufficient for `"flux_sensor"` activation will die. `"flux_lysis"` is the `Flux` simulating the release of cellular contents on cell lysis, symbolized by a single `Signal`, `"s_cytosol"`. The `"lysis" mode` ensures that this `Flux` is executed only once, upon cell death.

Example 3.26: Cellular lysis by a toxin

```
signal([ name := "s_toxin", colour := _magenta ]);
signal([ name := "s_cytosol", colour := _cyan ]);

flux([ name := "flux_sensor"
  , signal := "s_toxin"
```

```
  , threshold := {10.0, 1.0}
]);

flux([ name := "flux_death", gate : "flux_sensor", amount
   := -1.0 ]);

flux([ name := "flux_lysis"
  , signal := "s_cytosol"
  , mode := "lysis"
  , amount := 10.0
]);
```

In Example 3.27, a similar scenario is presented, but in this case, the effect on the growth rate of the sensed signal is positive. The `Strain` grows at its maximum rate (0.035 min^{-1}) when nutrients in the medium are present at a concentration of at least 100, with a deviation of 1. Otherwise, it grows slowly (0.02 min^{-1}), with a standard deviation of 0.002 in both cases. Similar to the previous example, `"flux_nutrient"` serves as the sensor, and `"flux_growth1"` acts as the biomass `Flux`. Additionally, an alternative method using a `Function` to achieve the same behaviour is demonstrated. Note the use of the `"conc"` prefix to access the concentration of the `Signal`.

Example 3.27: Growth rate composition

```
strain([ name := "str_A", growth_rate := {0.02, 0.002} ]);
signal([ name := "s_nutrient", colour := _magenta ]);

flux([ name := "flux_nutrient"
  , signal := "s_nutrient"
  , threshold := {100.0, 1.0}
]);

flux([ name := "flux_growth1"
  , gate := "flux_nutrient"
  , amount := 0.015
]);

//equivalent
function([ name := "fun_nutrient"
  , input := {"_qs_s_nutrient"}
  , rnd_params := {0.0, 1.0}
```

```
]);

qgate([ name := "qga_nutrient", input := "fun_nutrient",
   value := 100.0 ]);

flux([ name := "flux_growth2"
  , gate := "qga_nutrient"
  , amount := 0.014
]);
```

3.3 Plasmid dynamics

This section encompasses DNA components exclusive to `Quantitative Plasmids` (`QPlasmids`) (Subsection 3.6.2), which are responsible for governing their intracellular dynamics. The `OriV` (Subsection 3.3.1) is a crucial element required for `QPlasmid` replication, while `Copy Control` elements (Subsection 3.3.2) can be introduced to constrain the copy number attained by the `QPlasmid`. Additionally, the `Partition System` (Subsection 3.3.3) is the element that dictates how `QPlasmid` copies are distributed among daughter cells during cell division.

3.3.1 OriV

Keywords: `oriv` `ori_v` `oriV`
Recommended prefixes: `ov_`
State: presence (quantitative)
Linked to (direct): `Copy Control`, `Randomness` or equivalent if stochastic, `Function` or equivalent if custom rate
Linked to (reverse): `QPlasmid`, logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `OriV` is a crucial component enabling the replication of `QPlasmids`. They facilitate explicit quantitative replication, copy by copy, throughout the cell's entire life span. This process is resource-intensive compared to other elements, and the presence of `OriVs` in the simulation can negatively impact performance. While it usually represents a genuine plasmidic origin of replication, it can also denote other sequence types, such as a viral origin of replication. Refer to Figure 3.5 and Table 3.5 for the comprehensive list of fields associated with the `OriV` element.

Table 3.4: Parameters of the Flux element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
signal / metabolite	Signal	ref.	Signal absorbed or emitted by the Flux. The special name "_biomass" is used to modify the growth rate. All the biomass Fluxes are added to the base growth rate.	"_biomass"
gate / internal_gate	any cellular element, usually Gate without Signals and Fluxes as input	ref.	Condition on the intracellular elements. The Flux only takes place if both the "gate" and the "signals_gate" evaluate to true.	"_ga_true" = default TRUE Gate
sgate / s_gate / signals_gate	any cellular element, usually Gate with only Signals and Fluxes as input	ref.	Condition on the Signals in the sourrinding environment of the cell and other Fluxes. The Flux only takes place if both the "gate" and the "signals_gate" evaluate to true.	"_ga_true" = default TRUE Gate
output	Molecule	ref.	Molecules produced by the Flux	"" = no output
amount_fun	any cellular element, usually Function	ref.	Custom optional Function that determines the amount of Signal absorbed/emitted.	"" = constant value or Distribution
amount_rnd	any cellular element, usually Randomness	ref.	Randomness that rules the stochasticity in the amount of Signal absorbed/emitted.	"" = automatically created default normal
threshold_rnd	any cellular element, usually Randomness	ref.	Randomness that rules the stochasticity in the threshold of Signal required.	"" = automatically created default normal
type / mode	string in { "always", "lysis" }	_	In "always" mode, the Flux is checked and executed every time step. In "lysis" mode, the Flux is executed once when the cell dies.	"always"
amount / amount_params	real or array of real	user-defined amount unit	Either a single deterministic value or parameters of the distribution describing the amount of "signal" exchanged. Their meaning depend on the distribution type of "amount_rnd". If an "amount_fun" is given, the Function and the distribution samples are added (the distribution acts as added noise).	{ 0.0, 0.0 } = no exchange
threshold	real	user-defined concentration unit	Either a single deterministic value or the parameters of the distribution describing the required minimum concentration of "signal" in the medium. This simple condition must hold together with "gate" and "signals_gate".	0.0
activation	real	_	Amount of activation transmitted to the "output" when the Flux is active	1.0
direction	string in "absorption", "emission", "both" or int in -1, 1, 0	_	Used to lock the direction of the flux. If the sampled amount gets the opposite sign, it is made zero. "absorption" = -1, "emission" = 1	"both" = 0
inverted / is_inverted	Boolean	_	If true, the amount is inverted (opposite sign i.e. opposite direction).	false
absolute / is_absolute	Boolean	_	If true, the amount is directly exchanged with the medium. If false (the typical desired behaviour), the amount is scaled by the cellular volume.	false
threshold_marker	string	_	User-given name to access the value of a stochastic "threshold" param	_

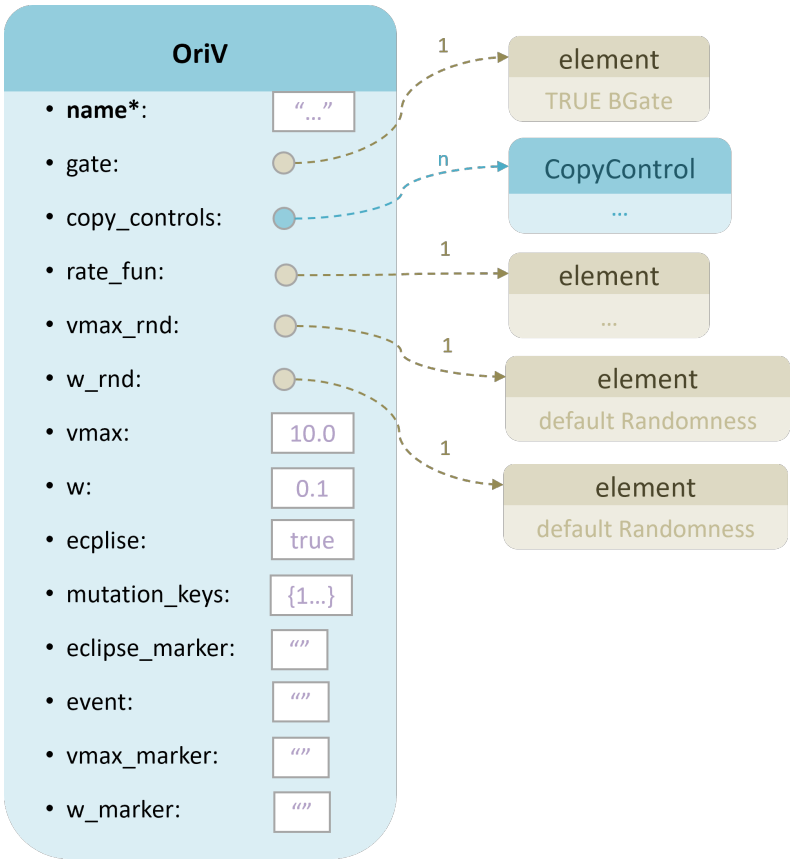

Figure 3.5: `OriV` **element**

Connections and state The OriV element is directly linked to Copy Control elements (Subsection 3.3.2), incorporating copy number control for the OriV. OriVs with stochastic replication rates employ Randomness elements to govern that behaviour, whereas those with dynamic rates dependent on the state of other elements connect to them, particularly Functions (Subsection 2.2.1).

The OriV is a DNA component of QPlasmids (Figure 3.13), essential for their replication. As quantifiable entities, OriVs exhibit both a binary state, denoted by presence (1) or absence (0) within a cell, and a continuous state reflecting their actual copy number. Since they are located within QPlasmids (??), the continuous state signifies the total copies of the OriV in a cell. Both of these states can be accessed via the OriV's name. The selection between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for BGates (??); or as a continuous input for logic elements (QGates, ??), Functions, Odes, and Historics (Section 2.2). The OriV's status can be reported through Cell Colours (??), Population Stats, Out Files, and Plots (??).

Replication events The replication events originating from a specific OriV can be monitored through the event field. The user-assigned name is employed to retrieve the count of events in the last time step. This count can serve various purposes, such as generating a response to the events (e.g. incurring a metabolic cost) or simply for reporting, logging, and/or counting, leveraging tools like Cell Colour (Subsection 2.4.1) and Population Stats (??).

Number of OriVs A QPlasmid can accommodate an unlimited number of OriVs. A QPlasmid lacking OriVs is considered a suicide plasmid, as it cannot replicate. When a QPlasmid possesses multiple OriVs, it can replicate independently from any of them, potentially at varying rates and under different conditions. Consequently, the total replication rate of the QPlasmid surpasses that of a counterpart with a single OriV. The optional eclipse period following replication can selectively impact only a subset of the OriVs. To simplify, the elongation phase of the replication process is not explicitly modelled. All components of the QPlasmid are replicated simultaneously, making the selection of a specific OriV irrelevant. The conclusion of the eclipse period similarly affects all components simultaneously.

Conditional behaviour Like most elements, the OriV exhibits conditional behaviour: it only functions when a specified condition is met. The condition can be set using the gate field, and replication events from the OriV take place only when the assigned Gate (??) (or any other element with a digital state) evaluates to true. By default, a TRUE BGate is assigned, ensuring that the OriV is always active.

In Example 3.28, an `OriV` is active only when `"mol_A"` is present. `"mol_A"` is the product of the `Operon` within the same `QPlasmid`, but it takes 10 minutes to be produced. Consequently, when a `QPlasmid` first appears in a cell, replication does not commence immediately.

Example 3.28: Conditional replication

```
molecule([ name := "mol_A", times := {20.0, 10.0} ]);
operon([ name := "op_A", output := {"mol_A"} ]);
oriv([ name := "ov_A", gate := "mol_A" ]);
qplasmid([ name := "qp_A", operons := "op_A", orivs := {"
   ov_A"} ]);
```

In Example 3.29, the conditional nature of the `OriV` is leveraged to implement strong copy number control. The `OriV` is active only when the concentration of copies of the `OriV` is less than 100 copies/μm^3. This conditional activation ensures a stringent regulation of the replication process, maintaining the copy number within the specified limit. Nevertheless, as discussed below, special considerations are necessary when utilizing copy number control in conjunction with post-replication eclipse periods.

Example 3.29: Strongly regulated copy number

```
qgate([ name := "qga_cc"
  , input := "ov_A"
  , operator := "<"
  , value := 100.0
  , auto_vol := "division" ]);

oriv([ name := "ov_A", gate := "qga_cc", vmax := 1000.0 ]);
qplasmid([ name := "qp_A", orivs := {"ov_A"} ]);
```

Eclipse period The `OriV` supports an optional eclipse period following replication events. During this period, the two involved copies (the old and the new) are considered not to be properly folded, rendering all their contained elements inactive. The duration of the eclipse period is plasmid-specific and is specified in the `QPlasmid` (see Subsection 3.6.2). The `eclipse` option of the `OriV` determines whether the eclipse period occurs when replication takes place from this `OriV`, with the default being `true`.

By default, any component of `QPlasmid` copies in the eclipse period is ignored (considered as non-present and inactive). This includes `OriVs`; the value accesses

through an `OriV`'s name only includes the copies that are not in the eclipse periof. However this behaviour can be overridden. The number of `OriV` copies in `QPlasmids` undergoing the eclipse period can be accessed using the `eclipse_marker` field. The given name is then used to access the value, making it an input to other elements. For example, users may include the "eclipse" copies in a strong maximum copy number condition created with a `QGate`. If only the "non-eclipse" ones were considered, the copy number might surpass the limit.

The default rate function The replication process is inherently stochastic and is modelled as a Poisson process (Gillespie algorithm), characterized by a rate. The rate is defined as the average number of replication events per minute initiated from an `OriV` of this type, irrespective of the `QPlasmid` it belongs to. gro 3.0 provides a built-in mechanism to easily specify this replication rate or the frequency of replication events.

By default, the rate is assumed to increase with the concentration of the `OriV` in the cell, but not in a linear fashion. The rate eventually saturates to a user-defined maximum rate, given by the `vmax` parameter, set at an arbitrary value of 100 reactions per minute by default. The non-linearity is modelled as a saturating exponential with the parameter `w`. A higher value of `w` leads to a faster approach to the maximum rate. This scaling mirrors the example of dosage effect in gene expression (see Subsection 3.1.3). With the default `w` value of 0.1, approximately 23 copies per μm^3 reach 90% of the maximum rate, and 46 copies reach 99%. Both `vmax` and `w` can be deterministic or stochastic, using the inheritable randomness system (Subsection 2.1.2). If an array of two values is assigned instead of a single value, it is interpreted as the mean and standard deviation of a normal distribution. The re-sampling schedule can be adjusted by assigning a custom `Randomness` (or other element) to the `vmax_rnd` and `w_rnd` fields, respectively. It is also possible to use a distribution other than the normal. To access the last value of an stochastic maximum rate, use `vmax_marker`.

Copy number control The replication rate can be subject to optional capping through a copy number control mechanism, such as antisense RNA or others. Each mechanism is represented by a `Copy Control` element. The `copy_control` field accepts a list of `Copy Controls` that affect the `OriV`. If no `Copy Control` is assigned to the `OriV`, it results in uncontrolled replication, unless it is halted by the condition specified at the `gate` field (explained before in this section). When multiple `Copy Controls` are assigned, their effects are summed by default.

In the provided example (Example 3.31), the `OriV` utilizes the built-in rate description. A `Copy Control` is assigned to the `OriV` and is also placed in the `QPlasmid`, alongside the `OriV`. Additional details on the `Copy Control` element are available in its dedicated section (Subsection 3.3.2).

Example 3.30: Softly regulated copy number

```
copy_control([ name := "cc_A", w := 0.01 ]);

oriv([ name := "ov_A"
  , copy_controls := {"cc_A"}
  , vmax := 1000.0
  , w := 0.1
]);

qplasmid([ name := "qp_A", orivs := {"ov_A"}, copy_controls
    := {"cc_A"} ]);
```

Shared copy number control When multiple `QPlasmids` share the same `OriV` or possess different `OriVs` with identical `Copy Controls` and/or carry these `Copy Control` elements, they interfere with each other's replication, as demonstrated in Example 3.31. Since the parameters of the two `OriVs` in the example have the same values (they have the same strength), when these `QPlasmids` coexist in the same cell, each of them attains only half the copy number it would reach when present alone.

Example 3.31: Softly regulated copy number

```
copy_control([ name := "cc_A", w := 0.01 ]);

oriv([ name := "ov_A"
  , copy_controls := {"cc_A"}
  , vmax := 1000.0
  , w := 0.1
]);

qplasmid([ name := "qp_A1", orivs := {"ov_A"},
   copy_controls := {"cc_A"} ]);
qplasmid([ name := "qp_A2", orivs := {"ov_A"},
   copy_controls := {"cc_A"} ]);
```

Modularity in copy number control In Example 3.32, the modulation of `Copy Controls` activity and the placement of `Copy Controls` and `OriVs` in separate `QPlasmids` are demonstrated. The `"cc_A"` is positioned in a distinct plasmid and remains

inactive when "mol_A", produced by "qp_molA", is present. Consequently, the replication of "qp_A" is regulated solely when "qp_cc" is present, and "qp_molA" is absent. Otherwise, "qp_A" undergoes uncontrolled replication. Both "qp_cc" and "qp_molA" are considered suicide plasmids as they lack an OriV.

Example 3.32: Conditional copy number control

```
molecule([ name := "mol_A", times := {20.0, 10.0} ]);
operon([ name := "op_A", output := {"mol_A"} ]);
bgate([ name := "bga_not_molA", input := {"-molA"} ]);
copy_control([ name := "cc_A", gate := "bga_not_molA", w :=
    0.02 ]);

oriv([ name := "ov_A"
  , copy_controls := {"cc_A"}
  , vmax := 1000.0
]);

qplasmid([ name := "qp_A", orivs := {"ov_A"} ]);
qplasmid([ name := "qp_cc", copy_controls := {"cc_A"} ]);
qplasmid([ name := "qp_molA", operons := {"op_A"} ]);
```

Custom rate functions It is possible to use a custom dynamic rate instead of the built-in system by assigning a Function or other quantitative element to the rate_fun field. In this case, the built-in parameters are disregarded, and the output of the custom Function determines the rate of the replication process.

In Example 3.33, the built-in behaviour is replicated using a custom rate Function. The rate is equivalent to that in Example 3.31.

Example 3.33: Default replication rate with a custom Function

```
copy_control([ name := "cc_A" ]);

function([ name := "fun_rate"
  , input := {"fun_oriV", "cc_A"}
  , type := "exp_product"
  , params := {1000.0, 0.1, -0.02}
  , auto_vol := "division"
]);
```

```
oriv([ name := "ov_A", rate_fun := "fun_rate" ]);
qplasmid([ name := "qp_A", orivs := {"ov_A"}, copy_controls
    := {"cc_A"} ]);
```

In Example 3.34, a custom replication rate function with Hill shape is implemented. The `"fun_cc"` function creates a Hill `Function` of order 1 (Michaelis-Menten kinetics) using the concentration of the `Copy Control` as input, with a maximum value of 1000 and a constant of 5. The result is then multiplied by the concentration of the `OriV` in `"fun_rate"`. It is worth noting that in this custom function, the automatic division by volume is not applied.

Example 3.34: Custom rate `Function`

```
copy_control([ name := "cc_A" ]);

function([ name := "fun_cc"
  , input := {"cc_A"}
  , type := "hill"
  , params := {1000.0, 5.0, 1.0}
  , auto_vol := "division"
]);

function([ name := "fun_rate"
  , input := {"fun_oriV", "_qm_vol", "fun_cc"}
  , type := "product"
  , params := {1.0, -1.0, 1.0}
]);

oriv([ name := "ov_A", rate_fun := "fun_rate" ]);
qplasmid([ name := "qp_A", orivs := {"ov_A"}, copy_controls
    := {"cc_A"} ]);
```

Mutation keys The `mutation_keys` field is utilized to specify a subset of mutations that can occur when the replication is initiated by this `OriV`. For additional details on DNA modifications affecting `Plasmids`, refer to Subsection 3.4.2.

Table 3.5: Parameters of the OriV element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
gate	any cellular element, usually Gate	ref.	Condition for the OriV to be active	"_ga_true" = default TRUE Gate
copy_controls	CopyControl	ref.	Sequence for copy number control that affects this OriV	{} = uncontrolled
rate_fun	any cellular element, usually Function	ref.	Custom rate Function, in. If assigned, vmax and w are ignored.	—
vmax_rnd	any cellular element, usually Randomness	ref.	Rules the stochasticity in the "vmax" param	default normal Randomness
w_rnd	any cellular element, usually Randomness	ref.	Rules the stochasticity in the "w" param	default normal Randomness
vmax / vmax_params	positive real or array of real	events/	Maximum replication rate. Either a single deterministic value or the params for "vmax_rnd"	100.0 (a sensible value, still quite arbitrary)
w	positive real or array of real	um3 / oriv copies	Used to scale the "vmax" depending on the copy number in a non-linear way. The higher the value, the sooner saturation is reached (less linear). Either a single deterministic value or the params for "w_rnd"	0.1 (a sensible value, still quite arbitrary)
eclipse / has_eclipse	Boolean	—	If true, the QPlasmid copies involved in the replication event undergo an eclipse period. The duration is specified in the QPlasmid.	true
mutation_keys / mutations_keys	array of positive integer	_	Keys of groups the replication mutations defined in the QPlasmid that are possible from this OriV.	{1}
eclipse_marker	string	_	To expose and access the number of elements undergoing the eclipse period after replication	_
event / event_marker	string	—	Exposes the number of replication events from this element during the last time step	—
vmax_marker	string	—	User-given name to access the value of a stochastic "w" param	—
w_marker	string	_	User-given name to access the value of a stochastic "vmax" param	—

3.3.2 Copy Control

Keywords: `copy_control`
Recommended prefixes: `cc_`
State: presence (quantitative)
Linked to (direct): `Randomness` or equivalent if stochastic, `Function` or equivalent if dynamic regulatory effect
Linked to (reverse): `OriV` (affects it), `QPlasmid` (container), logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description A `Copy Control` is a DNA component that exerts a negative effect on the replication rate of `QPlasmids` (Subsection 3.6.2) from the affected `OriVs` (Subsection 3.3.1), thereby limiting their maximum copy numbers. The complete list of fields for the `Copy Control` element is provided in Figure 3.6 and Table 3.6.

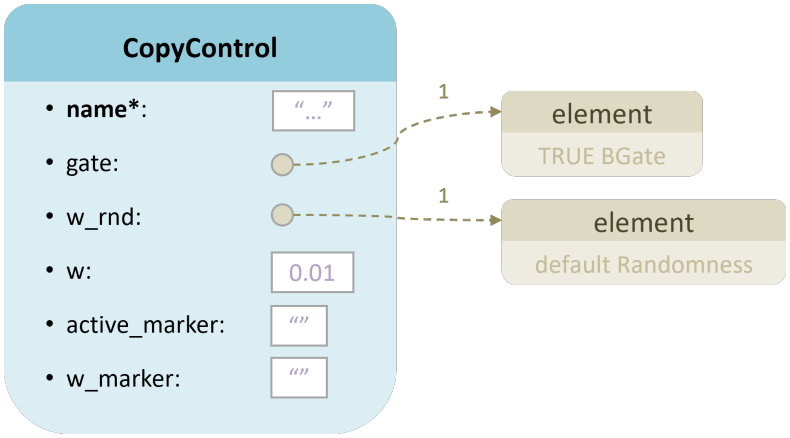

Figure 3.6: `Copy Control` **element**

Mechanism A `Copy Control` can represent various mechanisms of copy number control. It may signify the DNA sequence that produces anti-sense RNA or another molecule that interferes with replication. It is assumed that this molecule is produced rapidly, and therefore, the synthesis process is not explicitly simulated. The number of copies of the `Copy Control` directly influences the replication rate of the affected `OriVs`. If more detailed modelling of the interference process is needed, users can utilize `Operons` (Subsection 3.1.2) and `Molecules` (Subsection 3.1.1) or `Odes` (Subsection 2.2.2)

instead. The `Copy Control` provides a fast and convenient way to introduce replication with copy number control when it is not the main focus of the experiment. The decision on which `OriVs` are affected is made at the `OriV` level, through their `copy_controls` field (see **??** for detailed information).

Connections and state The `Copy Control` element is directly connected to `OriV` elements (Subsection 3.3.1), integrating copy number control for the `OriV`. `Copy Controls` with stochastic replication rates utilize `Randomness` elements (Subsection 2.1.2) to regulate that behaviour, while those with conditional activity connect to a `Gate` (**??**) or another element with binary state as their condition.

The `Copy Control` is a DNA component of `QPlasmids` (Figure 3.13), typically shared with their modulated `OriVs`, although they can exist separately. As quantifiable entities, `Copy Controls` exhibit both a binary state, denoted by presence (1) or absence (0) within a cell, and a continuous state reflecting their actual copy number. Situated within `QPlasmids` (**??**), the continuous state signifies the total copies of the `Copy Control` in a cell. Both of these states can be accessed via the `Copy Control`'s name. The selection between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for `BGates` (**??**); or as a continuous input for logic elements (`QGates`, **??**), `Functions`, `Odes`, and `Historics` (Section 2.2). The `Copy Control`'s status can be reported through `Cell Colours` (**??**), `Population Stats`, `Out Files`, and `Plots` (**??**).

Conditional behaviour Like most elements, the `Copy Control` operates only when its associated `Gate` (or equivalent binary element) evaluates to true. The default `Gate` is `"_ga_true"`, but it can be modified at the `gate` field. When the `gate` evaluates to false, the `Copy Control` is inactive and has no effect, as if it were not present at all.

Strength The strength of the `Copy Control` interference is represented by its `w` parameter, with an arbitrary default value of 0.01. This value does not correspond to any biologically meaningful magnitude. Empirical adjustment is required until the desired copy number is achieved. The strength value should be considered relative to the `w` parameter of the affected `OriVs` (see Subsection 3.3.1). A higher value for `w` results in a faster rate decrease with the copy number, a lower maximum rate (with respect to the `vmax` parameter of the `OriV`), and an earlier onset of rate decrease with the copy number. High values produce tight control with minimal variability in the copy number among cells, while low values result in loose control and high variability. Similar to the `w` parameter of `OriVs`, this value can be deterministic or stochastic, utilizing the inheritable randomness system (Subsection 2.1.2). If an array of two values is given, they are interpreted as the mean and standard deviation of a normal distribution. Other distributions can be applied by assigning a custom `Randomness` (or other element) to

`w_rnd`. To access the current value of a stochastic strength, use `w_marker`, which is assigned a user-defined string name that serves as an identifier for access from other elements.

Active marker While the name of the `Copy Control` element retrieves its raw copy number, the `active_marker` field is employed to access the number of copies that are active, resulting in an equivalent outcome to multiplying the raw copy number by the state of the `gate`. This value equals the raw copy number if the condition is `true` (the element is active) and zero otherwise. The user-given name assigned to the `active_marker` field is then used to access the value. This approach is valuable for users aiming to create their own custom copy number control `Functions` instead of relying on the built-in system, contributing to a reduction in the number of custom `Functions` and enhancing clarity. For examples, refer to the `OriV` section (Subsection 3.3.1).

Table 3.6: Parameters of the CopyControl element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
gate	any cellular element, usually Gate	ref.	Condition for the CopyControl sequence to be active	"_ga_true" = default TRUE Gate
w_rnd	any cellular element, usually Random-ness	ref.	Rules the stochasticity in the "w" param	default normal Randomness
w / w_params	positive real orr array of real	um3/ copy con-trol copies	Strength of the control mechanism. Used to apply a negative effect to the affected OriVs that depends on the number of CopyControl copies in a non-linear way. The higher the value, the strongest. Either a deterministic value or the params for the "w_rnd" distribution	0.01 (a sensible value, still quite arbitrary)
active_marker	string	_	To expose and access the number of active elements (copy number multiplied by the state of "gate")	_
w_marker	string	_	User-given name to access the value of a stochastic "w" param	_

3.3.3 Partition System

Keywords: `partition` `partition_sytem`
Recommended prefixes: `part_`
State: presence (quantitative)
Linked to (direct): `Randomness` or equivalent if stochastic, `Function` or equivalent if dynamic partition
Linked to (reverse): `QPlasmid` (container), logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `Partition System` is a DNA component of `QPlasmids` (Subsection 3.6.2) responsible for facilitating their symmetrical (or asymmetrical) distribution during cell division. The complete list of fields for the `Partition System` element is provided in Figure 3.7 and Table 3.7.

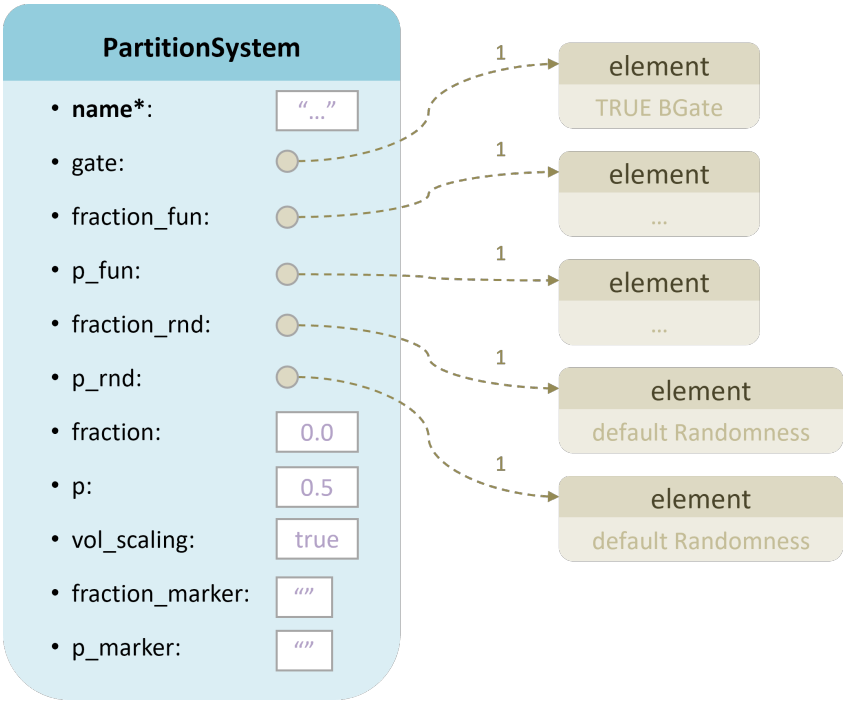

Figure 3.7: `Partition System` **element**

Connections and state The `Partition System` element is directly linked to `Gate` (??) or another element with binary state as conditions for its activity. It may also be connected to `Randomness` (Subsection 2.1.2) or equivalent elements if the copy number split it performs has stochastic components. Additionally, it can be linked to quantitative elements, especially `Functions` (Subsection 2.2.1), to serve as dynamic partition parameters.

The `Partition System` is a DNA component of `QPlasmids` (Figure 3.13). As quantifiable entities, `Partition Systems` exhibit both a binary state, denoted by presence (1) or absence (0) within a cell, and a continuous state reflecting their actual copy number. Situated within `QPlasmids` (??), the continuous state signifies the total copies of the `Partition System` in a cell. Both of these states can be accessed via the `Partition System`'s name. The selection between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for `BGates` (??); or as a continuous input for logic elements (`QGates`, ??), `Functions`, `Odes`, and `Historics` (Section 2.2). The `Partition System`'s status can be reported through `Cell Colours` (??), `Population Stats`, `Out Files`, and `Plots` (??).

Conditional behaviour Similar to most elements, the `Partition System` exhibits conditional behaviour. It performs its function only when a specified condition is met. The condition is provided through a `Gate` (??) or equivalent specified in the `gate` parameter. If the `Gate` evaluates to true during cell division, the copies of the `QPlasmid` are symmetrically distributed between the two daughter cells. The default `Gate` is `"_ga_true"`, which is always true.

In case the `gate` evaluates to false, the `Partition System` does not function, and the copies are randomly distributed following a binomial distribution, with a default value of 0.5 for the probability parameter (`p`). Users can adjust this parameter using the `p` field, with valid values ranging from 0.0 to 1.0. If a `QPlasmid` lacks a `Partition System` element, it is always randomly distributed with $p = 0.5$.

In the provided code snippet (Example 3.35), the behaviour of the `Partition System` is influenced by the presence of `"mol_A"`. If `"mol_A"` is present, the copies of `"qp_A"` are symmetrically distributed, half and half, during cell division. Conversely, in the absence of `"mol_A"`, the distribution of copies follows a random pattern.

Example 3.35: Regulated `Partition System`

```
molecule([ name := "mol_A", times := {20.0, 10.0} ]);
partition([ name := "part_A", gate := "mol_A" ]);
qplasmid([ name := "qp_A", partition := "part_A" ]);
```

Shared `Partition System` When multiple `QPlasmids` share the same `Partition System`, the random distribution remains unaffected, but interference occurs in achieving an exact symmetric split for each individual `QPlasmid`. All copies associated with the `Partition System` are treated as part of a common pool, without distinction between `QPlasmids`. The perfect split is executed based on the total amount in this shared pool, and then specific copies are selected. Consequently, individual `QPlasmids` may not undergo symmetric distribution independently, and there is a possibility of loss for some `QPlasmids`, particularly those with low copy numbers, when they share the same `Partition System`.

In the scenario presented in Example 3.36, both `"qp_A1"` and `"qp_A2"` share the same `Partition System`. In the absence of `"qp_A2"`, the copies of `"qp_A1"` experience a perfect distribution, ensuring that it is not lost despite its low copy number. However, when `"qp_A2"` is present, each daughter cell is expected to receive a concentration of 101 copies/μm, but there is a possibility that all copies of `"qp_A1"` may end up in the same cell due to the shared `Partition System`.

Example 3.36: Shared `Partition System`

```
partition([ name := "par_A" ]);
qplasmid([ name := "qp_A1", partition := "par_A", copy_num
    := 2.0 ]);
qplasmid([ name := "qp_A2", partition := "par_A", copy_num
    := 200.0 ]);
```

Asymmetry in perfect partitions and p parameter Introducing a bias or noise to symmetric partitions is achievable through the `fraction` parameter. This parameter, by default set at 0.5 in a deterministic manner, dictates the fraction of copies directed to one daughter cell (with the other receiving 1 - fraction). Users can assign an exact value or leverage a distribution to add stochasticity. For the latter, the list of real numbers representing the distribution's parameters is assigned to `fraction`. By default, these are interpreted as the mean and standard deviation of a normal distribution. A custom `Randomness` (Subsection 2.1.2) can be assigned to `fraction_rnd` for different distribution types and re-sampling schedules. Similarly, the `fraction_fun` allows the fraction to be dynamic, depending on the state of other elements, overshadowing `fraction` and `fraction_rnd` when set. The same principles apply to the p parameter, with `p_rnd` and `p_fun` serving as counterparts to `fraction_rnd` and `fraction_fun`. To access the current value of a stochastic parameter, employ `fraction_marker` and `p_marker`, respectively.

In Example 3.37, custom values are used for both the perfect and random partition.

A bias towards one of the daughter cells is introduced in both cases. In the perfect split case, a bit of noise is added as well, with the fraction sampled from a normal distribution with a mean of 0.6 and a standard deviation of 0.01.

Example 3.37: Custom `Partition System`

```
molecule([ name := "mol_A", times := {20.0, 10.0} ]);

partition([ name := "par_"
  , gate := "mol_A"
  , fraction := {0.6, 0.01}
  , p := 0.6
]);

qplasmid([ name := "qp_A", partition := "par_A" ]);
```

Optional volume effect During cell division, the daughter cells usually have slightly different volumes. By default, the relative volume of the daughter cells influences the copies they receive in both symmetric and random partition modes. The copy number is treated as a concentration rather than an absolute amount. The daughter cell with the larger size receives more copies of the plasmid or is more likely to do so. To disable this feature, set the `vol_scaling` option to false.

Eclipse marker The `Partition System` is the sole DNA element that remains active even during the eclipse period of `QPlasmids` (see Subsection 3.3.1 and Subsection 3.6.2) to ensure that the eclipsed copies do not alter the partition distribution. The predefined name `"_<partition name>_eclipse"` can be utilized to access the number of copies of the partition that are in the eclipse period after replication. This marker is predefined for internal use.

3.4 Mutation and gene editing

The elements in this section are dedicated to the modification of both `BPlasmids` and `QPlasmids` through the simulation of reactions involving them, such as mutations, recombination, gene editing, etc. This is achieved through two types of elements that work together, `Mutation` and `Mutation Processes`. While `Mutations` (Subsection 3.4.1) describe the stoichiometry of the modification, `Mutation Processes` (Subsection 3.4.2) elucidate the dynamics of the reactions.

Table 3.7: Parameters of the PartitionSystem element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
gate	any cellular element, usually Gate	ref.	Condition for the PartitionSystem sequence to be active. If active, the split is exact. If inactive, the split is random following a binomial distribution.	"_ga_true" = default TRUE Gate
fraction_fun	any cellular element, usually Function	ref.	Custom function for the fraction. Shadows "fraction".	—
p_fun	any cellular element, usually Function	ref.	Custom function for the fraction. Shadows "p".	—
fraction_rnd	any cellular element, usually Random-ness	ref.	Rules the stochasticity in the "fraction" param	default normal Randomness
p_rnd	any cellular element, usually Random-ness	ref.	Rules the stochasticity in the "p" param	default normal Randomness
fraction / frac-tion_params	real in [0.0, 1.0] or array of real	fraction	Fraction of copies asigned to each cell on division. Either a single deterministic value or the params for "fraction_rnd"	0.5 = perfect half
p / p_params	real in [0.0, 1.0] or array of real	prob.	Parameter of the binomial distribution used when the PartitionSystem is inactive. Either a single deterministic value or the params for "p_rnd"	0.5
vol_scaling	Boolean	_	If true, the number of copies received by each daughter cell is affected by the potential assymetry in the volume split: more volume = more copies.	true
fraction_marker	string	_	User-given name to access the value of a stochastic "fraction" param	—
p_marker	string	_	User-given name to access the value of a stochastic "p" param	—

3.4.1 Mutation

Keywords: `mutation`
Recommended prefixes: `mut_`
State: occurrence count
Linked to (direct): `Plasmid`
Linked to (reverse): `Mutation Process` (extension), logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `Mutation` element signifies an interconversion involving `Plasmids`, encompassing both actual mutations and various reactions related to `Plasmids`. Refer to Figure 3.8 and Table 3.8 for a comprehensive list of fields pertaining to the `Mutation` element.

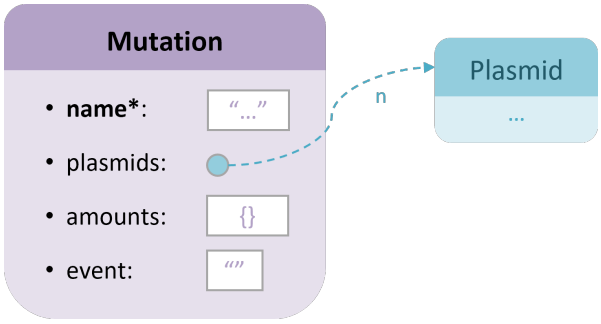

Figure 3.8: `Mutation` **element**

Connections and state The `Mutation` element establishes a direct connection with `Plasmids` (**??**) to delineate the stochastic nature of the associated reaction. Additionally, `Mutations` are assigned to `Mutation Processes` (Subsection 3.4.2) to incorporate dynamic information.

The `Mutation` is an abstract biological element, representing a DNA modification. It possesses both a binary state, denoting whether the modification has occurred (1) or not (0) within a cell, and a discrete quantitative state reflecting the count of occurrences throughout the cell's life span. The choice between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for `BGates` (**??**); or as a continuous input for logic elements (`QGates`, **??**), `Functions`, `Odes`, and

`Historics` (Section 2.2). The status of the `Mutation` can be reported through `Cell Colours` (??), `Population Stats`, `Out Files`, and `Plots` (??).

Mutation events The occurrences of a particular `Mutation` can be monitored using the `event` field. The user-assigned name is subsequently utilized to retrieve the number of events in the last time step. This value can be employed to generate a response to the events (e.g., a metabolic cost) or simply to report, log, and/or count them, utilizing `Cell Colours` (Subsection 2.4.1) and `Population Stats` (??).

Involved `Plasmids` This element solely delineates a reaction involving `Plasmid` in terms of stoichiometry. The kinetics are managed by the `Mutation Process` element (Subsection 3.4.2). The field `plasmids` comprises a list of `Plasmids`, utilized to specify which `Plasmids` are implicated in the reaction. They may be `BPlasmids` (Subsection 3.6.1) or `QPlasmids` (Subsection 3.6.2); combining them within the same `Mutation` is acceptable.

The `amounts` field is a list of integer numbers describing the stoichiometry of the reaction, indicating how many copies of each `Plasmid` are created or removed. Positive numbers denote creation, while negative numbers denote removal. An amount of 0 has no effect, and its associated `Plasmid` is disregarded. The order of amounts corresponds to the order of `plasmids`. If `amounts` is shorter than `plasmids`, it is padded with zeros (ignoring the rightmost `Plasmids`). If `amounts` is longer than `plasmids`, the rightmost amounts are ignored. For `BPlasmids`, which only admit amounts in -1, 1, any positive amount is interpreted as 1, and any negative one is replaced by -1.

In Example 3.61, various `Mutations` are illustrated. `"mut_Ain"` initiates the appearance of a plasmid, potentially simulating the arrival of a phage to a cell or transformation with foreign DNA. `"mut_Aout"` leads to the destruction of a plasmid copy, simulating the effect of digestive enzymes. `"mut_AtoB"` facilitates the conversion of one plasmid copy into another, resembling a mutation or gene editing. `"mut_homodimer"` combines two copies of a plasmid into one of another plasmid, simulating dimerisation. `"mut_hetdimer"` merges two different plasmids into one. `"mut_eclipse"`, while not causing long-term changes, destroys and produces one copy of the same plasmid, allowing for an eclipse period to be introduced (see how at Subsection 3.4.2). `"mut_repli"` removes one copy of a plasmid to produce two copies, potentially simulating replication (there is a dedicated built-in method for this purpose, Subsection 3.3.1). The distinction from `"mut_Ain"` lies in the ability of `"mut_repli"` to induce an eclipse period in both copies.

Example 3.38: Examples of `Mutation` reactions

```
qplasmid([ name := "qp_A" ]);
```

```
qplasmid([ name := "qp_B" ]);
qplasmid([ name := "qp_C" ]);

mutation([ name := "mut_Ain", plasmids := {"qp_A"}, amounts
    := {1} ]);
mutation([ name := "mut_Aout", plasmids := {"qp_A"},
   amounts := {-1} ]);
mutation([ name := "mut_AtoB", plasmids := {"qp_A", "qp_B"
   }, amounts := {-1,1} ]);
mutation([ name := "mut_homodimer", plasmids := {"qp_A", "
   qp_C"}, amounts := {-2,1} ]);
mutation([ name := "mut_hetdimer", plasmids := {"qp_A", "
   qp_B", "qp_C"}, amounts := {-1,-1,1} ]);
mutation([ name := "mut_eclipse", plasmids := {"qp_A", "
   qp_A"}, amounts := {-1,1} ]);
mutation([ name := "mut_repli", plasmids := {"qp_A", "qp_A"
   }, amounts := {-1,2} ]);
```

Table 3.8: Parameters of the Mutation element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
plasmids	array of Plasmid	ref.	Involved Plasmids, either created or removed	{}
amounts	array of integer	copies	Change in the amount of copies of each of the "plasmids", created if positive and removed if negative. Automatically completed with 0.0 to match the length of "plasmids".In the case of arbitrary real numbers assigned to Bplasmids, they are converted to 1 or -1 depending on their sign.	{}
event / event_marker	string	—	Exposes the number of mutation events during the last time step	—

3.4.2 Mutation Process

Keywords: `mutation_process`
Recommended prefixes: `mutp_`
State: occurrence count
Linked to (direct): `Mutation` (extends it), `Gate` or equivalent if conditional, `Randomness` or equivalent if stochastic rate, `Function` or equivalent if dynamic rate
Linked to (reverse): logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `Mutation Process` element signifies a process that induces alterations in the presence and/or quantities of `Plasmids`, encompassing mutations, recombination, or gene editing. A comprehensive list of fields for the `Mutation Process` element can be found in Figure 3.9 and Table 3.9.

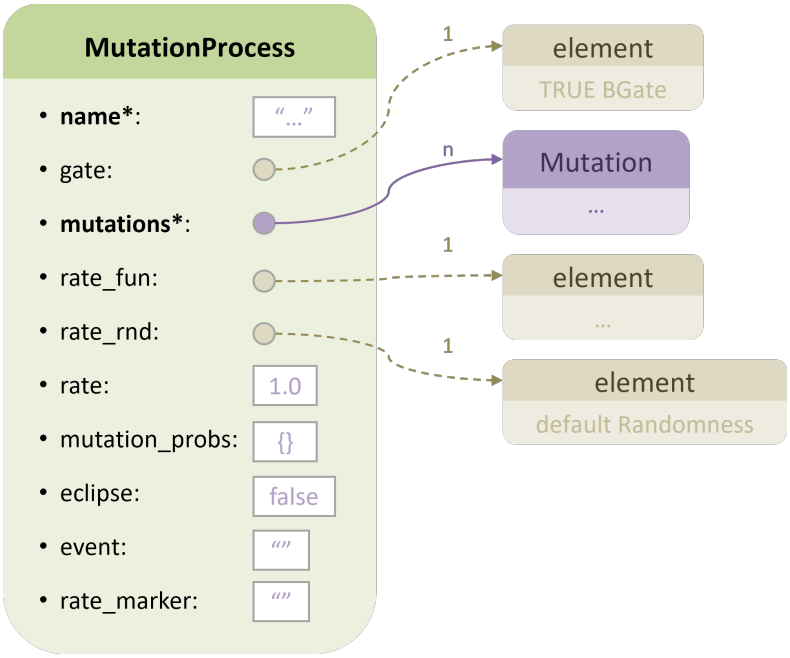

Figure 3.9: `Mutation Process` **element**

Connections and state The `Mutation Process` element establishes a direct connection with `Mutations` (**??**), which delineate the stoichiometry of the reaction that the `Mutation Process` augments with kinetics. Moreover, `Gates` (**??**) or other elements with digital states can serve as conditions for the activity of the `Mutation Process`. `Randomnesses` (Subsection 2.1.2) or equivalent elements can be linked to introduce stochasticity in the rate of occurrence of the reaction, and any quantitative element, particularly `Functions` (Subsection 2.2.1), can be assigned for a dynamic rate.

The `Mutation Process` is a biological process, representing a DNA reaction. It possesses both a binary state, denoting whether the reaction has occurred (1) or not (0) within a cell, and a discrete quantitative state reflecting the count of occurrences throughout the cell's life span. The choice between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for `BGates` (**??**); or as a continuous input for logic elements (`QGates`, **??**), `Functions`, `Odes`, and `Historics` (Section 2.2). The status of the `Mutation Process` can be reported through `Cell Colours` (**??**), `Population Stats`, `Out Files`, and `Plots` (**??**).

Kinetics This element extends the `Mutation` with kinetic properties. As `Plasmids` are expected to be present in low concentrations, a stochastic simulation molecule by molecule is more suitable than differential equations. `Mutation Processes` are modelled as Poisson processes and the state update is done by the Gillespie algorithm, considering all the `Mutation Processes` and replications (`OriVs`, Subsection 3.3.1) active in a cell.

`Mutation` events The reaction events produced by a specific `Mutation Process` can be tracked using the `event` field. The user-given name is then used to access the number of events in the last time step. This value can be used to generate a response to the events (e.g. a metabolic cost) or just to report, log, and/or count them, using `Cell Colour` (Subsection 2.4.1) and `Population Stats` (**??**).

Conditional behaviour A `Mutation Process`, like any cellular process, can be made conditional by assigning a `Gate` (**??**) or equivalent to its `gate` field. The process will only be active and eligible as the next event while the `Gate` evaluates to true. By default, a `TRUE BGate` is assigned, rendering the reaction always active.

Rate In the simple case that the process has a constant rate, it can be assigned at the `rate` parameter, given in reactions/min. This rate can be stochastic using the inheritable randomness system (Subsection 2.1.2) by assigning an array of values, interpreted by default as the mean and standard deviation of a normal distribution. A different distribution family or custom re-sampling schedule can be set at the `rate_rnd`

field using a custom `Randomness` (Subsection 2.1.2). To access the current value of a stochastic rate parameter, use `rate_marker`.

The rate may depend on the state or concentration of some elements, i.e. be dynamic. In this case, a custom `Function` (Subsection 2.2.1) or other quantitative element can be assigned to `rate_fun`. If a custom `Function` is assigned, the `rate` and `rate_rnd` fields are ignored.

Example 3.39 illustrates the utilization of the conditional nature and the rate aspect of the `Mutation Process` element. In this scenario, the `Mutation` transforms one instance of a `QPlasmid`, denoted as `"qp_A"`, into another variant, `"qp_B"`. Two distinct `Mutation Processes`, `"mutp_AtoB1"` and `"mutp_AtoB2"`, are introduced, each differing in their rates. `"mutp_AtoB1"` features a custom rate contingent upon the concentration of `"qp_A"`, whereas `"mutp_AtoB2"` possesses a fixed rate. The `Gates` employed as conditions ensure that both reactions cannot be simultaneously active.

Example 3.39: Conditional `Mutation Processes` with fixed and custom rates

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);
mutation([ name := "mut_AtoB", plasmids := {"qp_A", "qp_B"
   }, amounts := {-1,1} ]);

qgate([ name := "qga_less", input := "qp_A" , operator := "
   <", value := 100.0 ]);
bgate([ name := "bga_more", input := {"-qga_A"} ]);

function([ name := "fun_A"
  , input := {"qp_A"}
  , type := "sum"
  , params := {0.1}
  , auto_vol := "division"
]);

mutation_process([ name := "mutp_AtoB1"
  , gate := "qga_less"
  , rate_fun := "fun_A"
  , mutations := {"mut_AtoB"}
]);

mutation_process([ name := "mutp_AtoB2"
  , gate := "bga_more"
  , mutations := {"mut_AtoB"}
```

```
  , rate := 2.0
]);
```

Replication-linked Mutation Processes Instead of as a free reaction, the `Mutation Process` can be utilized as a replication-linked mutation, specifically for `QPlasmids`. This association is established at the `QPlasmid` level (refer to Subsection 3.6.2). In this context, the rate is reinterpreted as the probability of occurrence during replication.

Alternative `Mutations` The `mutations` field specifies the `Mutation` (Subsection 3.4.1) to extend and is typically assigned a single `Mutation`. However, it is a list to accommodate the possibility of assigning multiple `Mutations` as well. It is crucial to provide at least one `Mutation`. When multiple `mutations` are present, the `Mutation Process` randomly selects one during execution. By default, each `Mutation` has an equal probability of being chosen. If custom probabilities are desired, they can be assigned at the `mutation_probs` field. These probabilities correspond to the `mutations` list in order. If the probabilities do not sum to 1, they are scaled accordingly. If fewer probabilities are given than `Mutations`, the rightmost `Mutations` are assigned a probability of 0.0 and disregarded. Any extra probabilities are ignored.

Example 3.40 illustrates the utilization of a single `Mutation Process` to achieve two effects instead of employing two distinct `Mutation Process` elements. In this scenario, one of the `Mutations` converts a single copy of a `QPlasmid`, designated as `"qp_A"`, into `"qp_B"`, while the other results in the creation of `"qp_C"`. During the occurrence of the `Mutation Process` event, either of the two conversions is selected, with the second one being three times more probable than the first.

Example 3.40: `Mutation Processes` with alternative `Mutations`

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);
qplasmid([ name := "qp_C" ]);
mutation([ name := "mut_AtoB", plasmids := {"qp_A", "qp_B"
   }, amounts := {-1,1} ]);
mutation([ name := "mut_AtoC", plasmids := {"qp_A", "qp_C"
   }, amounts := {-1,1} ]);

mutation_process([ name := "mutp_AtoBorC"
  , mutations := {"mut_AtoB", "mut_AtoC"}
  , rate := 2.0
  , mutation_probs := {0.25, 0.75}
```

```
]);
```

Optional eclipse period The `eclipse` parameter dictates whether the involved `Plasmids` enter an eclipse period following the execution of the selected `Mutation`. By default, this parameter is set to false, indicating no eclipse period. It is important to note that regardless of the `eclipse` parameter, `QPlasmid` copies currently undergoing an eclipse period due to other reactions (or replication) are disregarded (i.e. not illegible for applying the `Mutation`). If set to true, this feature exclusively affects `QPlasmids`, as only they possess the eclipse capability. The duration eclipse period is specified `QPlasmid` level (see Subsection 3.6.2).

Simulation of phages To simulate phages and hybrid vectors like phagemids, a combination of a `Mutation Process` and a `Flux` element (Subsection 3.2.1) can be employed. The example in Example 3.41 illustrates the simulation of a phage with two states: extracellular (`"s_phage"`) and intracellular (`"bp_phage"`). The transition from `"s_phage"` to `"bp_phage"` is achieved through the coupling of a `Flux` and a `Mutation Process`. The `"flux_phage_in"` element senses and absorbs `"s_phage"` when the concentration of phage in the medium exceeds 10. This flux is active under such conditions. The `"mutp_phage"` process becomes active when `"flux_phage_in"` is active, and the intracellular phage is not yet present. Additionally, its rate is contingent on the absorbed phage amount. Upon occurrence, `"bp_phage"` is created inside the cell.

This `BPlasmid` producing `Molecule` (`"mol_capside"`) represents the moment when the phage has replicated sufficiently and is ready to release copies into the medium. Assuming a lytic phage behaviour, causing cell death and lysis, the `"flux_lysis"` element induces cell death by reducing the growth rate below the default threshold (refer to Subsection 3.6.3). Simultaneously, `"flux_phage_out"` initiates the emission of `"s_phage"` upon cell lysis.

Example 3.41: Phage with a `Mutation Processes` and `Fluxes`

```
molecule([ name := "mol_capside", times := { -50.0, 30.0 }
   ]);
operon([ name := "op_phage", output := {"mol_capside"} ]);
bplasmid([ name := "bp_phage", operons := {"op_phage"} ]);
signal([ name := "s_phage", colour := _magenta ]);

flux([ name := "flux_phage_in"
  , signal := "s_phage"
  , amount := -1.0
```

```
  , threshold := 10.0
]);

bgate([ name := "bga_phageIn", input := {"flux_phage_in", "
   -bp_phage"} ]);
mutation([ name := "mut_phage", plasmids := {"bp_phage"},
   amounts := {1} ]);

function([ name := "fun_phageIn"
  , input := {"flux_phage_in"}
  , type := "sum"
  , params := {0.1}
]);

mutation_process([ name := "mutp_phage"
  , gate := "bga_phageIn"
  , rate_fun := "fun_phageIn"
  , mutations := {"mut_phage"}
]);

flux([ name := "flux_lysis"
  , gate := "mol_capside"
  , amount := -1.0
]);

flux([ name := "flux_phage_out"
  , signal := "s_phage"
  , mode := "lysis"
  , amount := 1.0
]);
```

3.5 Conjugation

This section encompasses elements dedicated to the intercellular transmission of `Plasmids` through direct physical contact. This process applies to both `BPlasmids` (Subsection 3.6.1) and `QPlasmids` (Subsection 3.6.2). Two primary elements facilitate this transmission: `Pilus` (Subsection 3.5.1), representing a transmission system akin to the type IV secretion system (T4SS); and `OriT` (Subsection 3.5.2), DNA sequences essential

Table 3.9: Parameters of the MutationProcess element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
gate	any cellular element, usually Gate	ref.	Condition for the PartitionSystem sequence to be active. If active, the split is exact. If inactive, the split is random following a binomial distribution.	"_ga_true" = default TRUE Gate
mutations*	array of Mutation	ref.	Alternative Mutations that can be seelcted when a mutation event occur	{}
rate_fun	any cellular element, usually Function	ref.	Custom mutation rate, lambda. If assigned, "rate" and "rate_rnd" are ignored.	_
rate_rnd	any cellular element, usually Random-ness	ref.	Rules the stochasticity in the "rate" param	default normal Randomness
rate / rate_params	positive real or array of real	events/ min	Reaction rate or lambda. Either a single deterministic value or the params for "rate_rnd"	1.0
mutation_probs	array of positive real	prob.	Probabilities of selecting each of the alternative "mutations" when a mutation event occurs	{} = uniformly selected completed with 0.0 if shorter than "mutations"
eclipse / has_eclipse	Boolean	_	If true, the plasmid copies involved in the mutation event undergo an eclipse period. Only appplied to QPlasmids.	false
event / event_marker	string	_	Exposes the number of mutation events produced by this process during the last time step	_
rate_marker	string	_	User-given name to access the value of a stochastic "rate" param	_

for transmitted Plasmids, establishing specificity between these two elements.

3.5.1 Pilus

Keywords: pilus
Recommended prefixes: pil_
State: activity (digital)
Linked to (direct): Gate or equivalent if conditional, Randomness or equivalent if stochastic rate, Function or equivalent if dynamic rate
Linked to (reverse): OriT (specificity), logic elements (Historic, BGate, Cell Colour), condition for any cellular element, and global reporting elements (Population Stat, Out File, Plot)

Description The Pilus element, despite its name, does not denote an individual pilus but instead represents a transmission system, such as a T4SS, utilized in bacterial conjugation. This system may consist of multiple pili. For detailed information on the fields associated with the Pilus element, refer to Figure 3.10 and Table 3.10.

Connections and state The Pilus element establishes a direct connection with Gates (??) or other elements with digital states to serve as conditions for the conjugation to take place (both at the donor and recipient side). Randomnesses (Subsection 2.1.2) or equivalent elements can be linked to introduce stochasticity in the conjugation rate and/or neighbour selection process, and any quantitative element, particularly Functions (Subsection 2.2.1), can be assigned for a dynamic conjugation rate and/or neighbour selection probabilities. The mapping between Pili and OriTs is done at the OriT elements (Subsection 3.5.2).

The Pilus is an abstract biological element, representing a functional part of a cell, the transmission system, rather than its detailed physical components. It only possesses a binary state, denoting whether the transmission system is active (1) or not (0), in other words, whether the cell is a valid donor using that system (1) or not (0). It is active when the user-defined donor's condition is met and there is at least one compatible Plasmid in the cell to transmit (a BPlasmid (Subsection 3.6.1) or functional QPlasmid (Subsection 3.6.2) copy carrying an active compatible OriT). This digital state serves as a Boolean condition for conditional elements and input for BGates (??) and can be saved inHistorics (Section 2.2). The status of the Pilus can be reported through Cell Colours (??), Population Stats, Out Files, and Plots (??).

Conditional conjugation The Pilus doesn't depict a tangible element but rather a functional aspect. Unlike Molecules (Subsection 3.1.1) or Plasmids (??), it doesn't

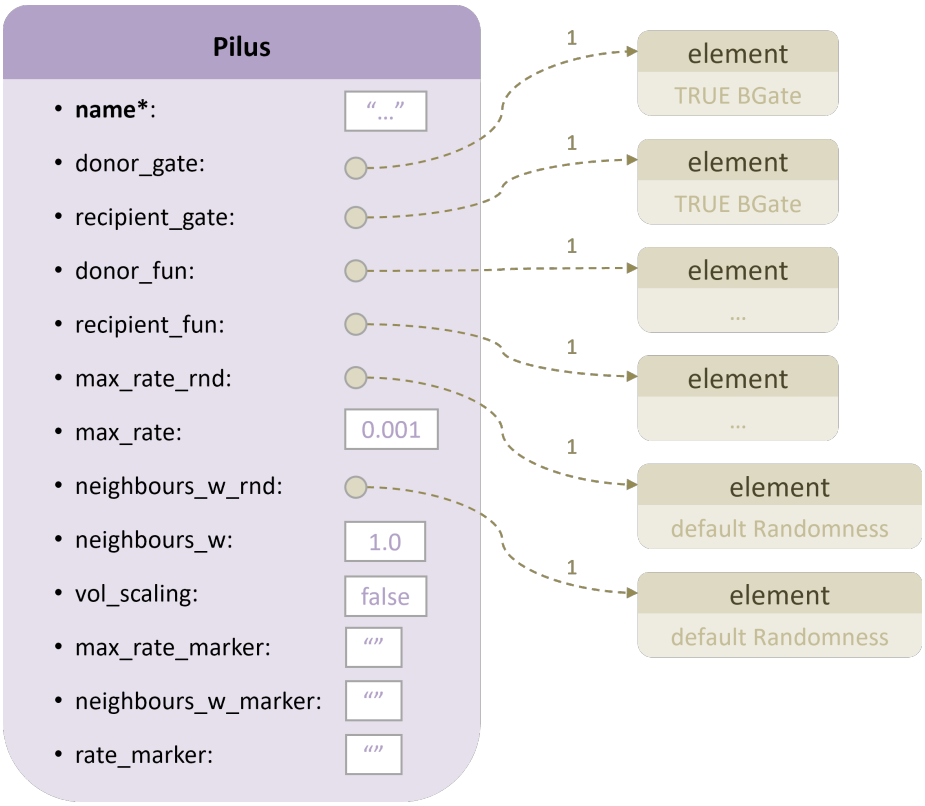

Figure 3.10: `Pilus` **element**

exist in a present or absent state but in an active or inactive state. Cells with an active `Pilus` can facilitate the transmission of compatible `Plasmids` through conjugation.

The conditions take two distinct forms. These are expressed as `Gates` (**??**) or other elements with binary state. The `donor_gate` sets the criteria a cell must meet to function as a donor, allowing it to transmit plasmids to other cells. This may involve the presence of proteins responsible for creating the transmission system or pili. Conditions specific to `OriT` (such as relaxases) are more appropriately specified at the `OriT` level (see Subsection 3.5.2). The presence of compatible `Plasmids` is implicit, as cells can only transmit the `Plasmids` they possess.

On the other hand, the `recipient_gate` defines the criteria for a cell to be a valid recipient candidate. This condition indicates that the transmission system can attach to the surface of the recipient cell, initiating a conjugation event. Despite meeting this condition, the conjugation may still be unsuccessful due to downstream incompatibilities, which are `OriT`-specific and detailed at the `OriT` level (refer to Subsection 3.5.2). A cell with a `Pilus`' `recipient_gate` that evaluates to false exhibits surface exclusion or belongs to an incompatible species/strain. Conversely, a cell with an `OriT`'s `recipient_gate` that evaluates to false shows proper entry exclusion. The implications differ: cells with surface exclusion are ignored by donors, while those with proper entry exclusion engage in attempted but unsuccessful conjugation, diminishing the observed conjugation rate.

Example 3.42 illustrates the imposition of conditions for conjugation from both the donor and recipient sides. In this scenario, only the original donors can function as donors, as transconjugant cells do not produce the transmission system. This condition is enforced through `"bga_donor"`, which takes the donor `Cell Type` (see Subsection 3.6.4) as input. On the recipient side, only cells of the `"bga_compatible"` `Cell Type` are eligible to act as recipients. This condition is implemented using the `"bga_compatible"` field.

Example 3.42: Conditional `Pilus` element

```
cell_type([ name := "cell_donor", plasmids := {"bp_A"} ]);
cell_type([ name := "cell_compatible" ]);
cell_type([ name := "cell_incompatible" ]);

pilus([ name := "pil_A"
  , donor_gate := "cell_donor"
  , recipient_gate := "cell_compatible"
]);

orit([ name := "ot_A", pilus := "pil_A" ]);
```

```
bplasmid([ name := "bp_A", orits := {"ot_A"} ]);
```

Example 3.43 demonstrates the implementation of basic surface exclusion to prevent cells already possessing the `Plasmid` from being considered potential recipients. This behaviour is not activated by default, so it must be explicitly defined.

Example 3.43: Surface exclusion to avoid repeated conjugations

```
bgate([ name := "bga_donor", input := {"-bp_A"} ]);

pilus([ name := "pil_A"
  , donor_gate := "bga_donor"
]);

orit([ name := "ot_A", pilus := "pil_A" ]);
bplasmid([ name := "bp_A", orits := {"ot_A"} ]);
```

The conjugation rate The conjugation rate is defined by the `max_rate` parameter, which can be either a deterministic value or stochastic. In the latter case, an array with scaling parameters for the probability distribution must be supplied. The default distribution is normal, so that the parameters are interpreted as the mean and standard deviation. This can be modified by assigning a custom `Randomness` (Subsection 2.1.2) to `max_rate_rnd`. The default value is a deterministic rate of 0.001 conjugations/min. To access the current value of a stochastic rate parameter, utilize `max_rate_marker`.

A custom `Function` (Subsection 2.2.1) or other quantitative element can be assigned to `donor_fun` to employ a dynamic conjugation rate that varies with this element's state. In this scenario, `max_rate` and `max_rate_rnd` are overridden, and the rate is determined by the output of `donor_fun` instead. Stochasticity must then be added at `Function` level.

The `vol_scaling` option is employed to automatically adjust the conjugation rate based on the cellular volume of the donor. It accounts for the observation that conjugation is more likely during the final stages of the cell cycle when donor cells are larger, potentially having a higher number of conjugation pili. By default, the assumption is that the rate provided by the user is the rate per unit of volume or conjugations per minute per μm^3. This option is disabled by default.

In Example 3.44, various methods for defining the conjugation rate from the donor's perspective are illustrated. `"pil_const"` employs a constant deterministic rate of 0.02 events per μm^3 and minute. `"pil_rnd"` utilizes a stochastic rate sampled from a normal

distribution with a mean of 0.001 and a standard deviation of 0.0001. It also features a custom `Randomness` element with reduced inertia. `"pil_custom"` employs a custom `Function` to describe a conjugation rate that varies with the volume of the donor. The rate is calculated as $0.0005 \cdot \text{vol}^2$. `vol_scaling` is kept as false as the cellular volume is already incorporated into the `Function`.

Example 3.44: Simple and custom conjugation rates at `Pilus` element

```
pilus([ name := "pil_const", max_rate := 0.02 ]);

randomness([ name := "rnd_A", inertia := 0.8 ]);
pilus([ name := "pil_rnd"
  , max_rate_rnd := "rnd_A"
  , max_rate := { 0.001, 0.0001 }
  , vol_scaling := true
]);

function([ name := "fun_vol"
  , input := {"_qm_vol"}
  , type := "product"
  , params := {2.0, 0.0005}
]);

pilus([ name := "pil_custom"
  , donor_fun := "fun_vol"
  , max_rate := { 0.0, 0.0001 }
]);
```

The effect of neighbours in the conjugation rate The previously specified rate represents the maximum achievable rate. The actual conjugation rate is influenced by the number of potential recipient cells, and it is adjusted accordingly. This adjustment is performed using a saturating exponential function, as detailed in the `Function` section (Subsection 2.2.1).

The shape of this function is determined by the `neighbours_w` parameter. A high value, approximately 4, results in a rate that saturates before reaching 1 neighbour, maintaining `max_rate` constant and independent of the number of neighbours. Conversely, very small values produce an almost linear dependence. For values under 0.1, the range from 0 to 4 neighbours exhibits an almost linear relationship. It's important to note that the rate at a neighbourhood size of 4 is not exactly `max_rate` but a lower value.

Users need to adjust `max_rate` accordingly. For `neighbours_w`, the rate at 4 neighbours is approximately one-third of `max_rate`. For intermediate values of `neighbours_w`, the relationship is saturating. The default value is 0.75, making the rate nearly saturate at 4 neighbours (approximately 0.95 times the `max_rate`), creating a behaviour similar to that of gro 2. In all cases, the rate is zero if there are no valid neighbours.

The `neighbours_w` parameter can be a single deterministic value or be stochastic. In the latter case, an array containing the scaling parameters of the probability distribution must be provided. The default distribution is normal, but it can be altered by assigning a custom `Randomness` (Subsection 2.1.2) to `neighbours_w_rnd`. To access the last value of a stochastic parameter, use `neighbours_w_marker`. Alternatively, a custom function can be employed through `neighbours_w_fun`. This function may be a deterministic or stochastic `Function` (Subsection 2.2.1), or any other element type. Its use supersedes `neighbours_w` and `neighbours_w_rnd`.

To access the current conjugation rate, considering both the maximum rate and the combined effect of neighbours, use `rate_marker`.

Neighbour selection The recipient cell is randomly selected from the pool of valid neighbours. The default behaviour implies that all neighbours are treated homogeneously, with each having the same probability of being selected. A different behaviour can be specified using the `recipient_fun` field. This field expects a custom `Function` to be evaluated at the potential recipient cell. It can include any kind of inputs, similar to a regular `Function`, with the addition of a special one: the `"_qm_neigh_distance"` marker, storing the distance to the donor. `Functions` containing `"_qm_neigh_distance"` should only be used in this specific context, as the marker retains the distance to the last potential donor and is updated solely when conjugation probabilities are being computed. The output of this `Function` serves as the relative weight of this neighbour. It is not necessary for it to be a probability or for all weights to add up to 1. When a custom `recipient_fun` is employed, it also affects the scaling of the `max_rate` by the neighbours. Instead of the number of neighbours, the sum of weights of the neighbours will be used.

Example 3.45 demonstrates how the number and type of neighbours influence the conjugation rate. In `"pil_simple"`, all neighbours have the same weight. The rate calculated at the donor is scaled by $1 - e^{-0.8 \cdot n}$ where n is the number of neighbours. For `"pil_custom"`, neighbours have different weights based on their growth rate. Cells that are growing faster are considered better recipients. The rate calculated at the donor is scaled by $1 - e^{-0.1 \cdot \sum 3 \cdot GR}$ along with some noise sampled from a normal distribution with a mean of 0.01 and a deviation of 0.001. $3 \cdot GR$ is also used as the weight of each potential recipient cell when selecting one of them.

Example 3.45: Effect of the number of neighbours in the Pilus element

```
pilus([ name := "pil_simple" , neighbours_w := 1.0 ]);

function([ name := "fun_gr"
  , input := {"_qm_gr"}
  , type := "sum"
  , params := {3.0}
]);

pilus([ name := "pil_custom"
  , recipient_fun := "fun_gr"
  , max_rate := { 0.01, 0.001 }
  , neighbours_w := 0.1
]);
```

3.5.2 OriT

Keywords: `orit` `ori_t` `oriT`
Recommended prefixes: `ot_`
State: presence (quantitative)
Linked to (direct): `Pilus` (specificity), `Randomness` or equivalent if stochastic, `Function` or equivalent if custom probability
Linked to (reverse): `QPlasmid` (container), logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description The `OriT` is the DNA component that facilitates the transmission of `Plasmids` (Subsection 3.6.1, Subsection 3.6.2) through compatible `Pilus` elements (Subsection 3.5.1). The complete list of fields for the `OriT` element is provided in Figure 3.11 and Table 3.11.

Connections and state The `OriT` element is directly linked to `Pilus` elements (Subsection 3.5.1) to establish specificity, determining which `Pili` can transmit `Plasmids` carrying the specific `OriT`. Conditionally active `OriTs` are connected to `Gates` (??) and other binary elements to modulate their state. `OriTs` with stochastic selection probabilities utilize `Randomness` elements (Subsection 2.1.2) or equivalents to govern their behaviour. For those with dynamic selection probabilities dependent on the state of other elements, connections are made, particularly to `Functions` (Subsection 2.2.1).

Table 3.10: Parameters of the Pilus element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
donor_gate	any cellular element, usually Gate	ref.	Condition that must evaluate to true in the donor cell	"_ga_true" = default TRUE Gate
recipient_gate	any cellular element, usually Gate	ref.	Condition that must evaluate to true in the recipient cell. Used to simulate surface exclusion	"_ga_true" = default TRUE Gate
donor_fun	any cellular element, usually Function	ref.	Custom Function that determines the maximum rate of conjugation. Evaluated at the potential donor cell. Shadows "max_rate".	—
recipient_fun	any cellular element, usually Function	ref.	Function that determines the chance of selecting a cell as a recipient. Evaluated at the potential recipient cell. Custom unit of magnitude that should match that of "neighbours_w"	"" = uniform chance (all the cells have the same chance)
max_rate_rnd	any cellular element, usually Randomness	ref.	Randomness that rules the stochasticity in the maximum conjugation rate	default normal Randomness
max_rate / max_rate_params	positive real or array of real	events/min	Maximim conjugation rate (can be scaled by neighbours). Either a single deterministic value or params of "max_rate_rnd"	0.001
neighbours_w_rnd	any cellular element, usually Randomness	ref.	Randomness that rules the stochasticity in "neighbours_w"	default normal Randomness
neighbours_w / neighbours_w_params	positive real or array of real	cells-1	Used to scale the maximum rate depending on the potential recipients in a non-linear way. The higher the value, the sooner saturation is reached (less linear). Either a single deterministic value or params of "neighbours_w_rnd"	0.75
vol_scaling	Boolean	_	If true, the "donor_function" and/or "max_rate" is multiplied by the cellular volume of the potential donor.	true
max_rate_marker	string	_	User-given name to access the value of a stochastic "max_rate" param	—
neighbours_w_marker	string	_	User-given name to access the value of a stochastic "neighbours_w" param	—
rate_marker	string	_	User-given name to access the last conjugation rate	—

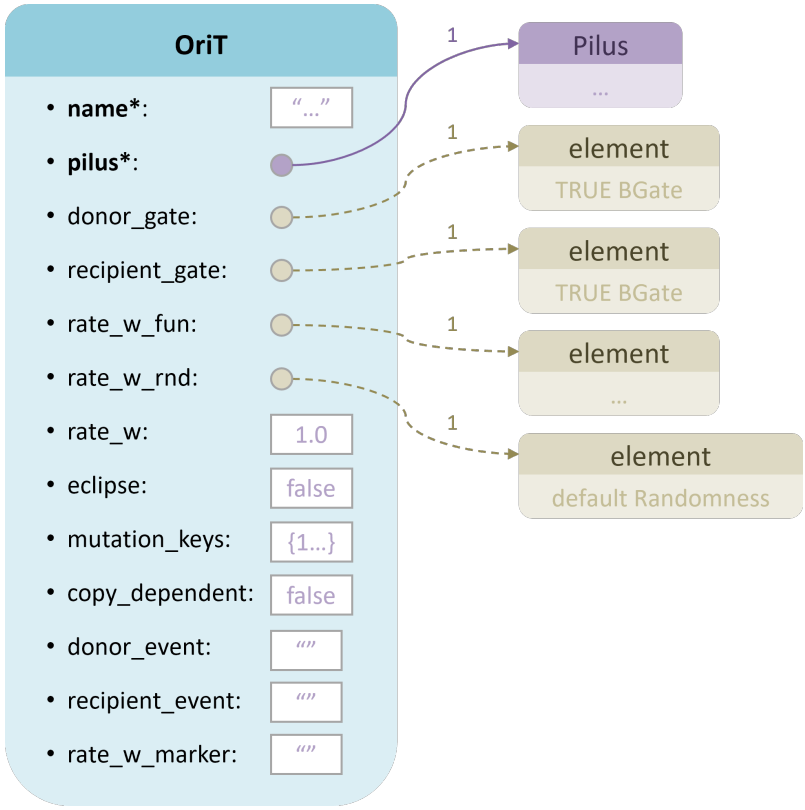

Figure 3.11: `OriT` **element**

Each `OriT` can be transmitted by a single `Pilus` (specified at the mandatory `pilus` field). Still, a `Pilus` can transmit an unlimited number of `OriTs`, and a `Plasmid` can carry an unrestricted number of `OriTs`. To enable a `Plasmid` to be transmitted by several `Pilus` elements, include multiple `OriTs` in it.

The `OriT` is a DNA component of both `BPlasmids` (Figure 3.12) and `QPlasmids` (Figure 3.13). As quantifiable entities, `OriTs` exhibit both a binary state, denoted by presence (1) or absence (0), and a continuous state reflecting their actual copy number. When situated within `QPlasmids` (**??**), the continuous state signifies the total copies of the `OriT` in a cell. Conversely, when employing `BPlasmids` (**??**), it represents the count of distinct `BPlasmids` harbouring the `OriT` within the cell. Both of these states can be accessed via the `OriT`'s name. The choice between these states is context-dependent, serving either as a Boolean condition for conditional elements and input for `BGates` (**??**); or as a continuous input for logic elements (`QGates`, **??**), `Functions`, `Odes`, and `Historics` (Section 2.2). The `OriT`'s status can be reported through `Cell Colours` (**??**), `Population Stats`, `Out Files`, and `Plots` (**??**).

Conditional transmission Analogously to the `Pilus` (see Subsection 3.5.1), the `OriT` has two conditions in the form of `Gates` (**??**) or equivalent binary elements. The `donor_gate` serves the same purpose as its counterpart in `Pilus`. When there is a single `OriT` per `Pilus`, the specific donor condition has no significant impact. However, the `recipient_gate` exhibits different behaviour compared to its homologue in `Pilus`, as it is used for proper entry exclusion, while that of `Pilus` is employed for surface exclusion.

Example 3.46 illustrates the distinction between surface and entry exclusion. Both `"cell_eex"` and `"cell_compatible"` are considered potential recipients, while `"cell_surfaceEx"` is excluded due to surface exclusion. However, the conjugation is deemed successful only when a neighbour of the `"cell_compatible"` `Cell Type` is chosen. If the selected neighbour belongs to the `"cell_eex"` `Cell Type`, the donor cell experiences a failed conjugation attempt.

Example 3.46: Entry exclusion vs surface exclusion

```
cell_type([ name := "cell_donor", plasmids := {"bp_A"} ]);
cell_type([ name := "cell_compatible" ]);
cell_type([ name := "cell_eex" ]);
cell_type([ name := "cell_surfaceEx" ]);

bgate([ name := "bga_goodSurface", type := "OR", input := {
   "cell_compatible", "cell_eex"} ]);

pilus([ name := "pil_A"
```

```
  , donor_gate := "cell_donor"
  , recipient_gate := "bga_goodSurface"
]);

orit([ name := "ot_A"
  , pilus := "pil_A"
  , recipient_gate := "cell_compatible"
]);

bplasmid([ name := "bp_A", orits := {"ot_A"} ]);
```

`OriT` competition In instances where distinct `Plasmids` capable of transmission through the same `Pilus` coexist within a single cell, a competitive interaction ensues. Within the context of a conjugation event, the random selection of one `Plasmid` occurs. By default, this selection process is uniform, providing each compatible `Plasmid` with an equal probability of being chosen.

Two parameters can modify this behaviour. The `rate_w` parameter facilitates the assignment of varying relative weights to different `OriTs`, thereby serving as a measure of their relative strength. Consequently, `Plasmids` bearing distinct `OriTs` possess differing probabilities of being selected for conjugation. Notably, these weights need not necessarily sum to 1, and all `Plasmids` carrying the same `OriT` share a common weight. To access the present value of a stochastic `rate_w` parameter, use `rate_w_marker`.

Example 3.47 illustrates two `OriTs` utilising the same transmission system or `Pilus` element. One of the `OriTs` possesses a custom strength (2), doubling that of the other (1, the default). Consequently, `Plasmids` carrying the `"ot_A"` will undergo transmission twice as frequently as those carrying `"ot_B"`. However, since there are two distinct `BPlasmids` sharing the `"ot_A"` designation, and only one with `"ot_B"`, each of the three `BPlasmids` holds an equal probability of being selected.

Example 3.47: Competition between `OriTs` sharing a `Pilus`

```
orit([ name := "ot_A"
  , pilus := "pil_common"
  , rate_w := 2.0
]);

orit([ name := "ot_B", pilus := "pil_common" ]);

bplasmid([ name := "bp_A1", orits := {"ot_A"} ]);
```

```
bplasmid([ name := "bp_A2", orits := {"ot_A"} ]);
bplasmid([ name := "bp_B", orits := {"ot_B"} ]);
```

The `copy_dependent` option, set to false by default, introduces a linear scaling effect on the probability of `Plasmid` selection, proportional to the copy number. The usage of `copy_dependent` is exclusive to `QPlasmids` (Subsection 3.6.2), specifically applicable when all `Plasmids` bearing the `OriT` are of the `QPlasmid` type and when this condition holds for all `OriTs` sharing the same `Pilus`. In cases where this criterion is not met, `copy_dependent` is automatically disabled. Generally, it is advisable to employ distinct `OriT` elements for `BPlasmids` and `QPlasmids`, and similarly for the `Pilus` elements. The default setting enables `copy_dependent` solely for `QPlasmid`-exclusive `OriTs`.

Eclipse period Similar to replications stemming from the `OriV`-based system (Subsection 3.3.1) and `Mutation Processes` (Subsection 3.4.2), the copies of the transmitted `Plasmid` may undergo an eclipse period subsequent to the conjugation event. This phenomenon impacts both copies, including the one transmitted and the one retained in the donor cell. This feature, exclusive to `QPlasmids`, is deactivated by default. If enabled though the `eclipse` field in scenarios where the `OriT` is present in both `BPlasmids` and `QPlasmids`, the eclipse period is selectively applied solely to the `QPlasmids`.

Conjugation event markers The creation of markers for conjugation events is facilitated by assigning names to the `donor_event` and/or `recipient_event` fields. These markers meticulously monitor the count of successful conjugation events within the last time step involving the specified `OriT`. The `donor_event` marker operates at the donor, while the `recipient_event` marker functions at the recipient. Their quantitative nature is instrumental in accurately tracking rare occurrences of multiple events within the same time step. These markers find application in various contexts, including the establishment of metabolic costs associated with conjugation events, implementing cooldown periods post-conjugation, or simply tallying the number of conjugation events. The assigned names will subsequently serve as identifiers for accessing these markers.

Example 3.48 exemplifies the utilization of the donor cell's event marker to create a cool-down effect. A minimum waiting time of 25 minutes between conjugation events is enforced, as indicated by the degradation time of `"mol_cooldown"`. Following a conjugation event, this `Molecule` (see Subsection 3.1.1) is generated in the donor cell. It is imperative that the synthesis time of the `Molecule` is smaller than the time step size.

Example 3.48: Donor cool-down using the event markers of the `OriT` element

```
molecule([ name := "mol_cooldown", times := { -25.0, 0.0 }
   ]);
```

```
operon([ name := "op_cooldown", gate := "ot_A_donor",
   output := {"mol_cooldown"} ]);

bgate([ name := "bga_donor", input := {"-mol_cooldown"} ]);

pilus([ name := "pil_A", donor_gate := "bga_donor" ]);

orit([ name := "ot_A"
  , pilus := "pil_A"
  , donor_event := "ot_A_donor"
]);

bplasmid([ name := "bp_A1", operons := {"op_cooldown"},
   orits := {"ot_A"} ]);
```

Example 3.49 demonstrates the application of the recipient event marker for tallying the number of successful conjugations. The `"stat_current"` variable monitors the count of conjugation events transpiring within the current time step, while `"stat_cumulated"` keeps a running total of conjugations throughout the entire simulation.

Example 3.49: Conjugation counting event markers of the `OriT` element

```
pilus([ name := "pil_A", recipient_event := "
   pil_A_recipient" ]);

pop_stat([ name := "stat_current"
  , input := "pil_A_recipient"
  , stats := "sum"
]);
```

3.6 Containers and tags

This section encompasses `Plasmids`, acting as containers for other DNA elements and facilitating their transfer between cells. `Boolean Plasmids` or `BPlasmids` (Subsection 3.6.1) adopt a digital representation (either present or absent state). `Quantitative Plasmids` or `QPlasmids` (Subsection 3.6.2), in contrast, are characterised by tracking an explicit quantitative copy number. They stand as the exclusive subtype capable of carrying the DNA components responsible for the quantitative dynamics outlined in

Table 3.11: Parameters of the OriT element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
pilus*	Pilus	ref.	Tansmission system employed by the OriT	
donor_gate	Gate	ref.	Condition that must evaluate to true in the donor cell	"_ga_true" = default TRUE Gate
recipient_gate	Gate	ref.	Condition that must evaluate to true in the recipient cell. Used to simulate true entry exclusion (surface exclusion is simulated with the "recipient_gate" of the "pilus"	"_ga_true" = default TRUE Gate
rate_w_fun	any cellular element, usually Function	ref.	Custom function for the rate weight. Shadows "rate_w".	_
rate_w_rnd	any cellular element, usually Function	ref.	Randomness provider for "rate_w"	default normal distribution
rate_w / rate_w_param	positive real or array of real	arbitrar (it is a relative magnitude)	Relative probability of selecting this OriT among all the active OriTs that the "pilus" can transmit. Either a single deterministic value or params for "rate_w_rnd"	1.0
eclipse / has_eclipse	Boolean	_	If true, the plasmid copies involved in the conjugation event undergo an eclipse period, both in the donor and in the recipient. Only appplied to QPlasmids.	false
mutation_keys / mutations_keys	array of positive integer	_	Keys of groups the replication mutations defined in the Plasmid that are possible from this OriT.	{1}
copy_dependent	Boolean	_	If true, the number of copies of the plasmid is considered in the probability of selecting this OriT among all the active OriTs that the "pilus" can transmit. Do not use if the OriT is included in BPlasmids.	false
donor_event	string	_	Exposes conjugation events that use this OriT in the donor cell for one time step after a succesful conjugation event.	_
recipient_event	string	_	Exposes conjugation events that use this OriT in the recipient cell for one time step after a succesful conjugation event.	_
rate_w_marke	string	_	User-given name to access the value of a stochastic "rate_w" param	_

Section 3.3.

Within this section, two additional elements serve as immutable tags for cell identification. `Strain` (Subsection 3.6.3)provides a description of the fundamental physical and growth properties of the cell. `Cell Type` (Subsection 3.6.4) extends the information provided by `Strain` incorporating an initial state concerning `Plasmids` and `Molecules` present at the creation of the cell.

3.6.1 Boolean Plasmid (BPlasmid)

Keywords: `bplasmid` `b_plasmid`
Recommended prefixes: `bp_` `p_`
State: presence (digital)
Linked to (direct): `Operon` and `OriT` (DNA components), `Mutation Process` (replication-linked mutations) `Randomness` or equivalent if stochastic loss probability, `Function` or equivalent if dynamic loss probability
Linked to (reverse): `Cell Type` (container), logic elements (`Function`, `Ode`, `Historic`, `BGate`, `QGate`, `Cell Colour`), condition or custom function for any cellular element, and global reporting elements (`Population Stat`, `Out File`, `Plot`)

Description

Description The `Boolean Plasmid` or `BPlasmid` is a `Plasmid` characterised by a digital representation, existing in either a present or absent state. This representation extends beyond bacterial plasmids and encompasses other DNA molecules, such as phages or chromosomes. A comprehensive list of fields associated with the `BPlasmid` element is presented in both Figure 3.12 and Table 3.12.

Connections and state The `BPlasmid` element establishes direct links with its compatible DNA components: `Operon` (Subsection 3.1.2) and `OriT` (Subsection 3.5.2). Additionally, `BPlasmids` can be endowed with `Mutation Processes` (Subsection 3.4.2), enabling the specification of mutations they may undergo during replication from an `OriT` following a conjugation event. In instances where `BPlasmids` exhibit stochastic loss probabilities upon cell division, the governing rules are dictated by `Randomness` elements (Subsection 2.1.2) or their equivalents. For `BPlasmids` featuring dynamic loss probabilities contingent on the state of other elements, connections are forged, notably to `Functions` (Subsection 2.2.1).

The `BPlasmid` represents a physical biological element with an exclusively digital state, denoted by either presence (1) or absence (0) within a given cell. Notably, it does not explicitly track copy numbers. This state is accessible via the `BPlasmid`'s

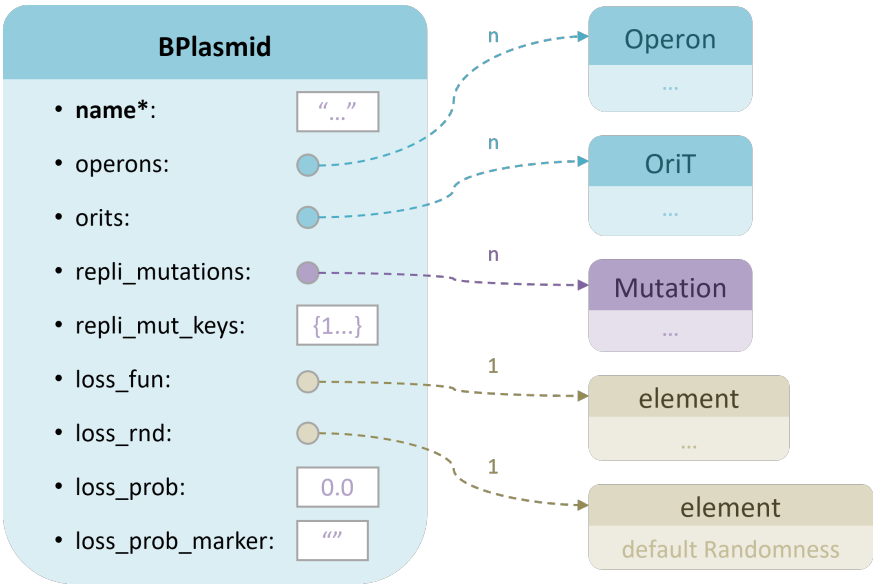

Figure 3.12: `BPlasmid` **element**

name, functioning as a Boolean condition for conditional elements and serving as input for `BGates` (??), or it can be stored by `Historics` (Subsection 2.2.3). The status of the `BPlasmid` can be conveyed through various reporting mechanisms such as `Cell Colours` (??), `Population Stats`, `Out Files`, and `Plots` (??).

DNA components Similar to any type of `Plasmid`, the `BPlasmid` serves as a container for placing DNA elements within cells. Specifically, the `BPlasmid` exclusively accepts `Operons` (Subsection 3.1.2) and `OriTs` (Subsection 3.5.2) as components. The `operons` field is designated for `Operons`, while the `orits` field accommodates `OriTs`. Both fields expect a list and are, by default, empty. In its default state, the `BPlasmid` does not possess any components. Functioning as the fundamental unit of DNA transmission, the `Plasmid` plays a pivotal role in both vertical and horizontal gene transfer.

Example 3.50 presents two instances of `BPlasmids`, each featuring `Operon` and `OriT` components. The `"mol_relA" Molecule` (Subsection 3.1.1) serves as a conjugation protein, crucial for `"ot_A"`. Specifically, `"bp_conjugative"` encompasses the `OriT` and an `Operon` responsible for producing `"mol_relA"`, facilitating its ability to undergo self-transmission. Conversely, `"bp_transmissible"` possesses the `OriT` but lacks the `Operon`, necessitating the presence of `"bp_conjugative"` in the same cell for transmission via conjugation.

Example 3.50: Conjugative and transmissible `BPlasmids`

```
pilus([ name := "pil_t4ts", max_rate := 0.01 ]);
molecule([ name := "mol_relA" ]);
orit([ name := "ot_A", pilus := "pil_t4ts", donor_gate := "
   mol_relA" ]);
operon([ name := "op_relA", output := {"mol_relA"} ]);

bplasmid([ name := "bp_conjugative"
  , operons := {"op_relA"}
  , orits := {"ot_A"}
]);

bplasmid([ name := "bp_transmisible"
  , orits := {"ot_A"}
]);
```

Boolean representation Choosing a `BPlasmid` to represent a plasmid means that the copy number is irrelevant, as a opposed to the `QPlasmid`. The `BPlasmid` has two states: present (true, 1) or absent (false, 0). This is equivalent to the plasmids from gro 2. When in the present state, the plasmid would actually be present at certain copy number, which is not recorded.

`Operons` and `OriTs` are quantitative elements however. If the same element is placed in several `BPlasmids`, each `BPlasmid` provides one copy. That should be taken into account when using copy number dependent features. The same element should not be placed in `BPlasmids` and in `QPlasmids` in the same simulation, specially when those copy number dependent features are used.

The state of the `Operon` in Example 3.51 will be 3 in cells where both `BPlasmids` are present, 2 if only `"op_B"` is present, and 1 if only `"op_A"` is present. This arises from the cumulative contribution of one copy from each `BPlasmid` carrying the respective `Operon`.

Example 3.51: Same `Operon` in two `BPlasmids`

```
operon([ name := "op_shared" ]);
bplasmid([ name := "bp_A", operons := {"op_shared"} ]);
bplasmid([ name := "bp_B", operons := {"op_shared", "
   op_shared"} ]);
```

Loss probability The user can assign a loss probability to the `loss_prob` parameter, representing the likelihood of one of the daughter cells receiving zero copies of the plasmid upon cell division. Typically calculated using a binomial distribution, this probability ensures that `Plasmid` loss does not occur in both daughter cells; at least one of them will invariably carry the `BPlasmid` post-division. The assigned probability can be either deterministic or stochastic. In the case of an array with two values, they are interpreted as the mean and standard deviation of a normal distribution by default. Customization of the default `Randomness` is feasible by overriding it with alternative distribution families and implementing custom re-sampling schedules at `loss_prob_rnd`. `loss_prob_marker` is used to access the present value of a stochastic parameter. Alternatively, the probability can be contingent on the states of other elements by assigning a `loss_prob_fun`, which supersedes `loss_prob` and `loss_prob_rnd`.

Replication mutations The `repli_mutations` and `repli_mut_keys` function analogously to those elucidated for `QPlasmids` at Subsection 3.6.2. However, in this context, these mutations can only occur during replication from an `OriT`, as `BPlasmids` lack `OriVs` and explicit replication directly from them.

3.6.2 QPlasmid

Keywords: `qplasmid` `q_plasmid`
Recommended prefixes: `qp_`
State: presence (quantitative)
Linked to (direct): `Operon`, `OriT`, `OriV`, `Copy Control`, `Partition System`, `Mutation Process`
Linked to (reverse): `Cell Type`

Description The `QPlasmid` is a `Plasmid` with quantitative representation i.e. explicit copy number. It may represent an actual bacterial plasmid or any other DNA molecule, like a phage or chromosome. The complete list of fields for the `QPlasmid` element is provided in Figure 3.13 and Table 3.13.

Components `QPlasmids` can contain `Operons` and `OriTs` as `BPlasmids` do. In addition, they can also contain elements related to intracellular plasmid dynamics: `OriV`, `Copy Control` and `Partition System`. The bahaviour of these elements was explained in their respective sections. A list of elements is expected in all the categories except the `Partition System`. In the current version only one `Partition System` can be assigned to each `QPlasmid`. By default, the lists are empty (no components) and the copies of

Table 3.12: Parameters of the BPlasmid element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
operons	Operon	ref.	Operons that the Plasmid includes	{} = no output
orits / oriTs	OriT	ref.	OriTs that the Plasmid includes	{} = no oriT
repli_mutations	array of Mutation-Pocess	ref.	Optional mutations that occur on plasmid replication from OriT (conjugation) with a chance given by the MutationProcess rate parameter.	{} = no mutations
repli_mut_key / repli_mutation / repli_mutation	array of positive integer	_	Arbitrary numerical keys used to select subsets of the MutationProcesses from OriTs	filled with 1s = all the OriTs can procude them
copy_num / copy_num_params / copy_num_dist	positive real, array of real, Distribution or dictionary	copies /um3	Typical copy number in concentration units. Used to initialize the number of plasmid copies in newly created cells. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution	1.0
loss_prob_fun	any cellular element, usually Function	ref.	Custom function for the loss probability. Shadows "loss_prob"	—
loss_prob_rnd	any cellular element, usually Random-ness	ref.	Randomness that rules the loss probability	default normal Randomness
loss_prob	real in [0,1] or array of real	_	Probability of the plasmid being lost on cell division. Either a single deterministic value or params for "loss_prob_rnd"	0.0 = never lost
loss_prob_marker	string	_	User-given name to access the value of a stochastic "loss_prob" param	—

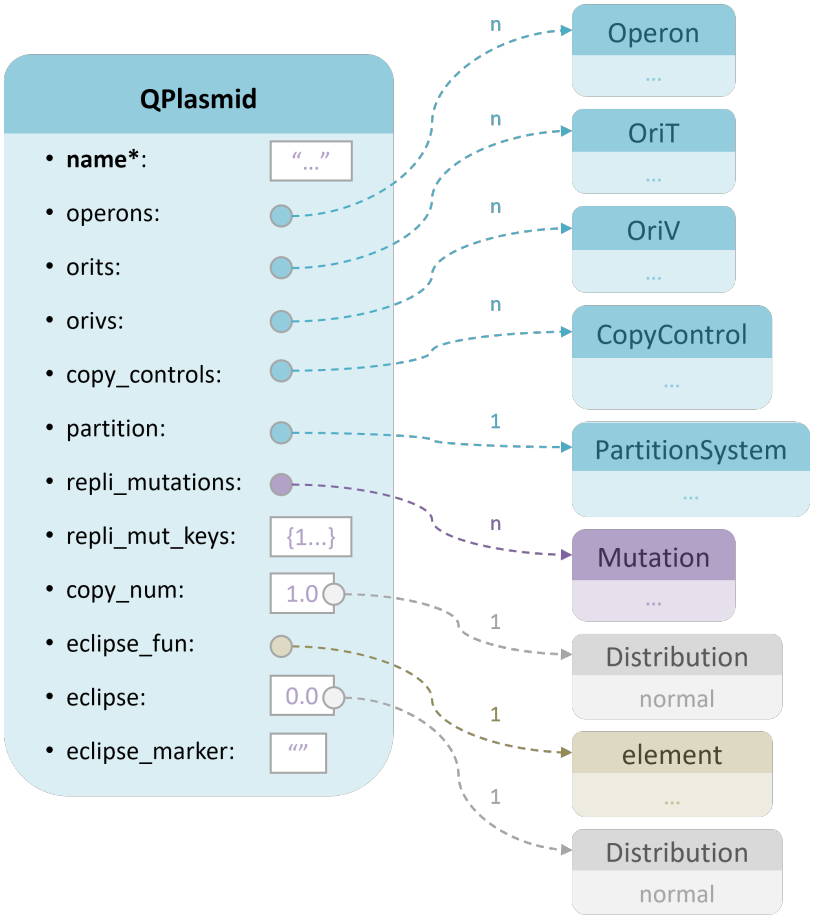

Figure 3.13: `QPlasmid` **element**

the QPlasmid are randomly distributed between the daughter cells (following a binomial distribution).

As all the components are quantitative, the copy number of the QPlasmid is transmitted to that of the components. In the case of a QPlasmid containing several copies of the same element, the copy number is multiplied accordingly. In Example 3.52, the number of copies of the OriV will be double that of the QPlasmid, as each copy of the QPlasmid bears to copies of the OriV.

Example 3.52: Dosage of the components of a QPlasmid

```
operon([ name := "op_A" ]);
oriv([ name := "ov_A" ]);
qplasmid([ name := "qp_A", operons := {"op_A"}, orivs := {"
   ov_A", "ov_A"} ]);
```

Eclipse period and marker The QPlasmids can optionally undergo an eclipse period after replication, typically after replication from an OriV, but it can optionally take place also after conjugation or Mutation Processes. This affects individual copies of the QPlasmid, not the whole element. During this period, the copy is ignored for all the biological processes: gene expression, replication, conjugation, mutations...The number of copies is effectively reduced by one and so that that copy is also not counted in OutFiles and Plots (only the active copies are considered). The eclipse can be used to set a deterministic duration in minutes or the parameters of a normal distribution in case a stochastic behaviour is desired. If a custom (non-normal) distribution is required, it can be assigned instead to the eclipse_dist field. This field expects a distribution dictionary or reference, being any distribution family a valid one. Notice that the inheritable randomness system (Randomness) is not used in this case. If a distribution is specified, the potential value at eclipse is ignored. If none of the two fields is set, there is no eclipse period, as the default duration is of 0 minutes. Additionally, it is possible to make the eclipse period dependent on the state of other elements through the eclipse_fun field. A custom Function or any other element can be assigned. In this case, the value of eclipse_fun is added to that sampled from eclipse or eclipse_dist as the duration of the eclipse periods must be different from copy to copy.

The number of plasmid copies that are in the eclipse period can be exposed using eclipse_marker. This user-given name is then used to access the value from other elements.

Replication mutations It is possible to assign Mutation Processes that can occur when the Plasmid is replicated (from either an OriV or OriT). This is done via the

`repli_mutations` field. The rate of the MutationProbability is interpreted as the probability of the mutation occurring instead of the regular replication rate. If the rate if greater than 1.0, it is considered as 1.0. If the `Gate` associated to a `Mutation Process` evaluates to false, its probability is considered to be zero. The `Mutation Processes` that are assigned to at least one `QPlasmid` are not considered as spontaneous mutations any more i.e. they only occur in the context of replication. The field `repli_mut_keys` can be used to assign a numeric key to each of the `repli_mutations`. This keys are used at `OriVs` and `OriTs` to decide which are able to produce which mutations.

In Example 3.53 a `QPlasmid` (`"qp_A"`) can undergo a mutation when it is replicated. The newly created copy will be of the type `"qp_B"` instead of `"qp_A"` whenever the mutation occurs. The copy of `"qp_A"` that serves as scaffold is both removed and created in the `Mutation` element to make it undergo an eclipse period too. By default, `Mutation Processes` do not produce an eclipse period; that is why the `eclipse` option of `"mutp_AB"` is set to true explicitly. The constant rate of 0.5 acts as the chance of mutation on replication. As there is no custom `Gate` linked, the mutation is always active. `"qp_A"` has two `OriVs` but only the replications that started by `"ov_A2"` can produce mutations, as only this one matches the key of 2. `"ov_A1"` has the default key, 1. The replication of `"qp_B"` do not produce mutations as its `repli_mutations` field is empty.

Example 3.53: Replication mutations of a `QPlasmid`

```
operon([ name := "op_A" ]);
operon([ name := "op_B" ]);
oriv([ name := "ov_A1" ]);
oriv([ name := "ov_A2", mutation_keys := {2} ]);

mutation([ name := "mut_AB"
  , plasmids := {"qp_A", "qp_A", "qp_B"}
  , amounts := {-1, 1, 1}
]);

mutation_process([ name := "mutp_AB"
  , rate := 0.5
  , mutations := {"mut_AB"}
  , eclipse := true
]);

qplasmid([ name := "qp_A"
  , operons := {"op_A"}
  , orivs := {"ov_A1", "ov_A2"}
```

```
  , repli_mutations := {"mutp_AB"}
  , repli_mut_keys := {2}
  , eclipse := {4.0, 0.4}
]);

qplasmid([ name := "qp_B"
  , operons := {"op_B"}
  , orivs := {"ov_A1", "ov_A2"}
]);
```

Copy number The fields `copy_num` or `copy_num_dist` can be used to set a default copy number in concentration: copies per um3. `copy_num` expects a positive real number (as it is a concentration). This one will be deterministic. A custom distribution can be specified at `copy_num_dist` to get stochastic behaviour. `copy_num_dist` overrides `copy_num`. If none of the fields is set, the default copy number is 1.0. This copy number is only used as a fallback value for the initial copy number of newly created cells in case no other value is given by the user. It is never used during the simulation of plasmid dynamics and is not required if the user always provides the copy number at `Cell Type` level. Example 3.54 shows the different ways to describe the copy number.

Example 3.54: Default copy number of `QPlasmid`

```
qplasmid([ name := "qp_A", copy_num := 100.0 ]);
qplasmid([ name := "qp_B", copy_num := {100.0, 1.0} ]);
qplasmid([ name := "qp_C", copy_num_dist := [ type := "
  uniform", params := {90.0, 110.0} ] ]);
myDist := [ type := "uniform", params := {90.0, 110.0} ];
qplasmid([ name := "qp_D", copy_num_dist := "dist_copyNum"
  ]);
```

3.6.3 Strain

Keywords: `strain`
Recommended prefixes: `str_`
State: constant tag (digital)
Linked to (direct): _
Linked to (reverse): `Cell Type`

Table 3.13: Parameters of the QPlasmid element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
operons	Operon	ref.	Operons that the Plasmid includes	{} = no output
orits / oriTs	OriT	ref.	OriTs that the Plasmid includes	{} = no oriT
orivs / oriVs	OriV	ref.	OriVs that the Plasmid includes	{} = no oriV
copy_controls	CopyControl	ref.	Sequences fo copy number control that the Plasmid includes	{} = no copy control sequences
partition / partition_system	PartitionSystem	ref.	Optional sequence for the exact split on cell division	"" = random partition with p = 0.5
repli_mutations	array of MutationPocess	ref.	Optional mutations that occur on plasmid replication from OriVs or OriTs with a chance given by the MutationProcess rate parameter.	{} = no mutations
repli_mut_keys / repli_mutation_keys / repli_mutations_keys	array of positive integer	—	Arbitrary numerical keys used to select subsets of the MutationProcesses from Orivs and OriTs	filled with 1s = all the OriVs and OriTs can procude them
copy_num / copy_num_params / copy_num_dist	positive real, array of real, Distribution or dictionary	copies /um3	Typical copy number in concentration units. Used to initialize the number of plasmid copies in newly created cells. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution	1.0
eclipse_fun	any cellular element, usually Function	ref.	Custom function for the duration of the eclipse period. Added to "eclipse" (does not shadow it)	_
eclipse / eclipse_params / eclipese_dist	positive real, array of real, Distribution or dictionary	min	Duration of the eclipse time that follows plasmid replication, conjugation and mutation. During this period, the involded copies of the QPlasmid are ignored, thus reducing the effective copy number. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution	0.0 = no eclipse time
eclipse_marker	string	_	To expose and access the number of copies undergoing the eclipse period after replication	_

A `Strain` describes the basic properties of a cell, related to morphology, growth and division.

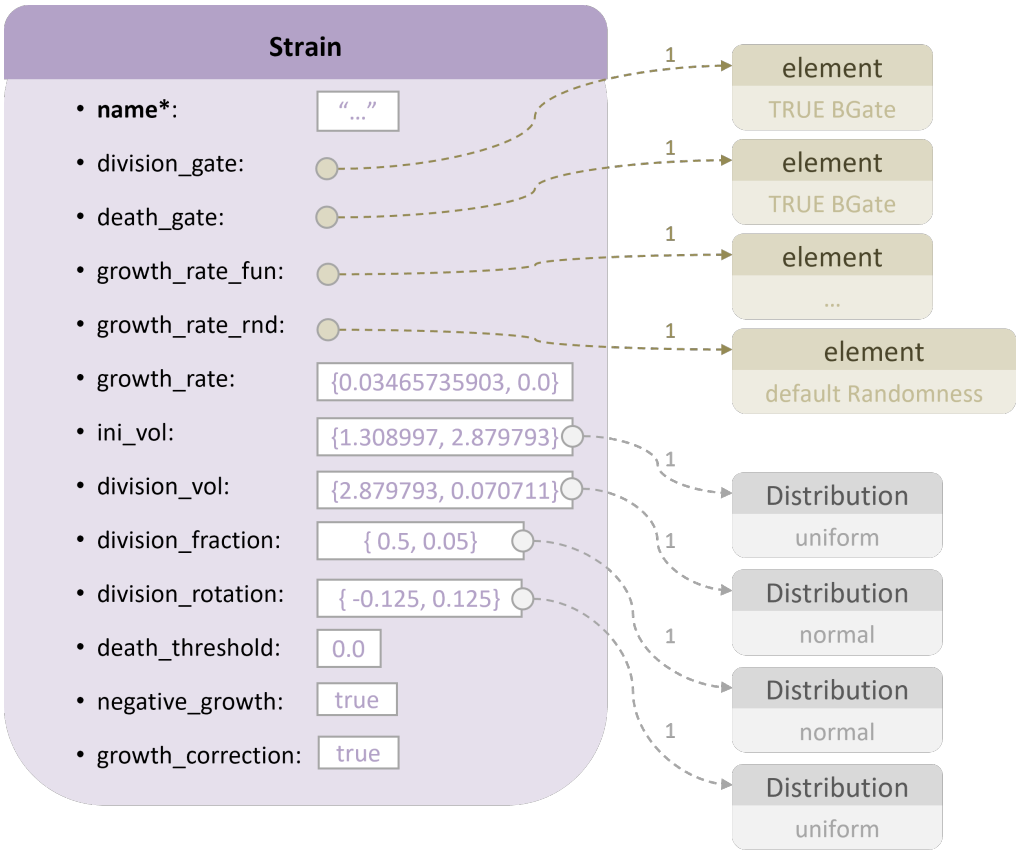

Figure 3.14: `Strain` **element**

Growth rate The growth rate can be described using the `growth_rate` field. It admits a single value for a deterministic behaviour or a list of real numbers to produce variability among the population. They are interpreted as the scaling parameters of a probability distribution. By default, they are interpreted as the mean and standard deviation of a normal distribution. The default value is the maximum theoretical growth rate of E. coli, deterministic. A distribution other than the normal can be used by setting a custom `Randomness` at `growth_rate_rnd`. If scaling parameters are provided but not a custom `Randomness`, a standard normal one is automatically created.

This growth rate acts as a base rate; if biomass `Fluxes` are active in the cell, they

may affect the total growth rate. The default behaviour is that all the biomass `Fluxes` (see `Flux` section at Subsection 3.2.1) are added to the `growth_rate`. Alternatively, a custom behaviour can be specified using `growth_rate_fun`. This field expects a custom `Function` of biomass `Fluxes`, but any other elements can be used without restriction. If provided, it shadows `growth_rate` and the default addition. The stochastic behaviour can be created at `Strain` level using `growth_rate` and `growth_rate_rnd` or at `growth_rate_fun` level, using a stochastic `Function`.

When performing calculations and including randomness, the resulting growth rates may be negative. The `negative_growth` option rules how that situation is handled. By default it is true, meaning that negative growth rates are allowed. That does not mean that the cells decrease in size. Due to the limitations of the physics engine, this is not possible. The cells keep their current size, but the negative value of the rate and total biomass flux is stored in the cellular state to be used as input for other elements. If `negative_growth` is set to false, negative growth rates and a negative total biomass fluxes are substituted by 0.0 in the state. This is only relevant in the case that those values are uses as input to other elements.

The effect of the discretization of the growth differential equation is that the observed growth rate is slightly smaller than the parameter value. This effect increases with bigger time step sizes. By default, it is compensated automatically. To disable the correction, set `growth_correction` to false. Notice that only the growth rate equation is corrected. Users should take into account the discretization error when defining custom differential equations.

Example 3.55 shows the default way to alter the growth rate, using biomass `Fluxes`. The `Strain` has a base growth rate that will be increased in the cells where `"mol_A"` and/or `"mol_B"` are present. They act as markers of the presence of certain nutrients within the cells. The value of the `Fluxes`, which is added to the growth rate, is the one defined by their `amount` parameter when their `gate` evaluates to true and 0 otherwise. The first `Flux` is deterministic while the second one is stochastic. The first one only modifies the mean of the total stochastic growth rate while the second one also modifies its deviation. The total growth rate will be maximal when both `Molecules` are present and minimal when none of then is present.

Example 3.55: Default growth rate calculation

```
strain([ name := "str_base", growth_rate := { 0.01, 0.001 }
    ]);

molecule([ name := "mol_A" ]);
molecule([ name := "mol_B" ]);
```

```
flux([ name := "flux_growthA", gate := "mol_A", amount := 0
   .005 ]);
flux([ name := "flux_growthB", gate := "mol_B", amount := {
   0.02, 0.001} ]);
```

Example 3.56 shows how to use custom `Function` for the growth rate, instead of just values. There are two `QPlasmids` that produce a metabolic cost. The costs is calculated by `"fun_cost"` as a function of their concentrations. `"str_1"` uses `"fun_cost"` directly as the custom growth rate `Function`. The value of `"fun_cost"` and the `growth_rate` distribution are added. `"str_2"` uses `"fun_cost"` indirectly through a `Flux`. Both are equivalent.

Example 3.56: Custom growth rate calculation

```
qplasmid([ name := "qp_A" ]);
qplasmid([ name := "qp_B" ]);

function([ name := "fun_cost"
  , input := {"qp_A", "qp_B"}
  , type := "sum"
  , params := {-0.001,-0.0015}
  , auto_vol := "division"
]);

flux([ name := "flux_", amount_fun := "fun_cost" ]);

strain([ name := "str_1"
  , growth_rate_fun := "fun_cost"
  , growth_rate := { 0.035, 0.003 }
]);

strain([ name := "str_2"
  , growth_rate := { 0.035, 0.003 }
]);
```

Example 3.57 shows how to implement Monod growing kinetics using a custom `Function`.

Example 3.57: Monod growth rate

```
signal([ name := "s_nutrient", colour := _magenta ]);

function([ name := "fun_monod"
  , input := {":qm_s_nutrient"}
  , type := "hill"
  , params := {0.035, 50.0, 1.0}
  , rnd_params := { 0.0, 0.001 }
]);

strain([ name := "str_A", growth_fun := "fun_monod" ]);
```

Cell death The cells that are in certain states die. There are two fields to describe the dying conditions: `death_threshold` and `death_gate`. `death_threshold` expects a real number that indicates the minimum allowed growth rate. This field is deterministic. By default, it is 0.0, so that the cells with negative growth die. Notice that if `negative_growth` is set to false, then this condition would never meet. `death_gate` expects a `Gate`, either Boolean or quantitative. The cells where the `Gate` evaluates to true die. The default `death_gate` is `"_ga_true"`, never true. An AND operation is performed over the two fields: the two conditions are necessary for the cell to die. The death is instantaneous and deterministic. In order to create an stochastic behaviour and/or the need to stay in the state for a period of time, users can use stochastic `Functions` for the input of the `Gate` or the growth rate calculation and/or the gene expression system with NoisyRegulations.

Example 3.58 shows the usage of a custom death condition. The cells must die when the toxin `Molecule` is produced. The production is noisy, so that there is a 0.5 chance of production of the toxin every time the random numbers are sampled. To make the death independent of the growth rate, the`death_threshold` parameter has been set to an arbitrary high number.

Example 3.58: Death conditions of `Strain`

```
molecule([ name := "mol_toxin", times := { -20.0, 10.0 } ])
  ;
noisy_regulation([ name := "nreg_toxin", acti_on := {1.0, 0
  .1} ]);
operon([ name := "op_toxin", regulation := "nreg_toxin",
  output := {"mol_toxin"} ]);
bplasmid([ name := "bp_toxin", operons := {"op_toxin"} ]);
```

```
strain([ name := "str_sensitive", death_gate := "mol_toxin"
   , death_threshold := 1.0 ]);
cell_type([ name := "cell_doomed", strain := "str_sensitive
   ", bplasmids := {"bp_toxin"} ]);
```

Division The division of cells is also conditional, with two conditions that must hold (an AND over both of them). The first one is the division size. Even though it can be modified at `division_vol`, it is advised to do it with caution, at big changes may break the physics engine. The other condition is a custom `Gate`, `division_gate`. The cells that reach the division volume but do not match the condition imposed by the `Gate`, stop growing and wait for the condition to hold so that they can divide.

Example 3.59 shows how to use the custom division condition and the custom death condition together. The `"qp_A"` `QPlasmid` may represent the chromosome, which has to be replicated before cell division. `"qga_division"` is the division condition, which states that there must be at least two copies of the chromosome. If the cell reaches the division volume without fulfilling this requirement, it cannot divide. It waits for 5 minutes for the division condition to hold. If it does not happen, the cell dies. It is done via `"bga_death"` as a custom death condition. An intermediary `Molecule` and its expression circuit is used to produce the 5 minutes delay. The predefined `"_bm_division_vol"` marker indicates whether the division volume has been reached (see below).

Example 3.59: Division conditions of `Strain`

```
oriv([ name := "ov_A" ]);
qplasmid([ name := "qp_A", operons := {"op_death"}, orivs
   := {"ov_A"} ]);
qgate([ name := "qga_division", input := "qp_A", operator
   := ">=", value := 2.0 ]);

molecule([ name := "mol_death", times := { -20.0, 5.0 } ]);
operon([ name := "op_death", gate := "_bm_division_vol",
   output := {"mol_death"} ]);

strain([ name := "str_A"
  , division_gate := "qga_division"
  , death_gate := "mol_death"
  , death_threshold := 1.0
]);
```

Morphology The element includes a series of fields to modify different aspects of the cellular morphology: the volume of the newly created cells, the volume required by a cell to divide or the symmetry and orientation of the division. All these properties are stochastic. All of them can be specified as a single deterministic value, as the scaling parameters of the default distribution type or as a custom distribution. The default distribution type is the normal except for the initial volume and the division rotation, which use a uniform. The current physics engine limits the modification of these parameters, as it is bases in heuristics. Small variations from the default parameters are usually valid, but important ones may completely break the physics.

Predefined `Strains` There are three predefined `Strains` ready to use. All their fields are as default except for the growth rate. Three growth rates are provided: the maximum theoretical growth rate of E. coli, a smaller one and no growth at all. The details of each of the `Strains` can be checked at **??**.

Predefined markers There are a series of predefined markers that can be used to access variables related to the cellular growth. The list is available in Table 3.15. The `"_bm_"` ones are Boolean while the `"_qm_"` ones are quantitative.

Table 3.14: Predefined Strains

NAME	GROWTH RATE
"_str_wt"	{ 0.03465735903, 0.0003465735903 }
"_str_slow"	{ 0.01, 0.0001 }
"_str_nogrowth"	{ 0.0, 0.0 }

Table 3.15: Available predefined growth markers

NAME	DESCRIPTION
"_bm_division_vol"	Whether the division volume is reached
"_qm_vol"	volume
"_qm_dvol"	delta volume
"_qm_gr"	total growth rate
"_qm_base_gr"	base growth rate

3.6.4 CellType

Keywords: `cell_type`
Recommended prefixes: `cell_`
State: constant tag (digital)
Linked to (direct): `Molecule`, `BPlasmid`, `QPlasmid`, `Strain`
Linked to (reverse): _

The `Cell Type` describes a type of cell as the combination of a `Strain` and a set of biological elements that are present.

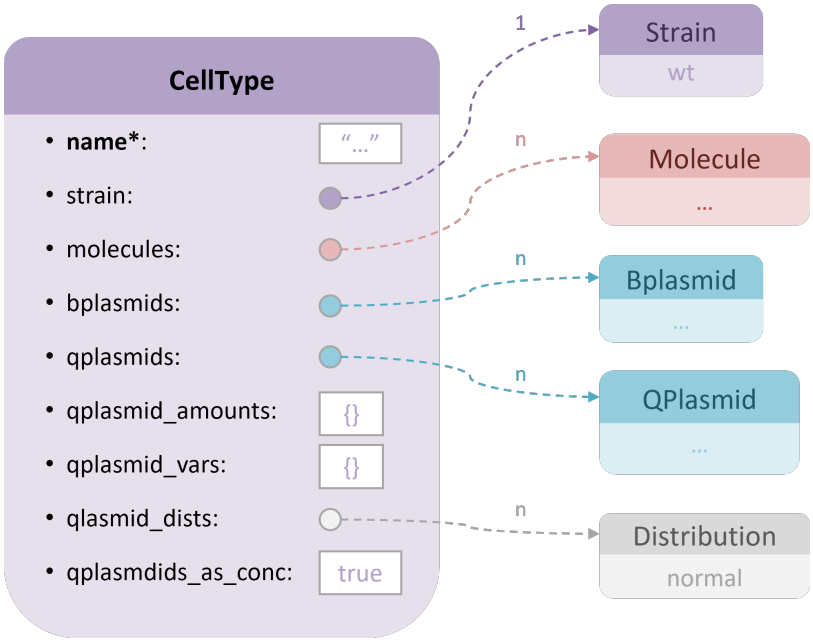

Figure 3.15: `Cell Type` **element**

Description A `Cell Type` is a convenient way to describe different types of cells in the simulation, as the combination of a `Strain` and the biological elements that are characteristic of that cell type. Those elements will be present in newly created cells. However, during the simulation, they can gain and lose elements and, consequently, differ from their original state. Even when this happens, the cells do not "lose" their identity as belonging to a `Cell Type`. The `Cell Type` of a cell is set on creation and

Table 3.16: Parameters of the Strain element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	__	ID name for referencing the objet	__
division_gate	any cellular element, usually Gate	ref.	Condition for the cell to divide, together with reaching the division volume (AND)	"_ga_true"
death_gate	any cellular element, usually Gate	ref.	Condition for the cell to die, together with having a growth rate under death_threshold (AND)	"_ga_true"
growth_rate_fun / growth_fun	any cellular element, usually Function	ref.	Custom growth rate function. If assigned, it replades the built-in behaviour of adding the biomass Fluxes.	"" = sum of the biomass Fluxes
growth_rate_rnd / growth_rnd	any cellular element, usually Randomness	ref.	Randomness that rules the base growth rate	"" = default normal
growth_rate / growth_rate_params / growth / growth_params	positive real or array of real	min-1	Deterministic value or parameters to scale the samples of "growth_rate_rnd". Their meaning depend on the type of distribution, normal by default.	{ 0.03465735903, 0.0 } mean and stddev
ini_vol / ini_vol_params / ini_vol_dist	positive real, array of real, Distribution or dictionary	um3	Volume of newly created cells. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution. Risk of breaking the physics if modified.	{ 1.308997, 2.879793 } lbound and ubound
division_vol / division_vol_params / division_vol_dist	positive real, array of real, Distribution or dictionary	um3	Division volume of the cells. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution. Risk of breaking the physics if modified.	{ 2.879793, 0.070711 } mean and stddev
division_fraction / division_fraction_params / division_fraction_dist	array of positive real in [0.0, 1.0]	fraction	Fraction of the cellular volume asigned to each daughter cell on division. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution. Risk of breaking the physics if modified.	{ 0.5, 0.05 } mean and stddev
division_rotation / division_rotation_params / division_rotation_dist	real in [-360.0, 360.0], positive real, array of real, Distribution or dictionary	angle degres	Relatiive rotation of daughter cells on division. Either a single deterministic one, lbound and ubound of a uniform distribution or a custom distribution. Risk of breaking the physics if modified.	{ -0.125, 0.125 } lbound and ubound
death_threshold	real	min-1	If the growth rate falls under this threshold (not included), the cell dies.	0.0
negative_growth	Boolean	—	If true, negative growth rates are allowed. The cellular volume is not change (volume decrease is not allowed by the physics engine) but the negative value is used for comparisson with "death_threshold".	true
growth_correction	Boolean	—	If true, the growth rate is corrected to offset the error introduced by discretization, considering the step size.	true

maintained immutable during the whole simulation. It is also inherited from mother to daughter cells. This allows for a convenient marking of cells that can be used as a condition. The `Strain` is assigned via the `strain` field, `"_str_wt"` by default. As the `Strain` is attached to the `Cell Type`, is is maintained and inherited too and can also be used to mark and filter the cells.

Boolean elements: `Molecules` and `BPlasmids` The cells of a `Cell Type` are also defined by the `Plasmid` they contain. The `BPlasmids` are given to the `bplasmids` parameter. the field expects a list of them and is empty by default. All the newly created cells will carry the `BPlasmids`, but they may lose them later on if the loss probability is not null. The `BPlasmids` contain the genetic circuits that rule the "bacterial program". This is similar to gro2. In this version, it is possible to assign `Molecules` too, to the `molecules` field, which is also a list and empty by default. The assigned `Molecules` are present (true or 1 state) in newly created cells. However, if there is not any active source (`Operon` or `Flux`), they will begin a degradation process and disappear after their degradation time. This feature is important to set a common initial state for systems with incompatible states, like the Toggle Switch or the Reppresilator.

Example 3.60 shows how to assign the initial `BPlasmids` and `Molecules` to a `Cell Type` element. `"mol_A"` is produced anyway by the plasmid that the `Cell Type` has, but assigning it makes it present from the initial state, without the expression time delays.

Example 3.60: Boolean components of the `Cell Type`

```
molecule([ name := "mol_A" ]);
operon([ name := "op_A", output := {"mol_A"} ]);
bplasmid([ name := "bp_A", operons := {"op_A"} ]);

cell_type([ name := "cell_A"
  , strain := "_str_nogrowth"
  , molecules := {"mol_A"}
  , bplasmids := {"bp_A"}
]);
```

Quantitative elements: `QPlasmids` Assigning `QPlasmids` is more complicated because they can be present at different copy numbers. The `QPlasmids` are assigned to the `qplasmids` field, a list, empty by default. If no other field related to `QPlasmids` is modified, the copy number is sampled form the fallback distribution of each `QPlasmid`. There are a series of fields to allow for assigning different initial copy numbers

of the same QPlasmids to different Cell Types: qplasmid_amounts, qplasmid_vars and qplasmids_dists. If the starting copy numbers of all the QPlasmids follow a normal distribution, then the means are given at qplasmid_amounts and the standard deviations at qplasmid_vars. In both lists, the order is used to assign the parameters to the QPlasmids. If the number of qplasmid_amounts is less than qplasmids, the rightmost QPlasmids will use their fallback distributions. If qplasmid_vars is shorter than qplasmid_amounts, the rightmost positions will be auto completed with zeros, meaning that those copy numbers will be deterministic. The fall-back distributions are not used in that case as qplasmid_amounts eclipses them. In case at least one QPlasmid does not follow a normal distribution, custom distributions have to be assigned for all via the qplasmids_dists.qplasmids_dists has preference over qplasmid_amounts and qplasmid_vars, and these have preference over the distributions defiend at QPlasmid level.

By default, the copy numbers are interpreted as concentration, in copies per um3, so that the initial absolute amounts depend on the initial cell volumes. If the qplasmids_as_conc is set to false, then the sampled copy numbers are interpreted as amounts rather than concentrations.

Example 3.61 shows how to specify the initial amounts of QPlasmids in different ways. The three QPlasmids have copy numbers defined at QPlasmid level that will be used in case no other information is given for them at Cell Type level. For "qp_A", a null distribution name is given at qplasmid_dists, so the value at qplasmid_amounts is used. As there is no value for it at qplasmid_vars, the default value of 0.0 is used and the value is deterministic. The initial copy number distribution for "qp_A" is exactly 200 copies/um ^{3}. For "qp_B", a custom distribution is provided, "dist_B", so its initial copy number follows a normal distribution with mean 20 and standard deviation of 1. As there is no information for "dist_C", the default distribution given at QPlasmid level is used: a normal distribution of mean 100 and deviation 1. All the copy numbers are in concentration units, as qplasmids_as_conc is true by default.

Example 3.61: Quantitative components of the Cell Type

```
qplasmid([ name := "qp_A", copy_num := 10.0 ]);
qplasmid([ name := "qp_B", copy_num := {100.0, 1.0} ]);
qplasmid([ name := "qp_C", copy_num := {100.0, 1.0} ]);

distribution ([ name := "dist_B", params := {20.0, 1.0} ]);

cell_type([ name := "cell_ABC"
  , qplasmids := {"qp_A", "qp_B", "qp_C"}
  , qplasmid_amounts := {200.0}
  , qplasmid_dists := {"","dist_B"}
```

```
]);
```

Table 3.17: Parameters of the CellType element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
strain	Strain	ref.	Strain of the cells	"_str_wt"
molecules	array of Molecule	ref.	Molecules that are present in the newly created cells	{} = none
bplasmids / plasmids / b_plasmids	array of BPlasmid	ref.	BPlasmids that are present in newly created cells	{} = none
qplasmids / q_plasmids	array of QPlasmid	ref.	QPlasmids that are (or may be if stochastic) present in newly created cells	{} = none
qplasmid_amounts / qplasmids_amounts / qplasmid_means / qplasmids_means	array of positive real	copies or copies/um3	Mean parameter of the normal distributions that rule the number of "qplasmids" copies in newly created cells. Autocompleted with 0.0 if shorter than "qplasmids"	{} = filled with 0.0, no copies added
qplasmid_vars / qplasmids_vars	array of positive real	copies or copies/um3	Stddev parameter of the normal distributions that rule the number of "qplasmids" copies in newly created cells. Autocompleted with 0.0 if shorter than "qplasmids"	{} = filled with 0.0, deterministic
qplasmid_dists / qplasmids_dists	array of Distribution	ref.	Distributions that override "qplasmid_amounts" and "qplasmid_vars". Used in case the desired distributions are not normal. If shorter than "qplasmids", normal distributions with "qplasmid_amounts" and "qplasmid_vars" parameters are used to complete.	"" = normal distribution
qplasmids_as_conc / qplasmids_as_concs	Boolean	_	If true, the sampled amounts of "qplasmids" are considered a concentration and thus scaled by the cellular volume. If false, it is interpreted as the absolute number of copies and not scaled.	true

Table 3.18: Summary of all the cell biological elements

NAME	KEYWORD	TYPE	DESCRIPTION	PREFIX	FRIENDS	STATE	LINKS
Molecule	molecule / protein	physica free	Any intracellular molecule (protein, RNA, metabolite, etc.) with Boolean representation	mol_ / m_ / prot_	In: _ . Out: Operon, Flux, CellType	presence (digital)	Subsection 3.1.1, Figure 3.1, Table 3.1
Regulation	regulation / promoter	physical, DNA	Promoter, riboswitch or any other regulation element a potential noisy behaviour	reg_ / prom_	In: _ . Out: Operon	copy number	Subsection 3.1.3, Figure 3.3, Table 3.3
Operon	operon	physica DNA	Actual bacterial operon or general logic block that links a a Regulation to output Molecules	op_	In: Regulation, Molecule. Out: Plasmid	copy number	Subsection 3.1.2, Figure 3.2, Table 3.2
Flux	flux	process	Emission or absortion of a Signal by a cell. That of biomass is a special flux that determines the growth rate.	flux_	In: Molecule, Signal. Out: _	exchanged amount	Subsection 3.2.1, Figure 3.4, Table 3.4
Pilus	pilus /conjugation	abstract	T4TS system that allows its bacterium to act as a conjugation donor	pil_ conj_	In: Randomness, Gate. Out: Function, Gate, Ode, CellColour, OriT	activity	Subsection 3.5.1, Figure 3.10, Table 3.10
OriT	orit / oriT / ori_t	physical, DNA	Origin of transfer that makes its Plasmid transmisible by conjugation	ot_ / orit_	In: Pilus. Out: Plasmid	copy number	Subsection 3.5.2, Figure 3.11, Table 3.11
OriV	oriv / oriV /ori_v	physica DNA	Origin of replication that produces the replication of its QPlasmid. Only for Qplasmid	ov_ / oriv_	In: CopyControl. Out: Qplasmid	copy number	Subsection 3.3.1, Figure 3.5, Table 3.5
Copy Control	copy_control	physical, DNA	Sequence that regulates the activity of an OriV to keep a (almost) constant copy number. Only for Qplasmid	cc_	In: _ . Out: OriV, Qplasmid	copy number	Subsection 3.3.2, Figure 3.6, Table 3.6
Partition System	partition / partition_system	physica DNA	Sequence that produces the exact split of copies between daughter cells on cell division. Only for Qplasmid	part_	In: _ . Out: Qplasmid	copy number	Subsection 3.3.3, Figure 3.7, Table 3.7
Mutation	mutation	abstract	Any transformation of Plasmids into other Plasmids	mut_	In: Plasmid. Out: MutationProcess	_	Subsection 3.4.1, Figure 3.8, Table 3.8
Mutation Process	mutation _process	process	Any process (spontaneous mutation, enzymatic activity, gene editing, etc.) that produces Mutations	mutp_	In: Mutation. Out: _	_	Subsection 3.4.2, Figure 3.9, Table 3.9
BPlasmid	plasmid / bplasmid / b_plasmid	physical, DNA	Actual bacterial plasmid or any (part of) DNA molecule (chromosome, phage, phagemid, etc.) Boolean representation (no coy number)	bp_ / p_ / plas_ / bplas_	In: Operon, OriT. Out: CellType	presence (digital)	Subsection 3.6.1, Figure 3.12, Table 3.12
QPlasmid	qplasmid / q_plasmid	physica DNA	Actual bacterial plasmid or any (part of) DNA molecule (chromosome, phage, phagemid, etc.) Quantitative representation (with copy number).	qp_ / qplas_	In: Operon, OriV, CopyControl, PartitionSystem, OriT. Out: CellType	copy number	Subsection 3.6.2, Figure 3.13, Table 3.13
Strain	strain	abstract	Type of cell with a specific geometry, growth and division behaviour	str_	In: _ . Out: CellType	presence (digital)	Subsection 3.6.3, Figure 3.14, Table 3.16
Cell Type	cell_type	abstract	A Strain plus an initial state (Plasmids and Molecules already present when a cell is created)	cell_	In: Molecule, Plasmid, Strain. Out: _	presence (digital)	Subsection 3.6.4, Figure 3.15, Table 3.17

Chapter 4

Medium elements

This elements exist at simulation/global level. They do not map to real biological elements but instead help the user place agents and signals in the world and extract information.

4.0.1 Grid

Keywords: `grid`
Recommended prefixes: _
Linked to (direct): _
Linked to (reverse): _

This element represents the medium grid where diffusion of extracellular Signals takes place.

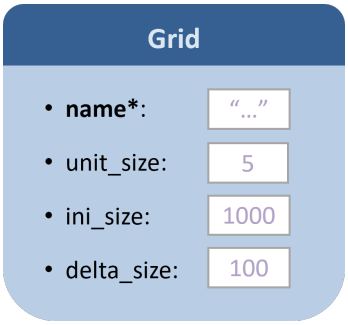

Figure 4.1: **Grid element**

Size The `unit_size` field is used to set the size of each grid unit in 2D as the length of one side (they are squares). The depth is not explicitly given but users have to take it into account to scale all the parameters that are concentrations, as they are relative to the volume of a grid unit. It is forced to be an integer multiple of pixels to ease visualization. Each pixel maps to 0.1 um. The lower the value, the more accurate; the higher, the faster. Values bigger than the default 5 pixels must not be used.

`ini_size` and `delta_size` refer to the number of grid units. They are the initial size and how much the grid grow when it is rescaled. They are low level parameters that only affect performance and users are unlikely to modify.

4.0.2 Signal

Keywords: `signal`
Recommended prefixes: `s_` `sig_`

Table 4.1: Parameters of the Grid element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	—	ID name for referencing the objet. Not used as there is a single grid per simulation.	—
unit_size	positive integer	pixels	Size of each grid cell (length of the side of a squared). The size in um is obtained by dividing by 10, as the scale is 10 pixel/um	5 = 0.5 um
ini_size	positive integer	grid cells	Length of the grid side (square) in number of grid cells when the simulation starts. The total number of grid cells is the suqare of this parameter.	1000
delta_size	positive integer	grid cells	Number of rows and colums added at a time to the grid whenever it is physically rescaled to fit the simulated space.	100

Linked to (direct): _
Linked to (reverse): Flux

An extracellular molecule that diffuses in the medium and can be emitted or absorbed by cells.

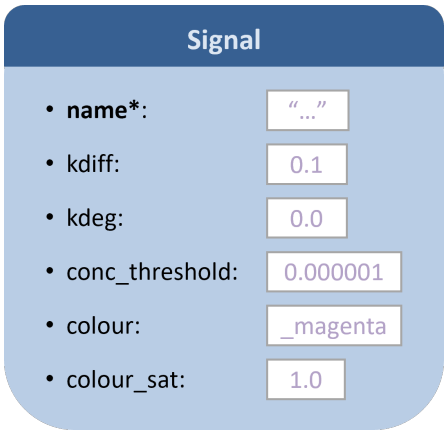

Figure 4.2: **Signal element**

Local concentration The local concentration of a Signal in the surrounding medium of a cell can be accessed by all the cellular elements by adding the `"_conc_"` prefix to the name of a Signal. For example, if a Signal is called `"s_A"`, its concentration is accessed as `"_conc_s_A"`. On the other hand, what the `"s_A"` identifier would retrieve is the total exchange of this Signal between the cell and the medium at that specific point of time (derivative), result of adding all the Fluxes that involve the Signal.

Constants The behaviour of the Signal is ruled by two constants: `kdiff` and `kdeg`. `kdiff` is the diffusion constant, in um^2/min.

Colour Signals can be assigned different colours for visualization purposes using the `colour` field, in RGB format. The intensity of colour is proportional to the concentration of Signal at each Grid unit. Once a saturation value is reached, the intensity is the maximum. Users can modify that saturation concentration via the `colour_sat` parameter. A good value is an estimation of the maximum concentration the Signal is going to reach during the simulation, so it is always possible to tell different concentrations apart visually.

Table 4.2: Parameters of the Signal element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
kdiff / k_diff	positive real	user-defined units of concentration · min-1	Diffusion constant	0.1
kdeg / k_deg	positive real in [0.0, 1.0]	user-defined units of concentration · min-1	Degradation constant	0.0
consc_threshold	positive real	user-defined units of concentration	Minimum concentration to store in the grid. Concentrations under this value are rounded to 0. The lower, the most accurate; the higher, the faster.	0.000001
colour / color	array of 3 real in [0.0, 1.0]	RGB in [0.0, 1.0]	Colour of the signal. Signals without a colour are not rendered.	_magenta
colour_sat / color_sat	positive real	user-defined units of concentration	Concentration at which the maximum intensity of colour is reached. Concentrations over this value are not distinguishable by colour.	1.0

Table 4.3: Summary of all the medium elements

NAME	KEYWORDS	TYPE	DESCRIPTION	PREFIX	FRIENDS
Grid	grid	medium	The grid where the diffusion of signals in the medium is computed. A single one per simulation.	_	_
Signal	signal	medium	A diffusible extracellular molecule	s_ / sig_	Out: Flux

Chapter 5

Elements for simulation control

This elements exist at simulation/global level. They do not map to real biological elements but instead help the user place agents and signals in the world and extract information.

5.1 Global Parameters

A series of parameters are not bounded to any specific element; they are the global parameters, shown in the following snippet (Example 5.1).

Example 5.1: Global parameters

```
global_params([ seed := 1234
  , seed_offset := 11
  , step_size := 0.1
  , sensitivity := 0.01
  , cells_per_thread := 10000
  , max_threads := 10
  , theme := _dark_theme
  , auto_zoom := false
  , show_plots := false
  , render_signals := true
  , batch_mode := false
  , show_performance := false
]);
```

.

Random seeds The `seed` is the master random seed of the simulation. It is used to generate secondary seeds for every stochastic element in the simulation. Simulations using the same `seed` are identical. Saving the seeds used for relevant experiments is recommended for reproducibility. `seed_offset` is the amount added to the current seed every time the experiment is restarted with a different random seed. The initial execution an any subsequent executions in case the green arrow of the GUI is pressed use the `seed` specified in the .gro file. Every time the orange arrow of the GUI is used to reload, a new seed is calculated by adding `seed_offset`.

Initial Boolean options `auto_zoom` indicates whether the automatic zoom functionality is active when the simulation starts. The automatic zoom locks the manual zoom and navigation to automatically adjust the view to the size of the colony. It is specially useful when recording videos. By default, it is false, but the user can switch it at any time using the GUI.

`show_plots` indicates whether the window with real time plots is visible when the simulation starts. Its visibility can be switched at any time using the GUI. By default, it is false.

`render_signals` indicates whether the medium Signals are rendered using colours. It is true but default. This parameter cannot be changed from the GUI.

`batch_mode` indicates whether the simulation is run in interactive mode (false, the default) or batch mode (true). The only difference is that in batch mode the application is automatically closed when a custom condition (Checkpoint, see Subsection 5.3.2), allowing for subsequent executions from the command line. The GUI is present in both modes to allow for the automatic gathering of Snapshots and Plots.

Efficiency-related parameters `sensitivity` is the relative change necessary to update the internal state of cells. For example, it determines the change in the Gillespie reactions rates that produces the current event to be discarded and a new one sampled. Changes under the sensitivity are considered to be negligible and the performance impair of having to update the state in response to minor changes is removed. The default value, 0.01 means a 1% change. With this value, the events are expected to be re-sampled every time step if the cells are growing and the rates are dependent on the volume. Higher values speed up the simulations at the cost of lower-quality approximate results.

`cells_per_thread` determines the number of cells that are run by each thread. The optimal number depends on the specific machine and the complexity of the simulation. `max_threads` serves as a cap to avoid the overhead of an excessive number of threads when the cell population gets very large.

`show_performance` is used to measure the performance. If set to true, the elapsed time is printed to the messages console, both in real world time and simulation time and their ratio. The print of the last run lapse appears whenever the simulation is paused.

Graphical theme The `theme` parameter expects a dictionary with a colour palette. The specific fields are explained in Table 5.2. Besides the default theme, there are two additional themes already defined at the library file: a light theme (`_light_theme`) and a dark theme (`_dark_theme`). Notice that quotation is not used in this case as they are not gro elements but dictionary ccl variables.

5.2 Population-level logic elements

This elements are used for aggregating individual cell-level data of several cells into population-level metrics (PopulationStat, Subsection 5.2.1), combine and transform those values by applying mathematical functions (PopulationFunction, Subsection 5.2.2) or create conditions out of them (PopulationBGate, **??** and PopulationQGate, Subsection 5.2.3). The population-level PopulatFunction, and PopulationBGate and PopulationQGate are analogous to the individual-level Function, BGate and QGate described in previous sections.

Table 5.1: Global parameters

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
theme	Theme or direct Theme dictionary	ref.	The graphical theme or colour palette to apply	_dark_theme
seed	positive integer	—	Random seed used as the master seed for all the randomness in the simulation	1234
seed_offset	positive integer	—	Offset added to the seed every new simulation	11
step_size	positive real	min	Duration in minutes of the time step. Should not be changed.	0.1
sensitivity	positive real	ratio	Minimum relative change in quantitative cellular magnitudes related to growth (volume, growth rate...) and Gillespie reaction rates necessary to trigger an update. The lowest, the more accurate; the higher, the faster.	0.01
cells_per_thread	positive integer	number of cells	Number of cells evaluated by each thread. The total number of parallel threads is the number of cells in the simulation divided by this parameter. The optimal number depends on the concrete machine, size of the simulation an the complexity of the biocircuits.	10000
max_threads	positive integer	threads	Maximum number of threads to use for the parallel update of cells. Used to cap multithreading to the available threads of each macnine.	10
auto_zoom	Boolean	_	If true, the automatic zoom feature is enabled when the simulation starts. It can be manually modified later.	false
show_plots	Boolean	_	If true, the plots window is visible when the simulation starts. Its visibility can be manually modified later.	false
render_signals	Boolean	_	If true, the diffusible signals that are in the medium are rendered using colours.	true
batch_mode	Boolean	_	If true, Checkpoints stop the simulation and close gro. If false, Checkpoints just pause the simulation and wait for user input.	false
show_performance	Boolean	—	If true, the real world time taken is printed to the messages console everytime the simulation is paused (manually or by a Checkpoint).	—

Table 5.2: Theme dictionary

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
background	string (hexadecimal colour)	_	Colour of the background (medium)	"#000000"
cell_outline	string (hexadecimal colour)	_	Colour of the cellular border	"#444444"
cell_selected	string (hexadecimal colour)	_	Colour of the cellular border of selected cells (disabled)	"#880000"
message	string (hexadecimal colour)	_	Colour of text messages	"#ffffff"
mouse	string (hexadecimal colour)	_	Colour of the cursor	"#ffffff"

The output of any of these elements can be fed into an output element to be printed (OutFile) or plotted (Plot). PopulationBGate and PopulationQGate can additionally be used as the condition of Timers.

5.2.1 Population Statistics (PopulationStat)

Keywords: `pop_stat`
Recommended prefixes: `stat_`
Linked to (direct): any cell-level element
Linked to (reverse): PopulationFunction, PopulationQGate, OutFile, Plot

Description This element aggregates a cell-level element into a population-level metric using statistics.

Input Any cell-level element can be assigned to the `input` parameter. Both logic and biological elements are eligible. Also, both the elements of Boolean nature and quantitative nature. The state of the chosen element in different cells is aggregated into a single population-level value. Assigning an element to this parameter is mandatory.

There is a predetermined Population Stat ready to use: `_stat_cellnum`. This is just a counter of the total number of cells in the simulation. Its implementation is as the

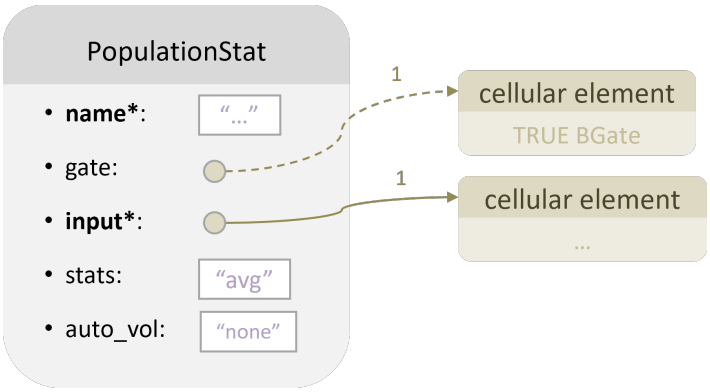

Figure 5.1: **PopulationStat element**

one in Example 5.2. To count every cell, the predefined cell-level BGate `"_ga_true"` is used as its state is true (1) in every cell. The utilized statistic is `"sum"` (see the full list of statistics below). This performs a summation of the chosen element. When the element is Boolean, as in this case, this equals to counting.

Example 5.2: Cell number predetermined PopulationStat

```
pop_stat([ name := "_stat_cellnum"
  , input := "_ga_true"
  , stats := {"sum"}
]);
```

The Population Stat in Example 5.3 tracks the composition of a cellular consortia. It calculates the fraction of all the cells in the simulation that are of `"cell_A"` type. The statistic is the average (by default) as no other is specified. When used with a Boolean input, as is the case, the average statistic equals a ratio.

Example 5.3: Filtered PopulationStat

```
cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B" ]);

pop_stat([ name := "stat_Acells"
  , input := "cell_A"
]);
```

The state of the selected input can be automatically converted between amount and concentration without the need of explicit Functions. The `auto_vol` option, analogous to that of Functions, is used to that purpose.

Besides cellular elements, there are some special keywords that can be used instead to extract data related to the position, size and growth of the cells. The complete set of keywords and their meanings can be consulted at Table 5.3. Notice the underscore at the beginning of the name.

Table 5.3: Special fields for PopulationStat

KEYWORD	DESCRIPTION
"_x"	position in x
"_y"	position in y
"_gr"	growth rate in min-1
"_gt"	generation time in min
"_volume"	volume in um3
"_d_volume"	volume increase in um3 min-1
"_length"	length in um
"_d_length"	length increase in um min-1

Filter Gate Any cell-level element can be assigned to the `gate` parameter to be used as a filter. Only the cells where the condition evaluates to true (if a Gate) or the element is present are used to calculate the statistic and the rest is ignored. There is no filter by default (all the cells in the simulation are used).

In Example 5.4 the CellType is used to filter the cells included in the calculation. Only the cells of the type `"cell_A"` are considered in the calculation; those of `"cell_B"` CellType are ignored.

Statistics The parameter `stats` is used to set a list of statistics to compute. The default statistic is the average, but others can be selected. The currently available ones include typical statistics: summation, average, standard deviation and range. The difference between summation and accumulation is that the summation is reset every time step, providing the sum of the current states of the cells while the accumulation provides the total sum from the beginning of the simulation. The typical application of accumulation is to count events, like conjugation events. There are two alternative ways to provide the statistic:

- As a list with the concrete statistics to use. Even if a single statistic is provided, the curly brackets of the list are required.
- A single keyword that translates into a group of related statistics

The complete list of available statistics and groups of them is given in Table 5.4.

Table 5.4: Available PopulationStat types

METRIC	KEYWORD	COMMENTS
accumulation	"acc"	_
sumation	"sum"	_
average	"avg"	_
standard deviation	"stdedv"	_
maximum	"max"	_
minimum	"min"	_
just the average	"avg"	= {"avg"}
classic	"classic"	= {"sum","avg","stdev"}
range	"range"	= {"min", "max"}
all	"all"	= {"sum","avg","stdev","min","max"}

In Example 5.4 the `"classic"` group of statistics is used. It includes the sum, the average and the standard deviation. The input is a special field: the growth rate. This PopulationStat calculates the sum, the average and the standard deviation in the growth rate of the cells of the `"cell_A"` CellType only.

Using the output of a Cell Statistics PopulationStats can be fed into output elements: OutFiles and Plots. In this case, the whole PopulationStat can be given to them and all the statistics that it includes will be printed/plotted serially. It is also possible to extract a specific statistic from the group. These are referenced by concatenating the name of the statistic to the name of the PopulationStat. For example, the PopulationStat at Example 5.4 includes the three `"classic"` statistics. The `"file_allstats"` OutFile (see Subsection 5.5.2) will print the three of them. The `"file_avg"` OutFile, on the other hand, will print the average only. The other two statistics would have been accessed in an analogous way as `"stat_gr_sum"` and`"stat_gr_stddev"`. The input to a PopulationFunction or PopulationGate must ve a single statistic and not the whole pack. If the whole element is referenced and it has a single statistic, that one would be picked. If it has more than one, the first one will be selected, using this ordering: "sum", "avg", "stddev", "min", "max". The user should be careful not to reference statistics that are not included in the chosen PopulationStat.

Example 5.4: Extraction of individual statistics from a PopulationStat

```
cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B" ]);

pop_stat([ name := "stat_gr"
  , input := "_gr"
  , gate := "cell_A"
  , stats := "classic"
]);

out_file([ name := "file_allstats", pop_fields := {"stat_gr
   "} ]);
out_file([ name := "file_avg", pop_fields := {"stat_gr_avg"
   } ]);
```

5.2.2 Population Function

Keywords: `pop_function`
Recommended prefixes: `pfun_`
Linked to (direct): PopulationStat and other PopulationFunction
Linked to (reverse): PopulationBGate, PopulationQGate, OutFile, Plot

This element applies a mathematical function to one or more population-level elements. This is the analogous of the individual-level Function.

The behaviour and parameters of this element are analogous to those of the cellular Function (Subsection 2.2.1) so the user is referred to that section for details. The main difference relies on the input being population-level elements in this case and cellular elements in the case of the Function. Another important difference is that the PopulationFunction is deterministic i.e. it lacks the randomness-related parameters that the Function has.

Different Population Function elements can be chained to compute composed functions. This should be done in a unidirectional way; reference loops must be avoided as they would produce undefined behaviour.

In Example 5.5 a Population Function is used to calculate the ratio between two Population Stats. Each of these Population Stats computes the average growth rate of a different Cell Type. Their values are combined in a generalized product to create the ratio. The result could be then printed, plotted or fed into another Population Function for further calculations.

Table 5.5: Parameters of the PopulationStat element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
gate / filter_gate	any cellular element, usually Gate	ref.	If this condition evaluates to true, the cell is counted; ignored otherwise.	TRUE BGate
input* / elem*	Cellular Element / string in { "_x", "_y", "_gr", "_gt", "_vol", "_dvol", "_legth", "_dlength" }	ref.	The cellular element to aggregate. Alternatively, a cellular property from: x or y position, growth rate, generation time, volume, delta volume, length and delta length.	_
stats	array of strings in {"sum", "avg", "stddev", "min" "max"} or string in {"avg", "classic", "range", "all"}	—	The aggregation statistics to compute: summation, average, standard deviation, minimum, maximum. "classic" = {"sum","avg","stddev"}, "range" = {"min","max"}	"avg" = the average
auto_vol	string in {"none","prod	—	Used to automatically multiply or divide the "input" by the celllular volume to convert between amounts and concentrations.	"none" = nothing done

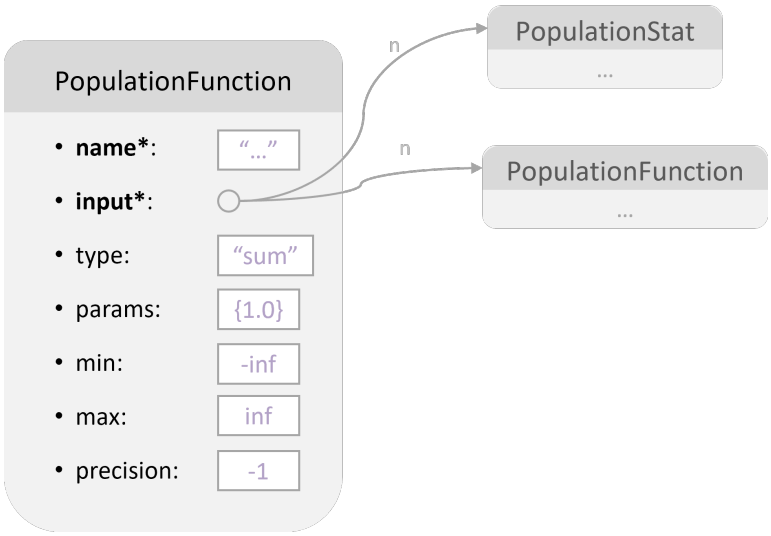

Figure 5.2: **PopulationFunction element**

Example 5.5: PopulationFunction to calculate a ratio

```
cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B" ]);
pop_stat([ name := "stat_grA", input := "_gr", gate := "
   cell_A" ]);
pop_stat([ name := "stat_grB", input := "_gr", gate := "
   cell_B" ]);

pop_function([ name := "pfun_grRatio"
  , input := { "stat_grA_avg", "stat_grB_avg" }
  , type := "product"
  , params := {1.0, -1.0}
]);
```

5.2.3 Population Quantitative Gate (PopulationQGate)

Keywords: `pop_qgate`
Recommended prefixes: `pqga_`
Linked to (direct): PopulationStat, PopulationFunction
Linked to (reverse): PopulationBGate, `Timer`, OutFile, Plot

A quantitative condition (equation or inequation) applied to population-level ele-

Table 5.6: Parameters of the PopulationFunction element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	—	ID name for referencing the objet	—
input* / inputs*	array of PopStat or another PopFunction	ref.	The global elements to compute with. Different function types support different number of inputs (extra inputs are ignored).	—
type	string in {"sum", "product", "exp", "log", "sigmoid", "hill"}	—	The mathematical function to compute	"sum" = weighted summation
params	array of real	—	Parameters whose meaning depend on the selected function type	{1.0} Together with the "sum" type creates a unit function that just retrieves the first input.
min	real	—	Minimum value. Values higher than this are capped.	-infinite
max	real	—	Maximim value. Values lower than this are capped.	infinite
precision	integer	decimal places	Number of decimal places to round the output	-1 = no rounding (may introduce float point rounding errors)

ments.

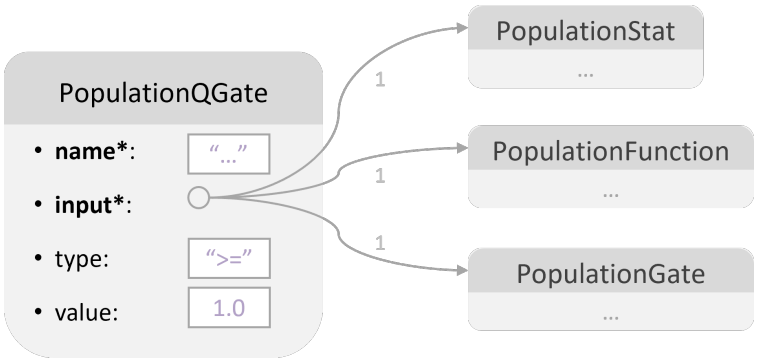

Figure 5.3: **PopulationQGate element**

As this element is analogous to the QGate explained in a previous section, the detailed explanation can be found at Subsection 2.3.1. The difference resides in the input: a population-level element in this case while the input for a QGate it is an individual-level element. The PopulationQGate is applied to a PopulationStat directly or indirectly (through a PopulationFunction).

The example in Example 5.7 uses a PopulationStat directly as the input of a PopulationQGate. The gate evaluates to true if the average growth rate of all the cells in the simulation is higher than 0.035 min^{-1}. Notice that the specific statistic (the average) has to be accessed by appending its name.

Example 5.6: Simple condition with a PopulationQGate

```
pop_stat([ name := "stat_gr", input := "_gr" ]);

pop_qgate([ name := "qpga_gr"
  , input := "stat_gr_avg"
  , value := 0.035
]);
```

Example 5.7 shows a more complex case where an intermediate PopulationFunction is used to perform a calculation between the PopulationStats and the PopulationQGate. This is an extension of the example previously shown in Example 5.5 where a PopulationQGate has been added. The resulting behaviour is a condition that evaluates to

true when the ratio between the average growth rate of the two types of cell is greater or equal that 1, i.e. when the average growth rate of the `"cell_A"` cells is not lower than those of the `"celll_B"` cells.

Example 5.7: Chaining a PopulationFunction with a PopulationQGate

```
cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B" ]);
pop_stat([ name := "stat_grA", input := "_gr", gate := "
   cell_A" ]);
pop_stat([ name := "stat_grB", input := "_gr", gate := "
   cell_B" ]);

pop_function([ name := "pfun_grRatio"
  , input := { "stat_grA_avg", "stat_grB_avg" }
  , type := "product"
  , params := {1.0, -1.0}
]);

pop_qgate([ name := "qpga_gr"
  , input := "pfun_grRatio"
  , value := 1.0
]);
```

Table 5.7: Parameters of the PopulationQGate element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
input	Pop Stat or Pop Function	ref.	The global element to compute with as the left side of the (in)equation	{}
operator / comp_operator	string in {">", ">=", "<", "<=", "=", "!="}	_	The comparison operator tu use. "=" and "!=" should not be used with real inputs"	">="
value	real	_	The righ hand value to compare the input with	1.0

Population Boolean Gate (PopulationBGate)

Keywords: `pop_bgate`
Recommended prefixes: `pbga_ pga_`
Linked to (direct): PopulationBGate, PopulationQGate
Linked to (reverse): PopulationBGate, `Timer`, OutFile, Plot

This is a Boolean condition (a logic gate) checked on population-level elements, analogous to the individual-level BGate.

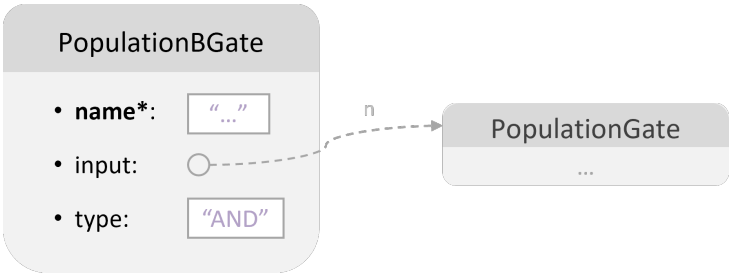

Figure 5.4: **PopulationBGate element**

As this element is analogous to the BGate explained in a previous section, the detailed explanation can be found at Subsection 2.3.2. The difference resides in the input: population-level elements in this case while the input for a BGate are individual-level elements. As the only Boolean elements at population level are PopulationBGate, PopulationQGate, the inputs of this element are other gates.

The typical input is a list of PopulationQGates. Other PopulationBGates can also be inputs, creating unidirectional chains to compute composited conditions. As in the PopulationFunction, loops are not allowed. The PopulationBGate in Example 5.8 evaluates to true when both the average growth rate of all the cells is at least 0.035 min^{-1} and the average growth rate of the `"cell_A"` cells is not lower than those of the `"celll_B"` cells.

Example 5.8: Composite condition using a PopulationBGate

```
pop_bgate([ name := "bpga_"
  , input := { "qpga_gr", "qpga_gr" }
]);
```

There are two predefined PopulationBGates, shown in Table 5.8, analogous to those

at cell-level.

Table 5.8: Predetermined PopulationBGates

NAME	FUNCTION	INPUT
"_pga_true"	"TRUE"	{}
"_pga_false"	"FALSE"	{}

Table 5.9: Parameters of the PopulationBGate element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
input / inputs	array of Pop Gate	ref.	The global elements to compute with. Use "-" preceding the name to negate an input.	{}
type	string in {"TRUE", "OPEN", "FALSE", "CLOSED", "YES", "NOT", "AND", "OR", "XOR", "NAND", "NOR", "XNOR"}	_	The logic function to compute	"AND": equivalent to YES when a single input given. Equivalent to TRUE when no inputs given.

5.3 Control elements

5.3.1 Timer

Keywords: `timer`
Recommended prefixes: `t_`

Linked to (direct): PopulationGate
Linked to (reverse): Checkpoint, CellPlacer, CellPlating, SignalPlacer, Snapshot, OutFile, Plot

Description The element that rules when the associated elements (Checkpoints, placers and output elements) are executed. This can be used to create one-time, periodic (loops), stochastic and conditional (if) executions.

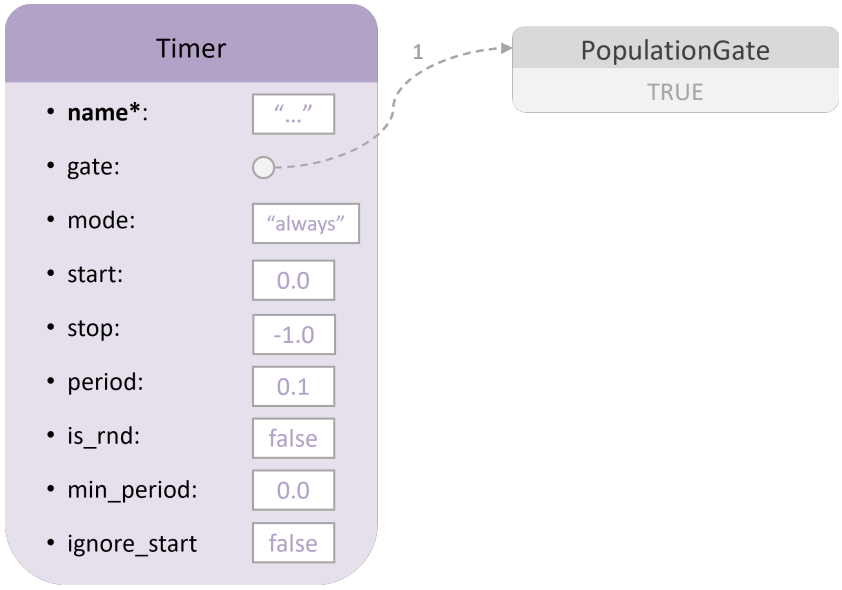

Figure 5.5: `Timer` **element**

Start, stop and period The basic fields of a `Timer` are `start`, `stop` and `period`, all of them positive real numbers and all of them in minutes. Additionally, they should be multiples of the time step size (typically 0.1) to avoid rounding issues. The `Timer` is active from `start` to `stop`, both included. During its lifespan, the `Timer` goes off every `period` minutes. The `Timer` in Example 5.9 goes off at time 200.0 minutes the first time, then at 201.0, 202.0.. and so on until time 1000.0 minutes, included. An associated element would be triggered at these time points. It is not mandatory to override the three fields. By default, `start` is 0.0, `stop` is infinite and `period` is the typical time step size (0.1 min). That means, an event every time step during the whole simulation. In order to make a

Example 5.9: Simple deterministic `Timer`

```
timer([ name := "t_everyMin"
  , start := 200.0
  , stop := 1000.0
  , period := 1.0
]);
```

One-time Timers Even though the typical behaviour of Timers is periodic, they can be used to define a one-time event too. There are two ways to achieve that, as shown in Example 5.10: (i) making the `stop` value the same as `start` as in `"t_oneTime"`; or (ii) setting the `period` to a negative value (or 0.0) as in `"t_same"`. The event takes place at the time specified in the `start` field. Note that, if the `stop` field was lower than the `start` one or negative, the `Timer` would never go off.

Example 5.10: One-time Timers

```
timer([ name := "t_oneTime"
  , start := 200.0
  , stop := 200.0
]);

timer([ name := "t_same"
  , start := 200.0
  , period := -1.0
]);
```

Stochastic Timers Timers can be stochastic. In that case, the period or time between events follows an exponential distribution (like Poisson processes), being the `period` field the mean waiting time. By default, Timers are deterministic. To turn a `Timer` into a stochastic one, the `is_rnd` (false by default) must be set to true. An additional field, `min_period`, may be used to stablish a minimum time between events. This value must be a positive integer, not greater than `period`. The timer in Example 5.11 has the same bounds and period that the previous one, but this one is stochastic. As the minimum time between events is 0.2 minutes, the time to the next event is calculated as 0.2 plus an exponential random variable with mean 0.8 min (1.0 - 0.2). In stochastic Timers, the last event is not guaranteed to match the `stop` bound. Any sampled waiting time smaller than the time step is rounded to the time step, as the maximum number of

events per time step is one. By default, the first event is deterministic and equal to the start parameter. In order to ignore this one to make the first event stochastic too, set the `ignore_start` option to true.

Example 5.11: Stochastic `Timer`

```
timer([ name := "t_everyMinRnd"
  , start := 200.0
  , stop := 1000.0
  , period := 1.0
  , is_rnd := true
  , min_period := 0.2
]);
```

Conditional Timers and their modes Timers can use a PopulationGate as a condition through their `gate` field. In the default `mode`, the `Timer` only goes off if the `gate` evaluates to true when it is evaluated. If a `Timer` is conditional, its `period` parameters turns into the evaluation frequency; the `gate` condition is checked every `period` minutes. The simplest and recommended behaviour when using the `gate` parameter is to leave the `period` one as default so that the condition is evaluated every time step and the `Timer` goes off as soon as the condition is true. This is the behaviour of `"t_maxCells1"` from Example 5.12.

In the default `mode`, if both parameters are used at the same time, the result can be understood as the intersection between the PopulationGate and the cyclic behaviour of a "normal" `Timer`. From start to stop, every period minutes, the Gate is checked and the `Timer` only produces an event if the Gate is on. This is the behaviour of `"t_maxCells2"` from Example 5.12. A `Timer` can be stochastic and conditional at the same time.

There are four alternative modes assigned via the `mode` parameter and summarized in Table 5.10. The default is `"always"` and has the behaviour described above. `"t_maxCells3"` from Example 5.12 is of the `"once"` mode so it would be triggered just once, the first time it is evaluated and its associated `gate` happens to be true. `"t_maxCells4"`, which is of the `"change"` mode, is triggered every time it is evaluated and the state of `gate` is true and it was false in the previous evaluation, i.e. when it switches from false to true. Notice that the change is relative to the previous evaluation and not to the previous time step when using a `period` different to the time step size.

The last mode is `trigger`, exemplified by `"t_maxCells5"`. Once it is evaluated and its `gate` happens to be true, this `Timer` will be periodically triggered until the `stop` time is reached, with independence of the state of the `gate`. This type of `Timer` reinterprets the `period` parameter as the period of those periodic `Timer` events. The evaluation

frequency in this case is every time step (the `Timer` is triggered as soon as the condition matches).

A comparison of the different modes with default and custom parameters is shown in **??**.

Table 5.10: Modes of the Timer element

MODE	DESCRIPTION
"always"	The Timer goes off whenever the condition holds.
"once"	The Timer only goes off the first time the condition holds.
"change"	The Timer goes off every time the condition switches from false to true.
"trigger"	Once the condition is true once, the Timer goes off every time it is checked regardeless of the value of the condition.

Example 5.12: Conditional timers with different modes

```
pop_qgate([ name := "pqga_maxCells"
  , input := "_stat_cellnum"
  , value := 10000.0
]);

timer([ name := "t_maxCells1"
  , gate := "pqga_maxCells"
  , start := 200.0
  , stop := 1000.0
]);

timer([ name := "t_maxCells2"
  , gate := "pqga_maxCells"
  , start := 200.0
  , stop := 1000.0
  , period := 1.0
]);
```

```
timer([ name := "t_maxCells3"
  , gate := "pqga_maxCells"
  , mode := "once"
  , start := 200.0
  , stop := 1000.0
]);

timer([ name := "t_maxCells4"
  , gate := "pqga_maxCells"
  , mode := "change"
  , start := 200.0
  , stop := 1000.0
]);

timer([ name := "t_maxCells5"
  , gate := "pqga_maxCells"
  , mode := "trigger"
  , start := 200.0
  , stop := 1000.0
  , cooldown := 1.0
]);
```

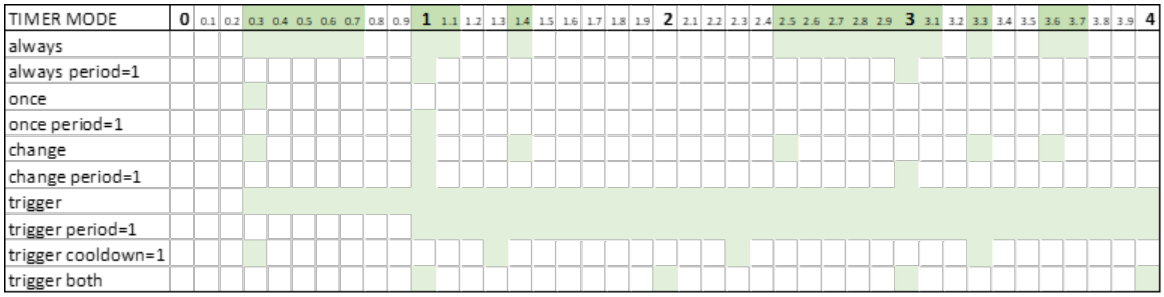

Figure 5.6: **Comparison of the `Timer` modes**. The upper rule indicates time in minutes, assuming a time step of 0.1 min. The case called "both" implies a period of 1.0

Default and implicit Timers Three default timers are provided to cover the most frequent use cases: a one-time `Timer` at the beginning of the simulation and periodic

Timers that last the whole simulation and go off every time step or every minute (see **??**). The three of them are deterministic and not conditional.

Table 5.11: Predefined Timers

NAME	START	STOP	PERIOD
"__t__start"	0.0	0.0	0
"__t__every__step"	0.0	infinite	0.1
"__t__every__min"	0.0	infinite	1.0

Downstream elements Timers can be linked to Checkpoints, any placer element and any output element (any element with a field called `timer`). These elements, with the exception of Checkpoint, have default timers used in case the user does not provide one. Placers have `"_t_start"` as default, so that they do their task just once at the beginning of the simulation. The user may override the `timer` field of a world placer with a `Timer` (either a predefined or a custom one). Output elements have `"_t_every_step"` as default i.e. their behaviour is periodic.

5.3.2 Checkpoint

Keywords: `checkpoint`
Recommended prefixes: `stop_ ckpt_`
Linked to (direct): `Timer`
Linked to (reverse): `_`

Description A Checkpoint is a stop in the simulation. In batch mode, it throws the end of the simulation so that the gro application is closed and a new instance of it can run the next experiment. That implies that only one Checkpoint is executed: the one occurring the first. In interactive mode, on the other hand, a Checkpoint just pauses the simulation and prompts a notification. The simulation can then be examined or resumed by the user.

Timer The associated *timer* determines when the Checkpoint is executed. An optional custom message is shown in interactive mode only.

Table 5.12: Parameters of the Timer element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the objet	_
gate	PopGate	ref.	Population-level Gate used as a condition for the timer to go off	TRUE BGate
mode	string in {"always", "once", "change", "trigger"}	_	How the gate condition is used to determine when the timer goes off	"always" = every period time the gate condition is checked. An event is produced if it evaluates to true.
start	positive real	min	Time when the timer starts	0.0
stop	real	min	Time where the timer ends. A negative value means forever.	-1.0 = forever
period	real	min	Between start and stop, the Timer checks the condition every period time. Must be a multiple of the step size (rounded otherwise). If negative, there is no period, only one checking at the start.	0.1 (the typical time step size)
is_rnd	Boolean	_	If random (true), the period is interpreted as the mean waiting time between checks (exponentially distributed).	false = deterministic
min_period	positive real	min	The minimum time between consecutive checks. Only applies to random Timers.	_ = deterministic
ignore_start	Boolean	_	Whether to ignore the first timr event, matching the start field. Useful for random timers	false

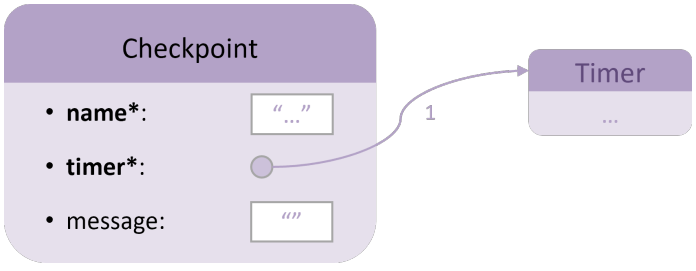

Figure 5.7: **Checkpoint element**

The Example 5.13 shows a simple periodic Checkpoint that pauses the simulation every 100 minutes. In the default interactive mode, a notification with the set custom message is prompted. The resulting prompt is shown in Figure 5.8.

Example 5.13: Simple periodic Checkpoint

```
timer([ name := "t_every100min", period := 100.0 ]);

checkpoint([ name := "stop_periodic"
  , timer := "t_every100min"
  , message := "Another 100 minutes elapsed"
]);
```

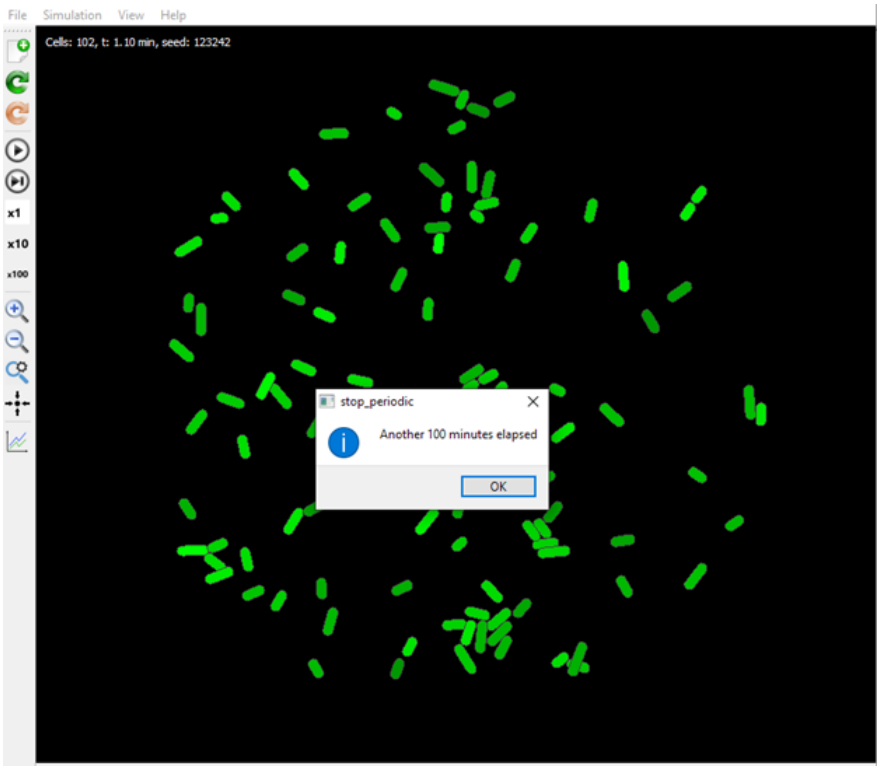

Figure 5.8: **Checkpoint prompt**

Cell number limit A classical gro behaviour from previous versions consists on stopping the simulation when a user-defined number of cells is reached. Such a behaviour is produced by linking a conditional `Timer` to a Checkpoint as is done in Example 5.14. This `Timer` goes off when its condition holds. The condition is a PopulationGate with "$_{s}tat_{c}ellNum$" in the left side, and 1000 in the right side, using the default ">=" comparison operator. "$_{s}tat_{c}ellNum$" is the predefined PopulationStat for the total number of cells. Any user-defined PopulationStat can be used instead. Any other parameter of the `Timer` is leaved as default. That means that it is evaluated every time step (0.1 minutes)during the whole simulation. The Checkpoint is executed when the total number of cells reaches 1000 and then the application is closed because the batch mode has been activated at the global parameters. Notice that in interactive mode, a "*once*" or "*change*" `Timer` mode would be more appropriate.

Example 5.14: Cell number Checkpoint

```
global_params([ batch_mode := true ]);
pop_qgate([ name := "qpga_cellNum"
  , input := "_stat_cellNum", value := 1000.0 ]);
timer([ name := "t_cellNum", gate := "qpga_cellNum" ]);

checkpoint([ name := "stop_cellNum"
  , timer := "t_cellNum"
]);
```

Table 5.13: Parameters of the Checkpoint element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the object	_
timer*	Timer	ref.	When the checkpoint acts	_
message	string	ref.	Custom message to show	""

5.4 Placer elements

This group of elements rule the placement of cells and molecular signals in the world of the simulation. They are triggered by a linked `Timer`.

They are:

- **CellPlacer** : to place new cells
- **CellPlating** : to remove or relocate existing cells
- **SignalPlacer** : to place or remove extracellular signals

5.4.1 CellPlacer

Keywords: `cell_placer`
Recommended prefixes: `cp_`
Linked to (direct): CellType, `Timer`
Linked to (reverse): `_`

Description This element places new cells in the simulation world. The type and number of cells as well as their location is user-defined.

Cell types `cell_types` is the only mandatory parameter. It is necessary to stablish the Strain of the placed cells and their initial state (in terms of Molecules and Plasmids). In the simplest case, a single CellType is given. This case is shown in the following example (Example 5.15).

Example 5.15: Simplest CellPlacer

```
cell_type([ name := "cell_A" ]);

cell_placer([ name := "cp_simple"
  , cell_types := {"cell_A"}
]);
```

This code, with all the optional parameters as default, places a single cell of the specified CellType at the center of the world when the simulation starts (time 0.0). Notice that, even if a single CellType is given, it has to be done as a list using the curly brackets.

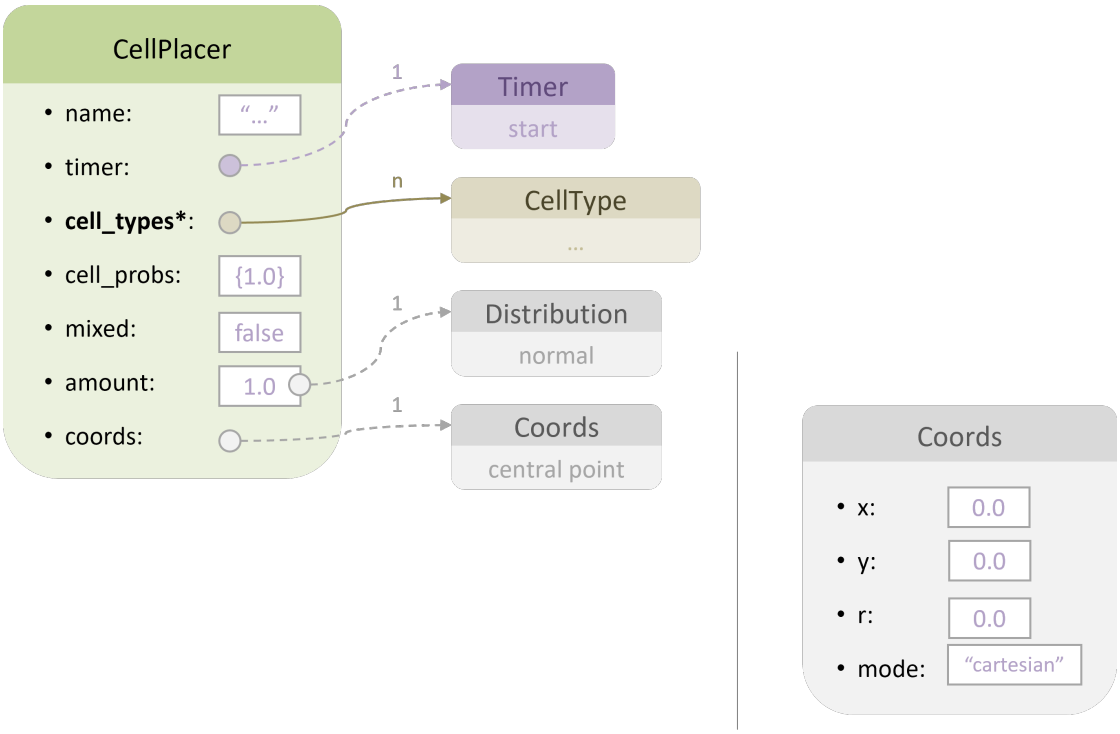

Figure 5.9: **CellPlacer element**

However, it is possible to provide a list of them for the CellPlacer to chose randomly. The `cell_probs` parameter is used to specify the probability of selecting each CellType. If it is left as default, then all the CellTypes will have the same probability of being selected. In the next example (Example 5.16) one of two alternative CellTypes is selected.

Example 5.16: CellPlacer with alternative CellTypes

```
bplasmid([ name := "bp_B" ]);
cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B", bplasmids := {"bp_B"} ]);

cell_placer([ name := "cp_simple"
  , cell_types := {"cell_A", "cell_B"}
  , cell_probs := {1.0, 3.0}
]);
```

`"cell_B"`, which carries a Plasmid that `"cell_A"` does not, has a selection probability

three times that of the `"cell_A"`. Notice that there is not need for the probabilities to add up to 1.0; they are automatically scaled if not. In this case, the processed probabilities are 0.25 and 0.75.

Number of cells The number of cells to place is controlled by the `amount` parameter. It can be a single value (deterministic) or an array of parameters that represent the mean and standard deviation of a normal distribution. By default, it is deterministic. Notice that the number of cells is a real parameter because it makes sense that way as the mean of the distribution. The sampled cell numbers are rounded to the nearest integer.

In the case of wanting the amount of placed cells to follow a distribution other than the normal, it can be specified using `amount_dist` instead. See Subsection 2.1.2 for more details about custom distributions. The Example 5.17 snippet shows three CellPlacers. `"cp_deterministic"` places exactly 100 cells every time it is executed. `"cp_normal"` samples the number of cells to place from a normal distribution with mean 100.0 and standard deviation 1.0 every time it is triggered. `"cp_uniform"` samples that amount from an uniform distribution with lower bound 90.0 and upper bound 110.0.

Example 5.17: CellPlacers with deterministic and stochastic cell numbers

```
cell_type([ name := "cell_A" ]);

cell_placer([ name := "cp_deterministic"
  , cell_types := {"cell_A"}
  , amount := 100.0
]);

cell_placer([ name := "cp_normal"
  , cell_types := {"cell_A"}
  , amount := {100.0, 1.0}
]);

cell_placer([ name := "cp_uniform"
  , cell_types := {"cell_A"}
  , amount_dist := [type := "uniform", params := { 90.0, 11
    0.0 }]
]);
```

Location The default location is the central point of the world. A different one can be selected using the `coords` parameter, which expects a dictionary with the coordinates. Just providing the x and y coordinates describes a point where all the cells will be placed, as in `"cp_point"` from Example 5.18. The cells are placed at (10.0, -10.0) in um. When multiple cells are placed at the same point, they are displaced by the shoving forces but placing many cells at the exact same stop may cause unwanted behaviours in the physics engine.

Example 5.18: CellPlacers with deterministic and stochastic location

```
cell_type([ name := "cell_A" ]);

cell_placer([ name := "cp_point"
  , cell_types := {"cell_A"}
  , amount := 100.0
  , coords := [ x := 10.0, y := -10.0 ]
]);

cell_placer([ name := "cp_circle"
  , cell_types := {"cell_A"}
  , amount := 100.0
  , coords := [ x := 10.0, y := -10.0, r := 100.0, mode :=
    "polar" ]
]);
```

In `"cp_circle"` a radius is given to describe a circle or radius 100.0 um centred at (10.0, -10.0). The cells are randomly placed within the circle. The fourth parameter, `mode`, determines how the cells are distributed within the circle. The default value is `"cartesian"`, which places the cells uniformly in the space. The alternative mode, `"polar"`, is used in the example to place a higher density of cells at the center of the circle and lower as we reach the border. The parameters of the coordinates dictionary are described in Table 5.14.

Mixed or not The `mixed` parameter is only relevant when more than one cell is placed. It is false by default, meaning that a single CellType is chosen for all the placed cells every time the CellPlacer is executed. If made true, a CellType is selected independently for each placed cell, obtaining a mix where the abundance of each CellType matches, on average, its assigned probability. Notice that in the case of `mixed` = false, the CellType is selected every time the placer is executed and not just once in the simulation. If the placer is linked to a periodic `Timer`, a mix of the CellTypes may be obtained anyway as

Table 5.14: Coordinates dictionary

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
mode	string in {"cartesian", "polar"}	—	How elements (Cells or Signals) will be distributed within the circle. Cartesian equals uniformly; in the polar mode, the density of elements is higher at the center than in the border.	"cartesian"
x	positive real	um	Position in x of the center of the circle	0.0
y	positive real	um	Position in y of the center of the circle	0.0
r	positive real	um	Radius of the circle. If 0, the element is a point and the placement of elements is deterministic.	0.0

it is executed several times.

Timing By default, the CellPlacer is executed once at the beginning of the simulation (at time 0.0). Any `Timer` can be assigned via the `"timer"` parameter to execute the placer periodically or conditionally.

5.4.2 CellPlating

Keywords: `cell_plating`
Recommended prefixes: `cpt_`
Linked to (direct): cell-level Gate, `Timer`
Linked to (reverse): `_`

Description While a CellPlacer creates new cells, the CellPlating element removes and/or relocates existing ones. This can be used to simulate cell plating, introduce perturbations or mix the cells to destroy the spatial structure.

Take and put fractions The `take` and `take_dist` parameters rule the fraction of cells that are taken from the world. As a fraction, `take` should be in the [0.0, 1.0] range. In the case of exactly 0.0 (or a negative number), no cells are taken; if exactly 1.0 or a greater number, all the cells are taken. In the intermediate cases, the cells to take are selected randomly. The default fraction is 0.9, meaning that only 10% of the existing cells are maintained.

Table 5.15: Parameters of the CellPlacer element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name	string	_	ID name for referencing the object	_
timer	Timer	ref.	The Timer that triggers the placer. Cells are placed every time the Timer goes off.	"_t_start" = once at the start
cell_types*	array of CellType	ref.	The CellTypes to be placed. If more than one provided, one is selected randomly every time	_
cell_probs / cells_probs	array of positive reals, ideally in [0.0, 1.0] (scaled if not)	prob.	The probability of selecting each type of cell. Only applies when more than one CellType is given.	{} = all the types have equal probability
mixed / mix	Boolean	_	If true, the different CellTypes are mixed in a proportion given by their probabilities. If false, a unique CellType is randomly selected every time.	false
amount / amount_params / amount_dist	positive real, array of real, Distribution or dictionary	cells	Number of cells to be placed every time. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution.Real numbers are used as it may be stochastic (the sampled numbers are then rounded).	1.0 = a single cell
coords	Coords dictionary	ref.	Coordinates of the circle where the cells are placed. If the radius is not 0, the cells are randomly placed withing the circle.	At the center (0.0,0.0) with radius 0.0 and cartesian mode.

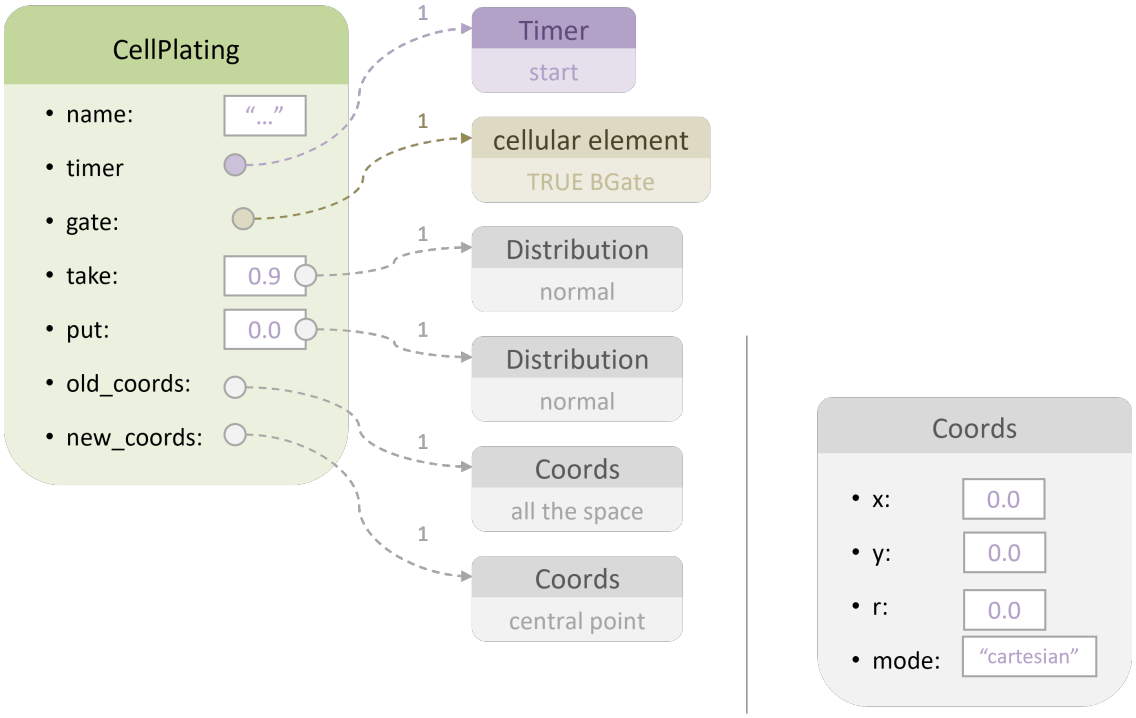

Figure 5.10: **CellPlating element**

By default, `take` is 0.9 and deterministic. If an array is assigned, the taken fraction is sampled from a normal distribution that uses these values as mean and standard deviation.

In the case of wanting the fraction to follow a distribution other than the normal, it can be specified using `take_dist` instead. See Subsection 2.1.2 for more details about custom distributions or Example 5.17 about the `amount_dist` parameter of CellPlacer (Subsection 5.4.1).

`put` and `put_dist` work in the exact same way. The difference is that they specify the fraction of cells to return to the simulated world (from those previously taken). The returned cells are relocated as their original position in the world is replaced by a new one. By combining the take and put sets of parameters, the CellPlating element can be used to just remove cells, to just relocate cells or to remove some cells and relocate others at the same time. By default, `put` is 0.0, what means that all the taken cells are removed and none is replaced. At the other extreme, a put fraction of 1.0 (or bigger) produces that all the taken cells are relocated and none is removed. For intermediate values, the concrete cells that are returned are selected randomly from the taken ones.

In Example 5.19, 100 cells are placed. Then a fraction of 0.8 is taken. That is 80

cells. A fraction of 0.25 is then returned. The result is that 20 cells are relocated, 60 cells are removed and 20 cells remain in their original position.

Example 5.19: CellPlating for removing and relocating cells

```
cell_type([ name := "cell_A" ]);
cell_placer([ name := "cp_A", cell_types := {"cell_A"},
   amount := 100.0 ]);

cell_plating([ name := "cpt_simple"
  , take := 0.8
  , put := 0.25
]);
```

New location The default location is the central point of the world. A different one can be selected using the `coords` parameter, which is identical to that of the CellPlacer element explained at Subsection 5.4.1. The parameters of the coordinates dictionary are described in Table 5.14.

Spatial filter An additional coordinates dictionary can be assigned to `old_coords` parameter as a filter for the taken cells. Only the cells within the defined circle are affected by the CellPlating element. In this case, the `mode` parameter of the coordinates dictionary is ignored.

Gate filter It is also possible to filter the cells by their internal state, using a cell-level Gate or any other cellular element assigned to the `gate` parameter. Only the cells where the condition/presence is true are affected by the CellPlating element. If both `old_coords` and `gate` are used, both conditions must hold for the cells to be affected. The number of cells to take is calculated using only the eligible cells and not all the cells.

In **??**, the CellPlating only affects the cells of the `"cell_A"` CellType (due to the BGate) that are within a radius 100 um from the center (due to the coordinates dictionary).

Example 5.20: Conditional CellPlating

```
cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B", bplasmids := {"bp_B"} ]);
```

```
cell_plating([ name := "cpt_A"
  , gate := "cell_A"
  , old_coords := [ x := 0.0, y := 0.0, r := 100.0 ]
]);
```

Timing By default, the CellPlacer is executed once at the beginning of the simulation (at time 0.0). There may be a use case where this is the desired behaviour as CellPlatings are always executed after CellPlacers. However, the timing is expected to be replaced by a custom one in most cases. Any `Timer` can be assigned via the `"timer"` parameter to execute the placer periodically or conditionally.

The CellPlating in Example 5.21 removes 90% of the cells every time the total cell number reaches 1000, so that the population oscillates between 100 and 1000. To do so, the predefined `"_stat_cellNum"` PopulationStat, which tracks the total size of the population, is used.

Example 5.21: CellPlating driven by cell number

```
pop_qgate([ name := "qpga_cellNum"
  , input := "_stat_cellNum", value := 1000.0 ]);
timer([ name := "t_cellNum", gate := "qpga_cellNum" ]);

cell_plating([ name := "cpt_periodic"
  , timer := "t_cellNum"
]);
```

5.4.3 SignalPlacer

Keywords: `signal_placer`
Recommended prefixes: `sp_`
Linked to (direct): Signal, `Timer`
Linked to (reverse): _

Description The SignalPlacer modifies the composition of the medium by adding or removing Signals. It is analogous to the CellPlacer, so that most of its parameters have been described at section (Subsection 5.4.1)

Table 5.16: Parameters of the CellPlating element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name	string	_	ID name for referencing the object	_
timer	Timer	ref.	The Timer that triggers the placer. Cells are placed every time the Timer goes off.	"_t_start" = once at the start
gate	any cellular element, usually Gate	ref.	The replating event only affects the cells where the condition evaluates to true	default TRUE BGate
take / take_params / take_dist	positive realin [0.0, 1.0], array of real, Distribution or dictionary	fraction of the cells	Fraction of the cells to be taken every time from those where the condition holds. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution.	0.9
put / put_params / put_dist	positive realin [0.0, 1.0], array of real, Distribution or dictionary	fraction of the cells	Fraction of the taken cells to be returned (and relocated) to the world. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution.	0.0
old_coords	Coords dictionary	ref.	Coordinates of the circle where the cells are taken. Only the cells within the circle are affected. If the radius is not 0, the cells are randomly placed withing the circle. The mode parameter has no effect.	At the center (0.0,0.0) with infinite radious = not spatial condition
coords / new_coords	Coords dictionary	ref.	Coordinates of the circle where the cells to put back are placed. If the radius is not 0, the cells are randomly placed withing the circle.	At the center (0.0,0.0) with radius 0.0 and cartesian mode

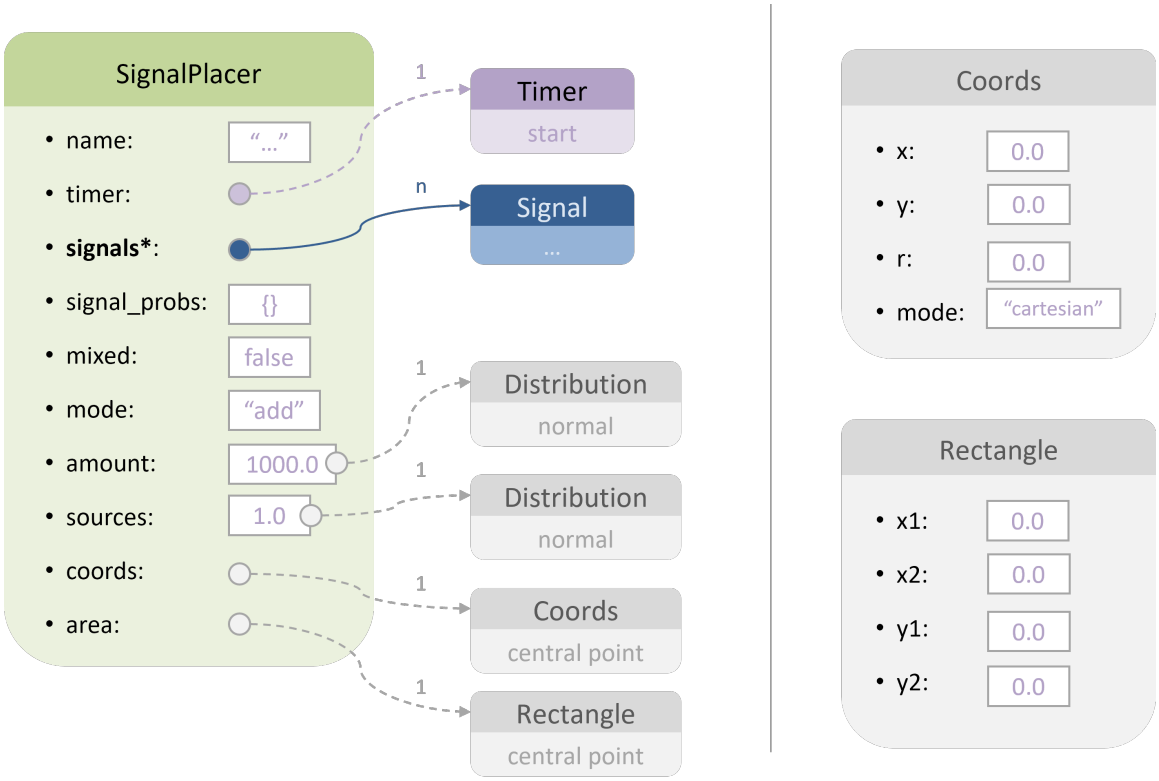

Figure 5.11: **SignalPlacer element**

Signal types `signals` is the only mandatory parameter. It is necessary to stablish which Signal or Signals will be placed. In the simplest case, a single Signal is given. This case is shown in the following example (Example 5.22).

Example 5.22: Simplest SignalPlating

```
signal([ name := "s_A" ]);

signal_placer([ name := "sp_A"
  , signals := { "s_A" }
]);
```

This code, with all the optional parameters as default, places an amount of 1.0 (user-defined units) of the specified Signal at the center of the world when the simulation starts (time 0.0). Notice that, even if a single Signal is given, it has to be done as a list using the curly brackets.

However, it is possible to provide a list of them for the SignalPlacer to chose randomly. The `signal_probs` parameter is used to specify the probability of selecting each Signal. If it is left as default, then all the Signals will have the same probability of being selected. In the next example (Example 5.23) one of two alternative Signals is selected.

`"s_B"` has a selection probability 9 times that of the `"s_A"`. Notice that there is not need for the probabilities to add up to 1.0; they are automatically scaled if not. In this case, the processed probabilities are 0.1 and 0.9.

Mode There are two possible modes, assigned through the `mode` parameter. The default mode is `"add"`, which means adding more signal to the amount that may already exist at the location. If the amount of signal to place is negative (see `amount` parameter below), it is subtracted from the existing amount. The alternative mode is `"set"`, which overrides the existing amount of Signal at the location with the new value. When used with an amount of 0.0, this mode completely removes the Signal.

Number of sources The Signal can be placed at a single spot (source) or at several ones. The number of sources is ruled by two parameters: `sources` and `sources_dist`. They work analogously to other similar sets of parameters previously described for CellPlacer and CellPlating. `sources` is either the deterministic value or the parameters of a normal distribution from which the number of sources is sampled every time the SignalPlacer is executed. By default, it is deterministic and equal to 1.0. Notice that `sources` expects a real number and not an integer. The sampled numbers are rounded to the nearest integer.

In the case of wanting a distribution other than the normal, it can be specified using `sources_dist` instead. See Subsection 2.1.2 for more details about custom distributions or Example 5.17 about the `amount_dist` parameter of CellPlacer (Subsection 5.4.1).

Location The default location is the central point of the world. A different one can be selected using the `coords` parameter, which expects a dictionary with the coordinates, exactly as the parameters with the same name in CellPlacer and CellPlating. Just providing the x and y coordinates describes a deterministic point where all the sources are placed. When multiple sources are placed at the same point, they are either added up if the mode is "add" or the last to be placed overrides all the others in "set" mode. As the medium Signals are stored in a discrete diffusion grid, the continuous coordinates in um are rounded to select the closest grid unit.

In `"sp_circle"` a radius is given to describe a circle or radius 100.0 um centred at (200.0, 0.0). 10 sources are randomly placed within the circle. The fourth parameter, `mode`, determines how the sources are distributed within the circle, exactly as in CellPlacer

(see Subsection 5.4.1). The parameters of the coordinates dictionary are described in Table 5.14.

The example in Example 5.23 and Figure 5.12 shows the effect of several parameters. 10 source spots are placed, randomly selecting a Signal for each one as the `mixed` parameters (see below) indicates. They are placed within a circle of radius 200 um (see below). In the particular execution shown in the figure, the proportion of Signal sources matches the given probabilities exactly by chance.

Example 5.23: SignalPlating with alternative Signals

```
signal([ name := "s_A" ]);
signal([ name := "s_B" ]);

signal_placer([ name := "sp_AB"
  , signals := { "s_A", "s_B" }
  , signal_probs := { 1.0, 9.0 }
  , mixed := true
  , sources := 10.0
  , coords := [ r := 200.0 ]
]);
```

Amount of signal Two parameters determine the amount of signal placed at every source spot: `amount` and `amount_dist`. They analogous to the three parameters that control the number of sources (see above). `amount` is either a deterministic value or the mean and standard deviation of a normal distribution from which the amount of signal is sampled. In the case of wanting a distribution other than the normal, it can be specified using `amount_dist` instead. A new amount is sampled per source and time the SignalPlacer is executed. That means that a different amount may be placed at each source spot. The default amount is 1000 user-defined units. See REF TO SIGNAL for a discussion on amount and concentration units.

The code in Example 5.24 produces 10 source spots. In this case, all of them are of the same Signal but they differ in the placed amount, which follows a uniform distribution in [10, 10000].

Example 5.24: SignalPlating with stochastic amount

```
signal([ name := "s_A" ]);

signal_placer([ name := "sp_A"
```

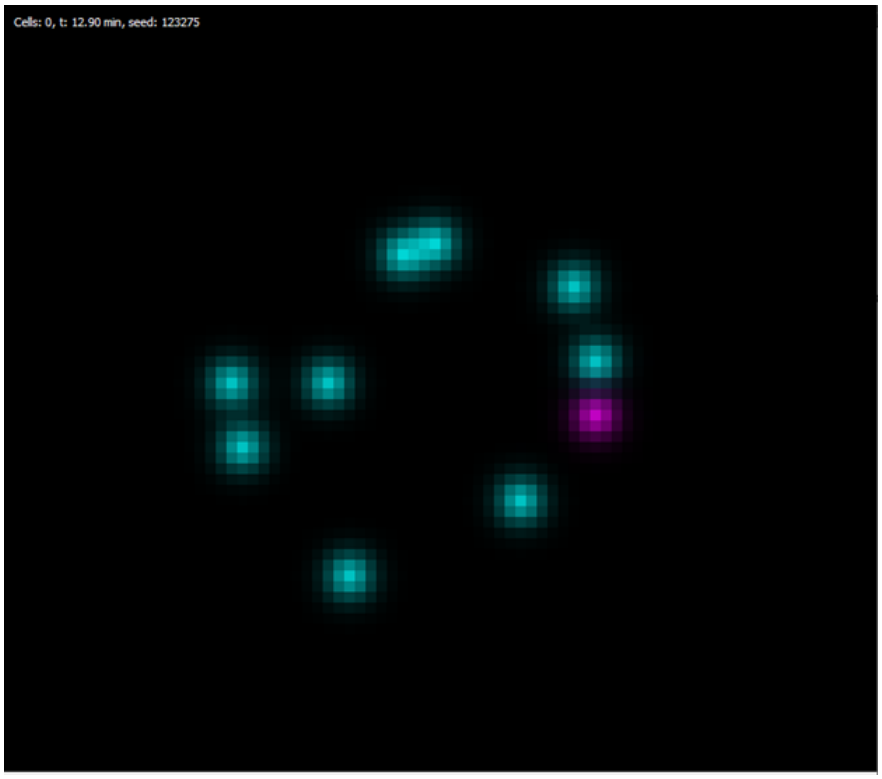

Figure 5.12: **SignalPlacer with alternative Signals** A Signal is randomly chosen for each source spot using user-defined probabilities.

```
  , signals := { "s_A" }
  , sources := 10.0
  , amount_dist := [ type := "uniform", params = { 10.0, 10
    000.0 } ]
  , coords := [ r := 200.0 ]
]);
```

Mixed or not Analogously to the parameter with the same name in CellPlacer, `mixed` is only relevant when more than one source spot. It is false by default, meaning that a single Signal is chosen for all the sources every time the SignalPlacer is executed. If made true, a Signal is selected independently for each spot, obtaining a mix where the abundance of each Signal matches, on average, its assigned probability.

Timing By default, the SignalPlacer is executed once at the beginning of the simulation (at time 0.0). As with the other types of placer, any custom `Timer` can be assigned via the `"timer"` parameter to execute the placer periodically or conditionally.

5.5 Output elements

5.5.1 Snapshot

Keywords: `snapshot`
Recommended prefixes: `img_`
Linked to (direct): `Timer`
Linked to (reverse): `_`

Description This element automatically takes snapshots and saves them as image files. Snapshots can be also taken manually using the GUI.

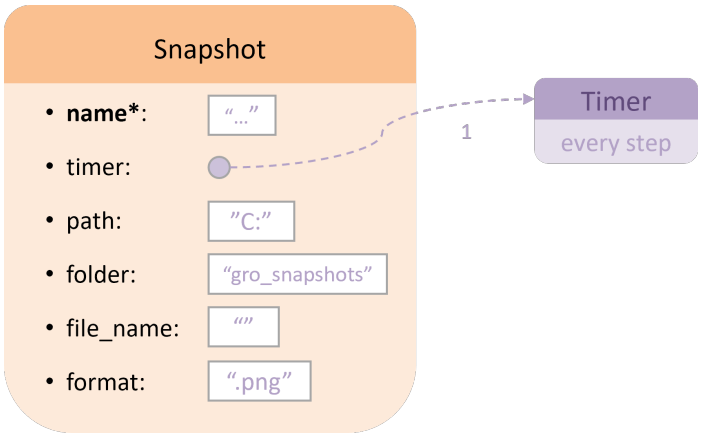

Figure 5.13: **Snapshot element**

File name and path The saved image files can be given a custom name through the `file_name` parameter. As a single Snapshot element may take several snapshots during a simulation, a number (starting at 0) is added to the custom name to indicate the order in which the snapshots were taken. If the default name is not overridden, the files are named with just the numbers.

Table 5.17: Parameters of the SignalPlacer element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name	string	_	ID name for referencing the object	_
timer	Timer	ref.	The Timer that triggers the placer. Signals are placed every time the Timer goes off.	"_t_start" = once at the start
signals*	array of Signal	ref.	The Signals to be placed. If more than one provided, one of them is selected randomly every time.	_
signal_probs / signals_probs	array of reals, ideally in [0.0, 1.0] (scaled if not)	prob.	The probabilities of selecting each signal. Only applies when more than one Signal is given.	{} = all the signals have equal probability
mixed / mix	Boolean	_	If true, the different Signals are mixed in a proportion given by their probabilities. If false, a unique Signal is randomly selected every time.	false
mode	string in {"set", "add"}	_	"add" mode adds (or substracts if negative) the amount of signal to the existing amount in the grid cell. "set" mode overwrites the grid cell with the given amount.	"add"
amount / amount_param / amount_dist	real, array of real, Distribution or dictionary	custom concentration units	Amount of signal to to be placed at every source spot. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution.Negative numbers substract signal and only make sense in "add" mode.	1000.0
sources / sources_params / sources_dist	positive real, array of real, Distribution or dictionary	source spots	Number of signal source spots to be placed every time. The amount of signal is added at every source spot. Either a single deterministic one, mean and standard deviation of a normal distribution or a custom distribution. Real numbers are used as it may be stochastic. If negative, there are not spot and the signal is placed at every grid cell of the circle. Only used in "add" and "set" modes	1.0 = a single spot
coords	Coords or direct Coords dictionary	ref.	Coordinates of the circle where the signals are placed. If the radius is not 0, the signals are randomly placed withing the circle. Only used in "add" and "set" modes	At the center (0.0,0.0) with radius 0.0 and cartesian mode.
area	Rectangle dictionary	ref.	Rectangular area where the signal is set in "full" mode	At the center (0.0,0.0)

The path where the images are saved is specified using two parameters: `path` and `folder`. This two parameters are concatenated to create the actual path. Using two separate fields is convenient as the user may want to use the same upper-level `path` for all the Snapshot and other output elements in the simulation while creating a dedicated `folder` inside for each of them. By default, the images are saved at `"C:/gro_snapshots"`. If the folder does not exist, it is created. Notice that running gro as an administrator may be necessary to save files at the default path.

Image format There are three supported formats for the images: `".png"`, `".jpg"`, `".bmp"`, `".tif"`and `".tiff"`. This is modified at the `format` parameter, which outputs .png files by default.

Timing By default, the Snapshot is executed every single step, something that may slow down the simulation and clutter the disc. Any custom `Timer` can be assigned via the `"timer"` parameter to execute the element periodically or conditionally.

The code in Example 5.25 produces periodic snapshots every 10 minutes, starting at time 100 minutes. They are saved at `"C:/gro/output/gro_images"`, and named "snap<number>.bmp" (see Figure 5.14).

Example 5.25: Periodic Snapshot

```
timer([ name := "t_snap", start := 100.0, period := 10.0 ])
   ;

snapshot([ name := "img_A"
  , timer := "t_snap"
  , path := "C:/gro/output"
  , file_name := "snap"
  , format := ".bmp"
]);
```

5.5.2 Output file (OutFile)

Keywords: `out_file cells_file`
Recommended prefixes: `file_ out_`
Linked to (direct): any cell-level or population-level element, `Timer`
Linked to (reverse): `_`

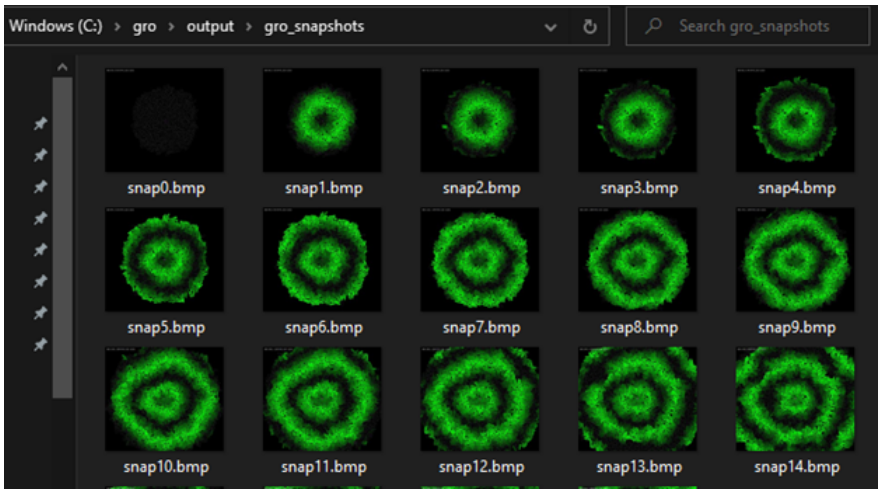

Figure 5.14: **Snapshot result.** Images are saved at the specified folder.

Description The OutFile element saves text files with the results of the simulation. The files are completely customizable and can include the internal state of individual cells and/or population-level statistics.

Individual vs population mode The element is able to create two types of files, using the `mode` parameter, with these possible values:

- `"population"`. Produces a single file per simulation. This includes population-level information (i.e. aggregated fields). Every time the OutFile is triggered, a new entry (line) is added to the file, starting with the time when it was taken as the first field.
- `"individual"`. Produces several files, one per each time the OutFile is executed. Each file contains information about the state of individual cells at a specific point of time. Each entry (line) in the files correspond to a cell and it starts with a unique ID of the cell as the first field.
- `"both"`. Produces both a population-level file and the individual-level files. This is the default mode.

Ancestry If the `ancestry` option is set to true, then information about the origin and history of every cell is included in the "individual" files. This information includes the unique numerical IDs of their mother and sibling (on division, the mother cell disappears and two daughter cells with new IDs appear). The ID of its original ancestor

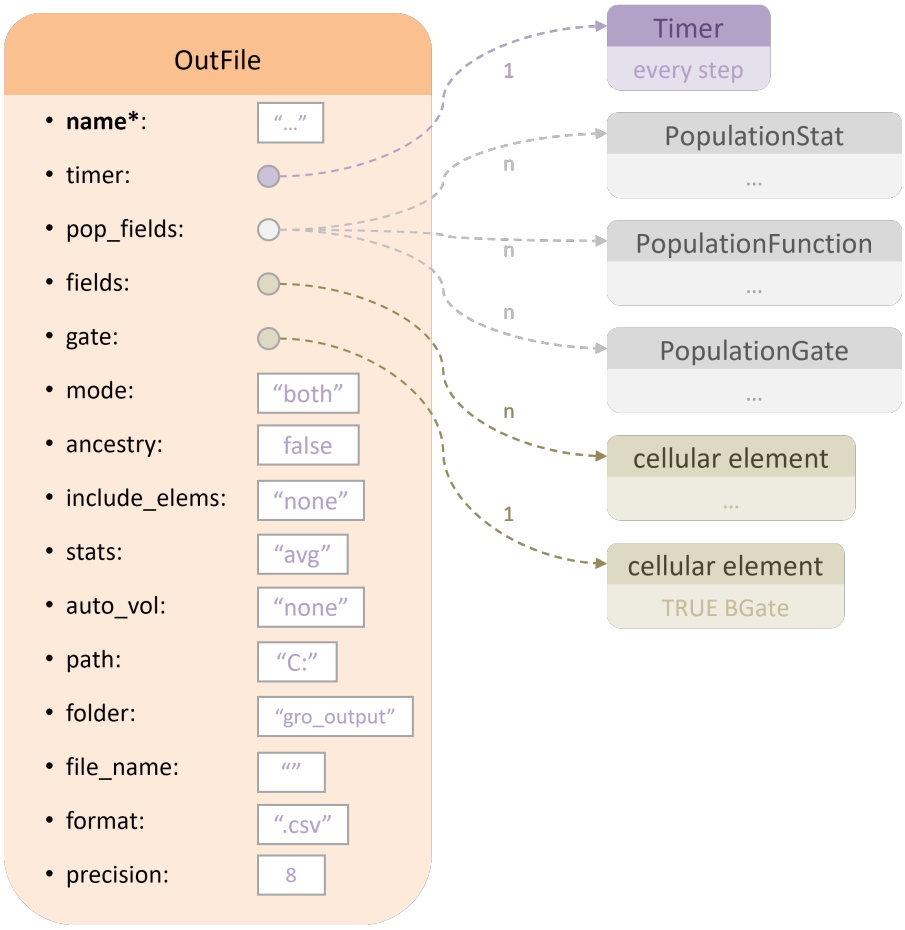

Figure 5.15: **OutFile element**

Table 5.18: Parameters of the Snapshot element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the object	_
timer	Timer	ref.	The Timer that triggers the snapshots. An snapshot is taken every time the Timer goes off.	"_t_every_step" = every time step
path	string	_	The path where the images are saved.	"C:"
folder	string	_	Name of the directory where the images are saved. The total path is the concatenation of "path" and "folder". If it does not exist, it is created.	"gro_snapshots"
file_name	string	_	Base name of the snapshot files. The total file name is made by adding a number that indicates the ordering.	"" = just the number
format	string in {".png",".jpg",".bmp", ".tif", ".tiff"}	_	Image file format	".png"

(placed with CellPlacer) is also included, as well as the generation number, considering the original one as 0. Finally, a Boolean chain of the cell division history from the ancestor is also included. The chain is inherited from the mother cell and, on division, one daughter cell adds a 0 to the chain while the other adds a 1. The degree of vertical relation between two cells equals the number of digits that match starting from the left. This is useful to study cell lineages and how some parameters have changed/evolved.

Population-level fields The elements that can be included as population-level fields are all the logic population-level elements described in previous sections: PopulationStat, PopulationFunction, PopulationBGate, PopulationQGate. An unbound number of them can be provided via the `pop_fields` list parameter. They will appear in the output file as columns in the same order they were provided. Due to their nature, these fields only appear at the population-level file and not at the individual ones. Consequently, if the selected `mode` is `"individual"`, this parameter is ignored.

Individual-level fields and automatic statistics The individual-level fields are provided via the `fields` list parameter. An unbound number of any cellular element, either logic or biological, is expected. This includes Functions, Gates, Odes, Molecules, Plasmids, CellTypes...among others. The special fields used to get the position, size and growth of a cell are also valid here. The complete list of them can be consulted

at Subsection 5.2.1. These elements will be included in the individual files, in the provided order. Their individual values for each cell will be registered there. The code in Example 5.26 produces a file every time step with two fields: the cell ID and `"mol_A"`. The state of that Molecule is printed for every cell in the simulation.

Example 5.26: Simple OutFile

```
molecule([ name := "mol_A" ]);

out_file([ name := "file_simple"
  , fields := {"mol_A"}
  , mode := "individual"
]);
```

The parameter `include_elems` can be used to automatically include all the elements of given types to avoid having to type all of them. The default mode is `"none"`, meaning that no elements are automatically added. The `"classic"` mode includes all the elements of these types: CellType, Molecule, Plasmid (both BPlasmid and Qplasmid). The `"all"` mode includes all the types of element with the exception of some of them that do not store their state or whose state is not relevant from the user point of view: Randomness, Pilus, Mutation, MutationProcess. If the number of cellular elements in the simulation is very high, using the `"all"` setting may slow it down noticeably.

The snippet in Example 5.27 and its result in figure Figure 5.16 exemplifies the `"classic"` mode. The CellTypes, Molecules and Plasmids are automatically included but not the Operons. That it why `"op_A"` has to be included explicitly.

Example 5.27: Automatic fields in OutFile

```
molecule([ name := "mol_A" ]);
molecule([ name := "mol_B" ]);
operon([ name := "op_A" ]);
bplasmid([ name := "bp_A", operons := {"op_A"} ]);
cell_type([ name := "cell_A", molecules := {"mol_A"},
   bplasmids := {"bp_A"} ]);
cell_type([ name := "cell_B", molecules := {"mol_B"} ]);

out_file([ name := "file_auto"
  , fields := {"op_A"}
  , mode := "individual"
  , include_elems := "classic"
```

```
]);
```

	A	B	C	D	E	F	G
1	id	op_A	cell_A	cell_B	bp_A	mol_A	mol_B
2	0	1	1	0	1	1	0
3	1	1	1	0	1	1	0
4	2	1	1	0	1	1	0
5	3	1	1	0	1	1	0
6	4	1	1	0	1	1	0
7	5	1	1	0	1	1	0
8	6	1	1	0	1	1	0
9	7	1	1	0	1	1	0
10	8	1	1	0	1	1	0

Figure 5.16: **Individual OutFile.** Each entry is a cell.

The parameter `gate` can be assigned a cell-level Gate or other element to act as a filter. Only the cells where the condition evaluates to true are included in the input. This can be used to select only the cells of interest for the output (more efficient). The `auto_vol` option is available to automatically convert amounts to concentrations and vice versa for all the individual-level fields.

These fields will also be included in the population-level file, after the `pop_fields`. To do so, a PopStat will be automatically created behind the scene for each of the elements. Two additional parameters are used to control how these PopStats are created: `gate` and `stats`. They match the `gate` and `stats` parameters of PopulationStat, described in section Subsection 5.2.1. The same `gate` is used to filter both the cells to include in the individual files and the ones to to aggregate using the `stats` statistics. Same as in PopulationStat, `stats` can be either a list of concrete statistics or a keyword indicating a group of them. All the automatically created PopulationStats will share these parameter values. In order to use statistics with different values, the user should manually create some PopStats or split the results into several population-level files, using several OutFile elements.

The Example 5.28 shows how the same result may be achieved either with the automatically generation of PopulationStats or manually created ones. `"file_1"` uses an externally created PopulationStat. The average (because that is the default statistic) of the element `"mol_A"` will be printed. As the nature of this element is Boolean, the calculated average matches the fraction of the cells where the Molecule is present. This approach allow for PopulationStats with different values for `gate` and `stats` to be

included in the same file. As there are no cell-level `fields` but the default `mode` is `"both"`, empty individual files will be generated. `"file_2"` uses `"mol_A"` as a cell-level field. As the `mode` is `"individual"`, this element only produces individual files, where the state of `"mol_A"` is printed for every cell of `"cell_A"` type (due to the `gate`). `"file_3"` produces the same output than `"file_1"` and `"file_2"` together. Besides printing the individual states of `"mol_A"`, because the `mode` is `"both"`, it also creates a PopulationStat equivalent to `"stat_A"` and prints it to the population file, which would look something like Figure 5.17.

Example 5.28: Alternative descriptions of an OutFile

```
cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B" ]);
molecule([ name := "mol_A" ]);

pop_stat([ name := "stat_A", gate := "cell_A", input := "
   mol_A" ]);

out_file([ name := "file_1"
  , pop_fields := {"stat_A"}
  , file_name := "out1"
]);

out_file([ name := "file_2"
  , fields := {"mol_A"}
  , gate := "cell_A"
  , mode := "individual"
  , file_name := "out2"
]);

out_file([ name := "file_3"
  , fields := {"mol_A"}
  , gate := "cell_A"
  , file_name := "out3"
]);
```

File name and path Analogously to the Snapshot element, saved files can be given a custom name through the `file_name` parameter. This is the name assigned to the single population-level file. Individual-level files get a number appended to the name,

	A	B
1	time	mol_A_avg
2	0	0.5
3	0.1	0.5
4	0.2	0.5
5	0.3	0.5
6	0.4	0.5
7	0.5	0.5
8	0.6	0.5
9	0.7	0.5
10	0.8	0.5

Figure 5.17: **Population OutFile.** Each entry is a time step.

indicating the time step. The time in minutes is typically 0.1 times the time step. In case the user modifies the time step size (not recommended), the new one should be manually recorded in order to make the conversion in the future. If the default name is not overridden, the individual files are named with just the numbers and the population file has no name.

The path where the images are saved is specified using two parameters: `path` and `folder`, as in the Snapshot element. By default, the files are saved at `"C:/gro_output"`. If the folder does not exist, it is created. Notice that running gro as an administrator may be necessary to save files at the default path.

File format and precision There are two supported formats: `".csv"` (better for automatic data processing) and `".tsv"` (better for visualization). This is modified at the `format` parameter, which outputs .csv files by default.

The `precision` parameter determines the formatting of real numbers, more specifically, the number of significant decimal places printed, 8 by default. Notice here the difference between significant places and fixed decimal places. It also affects the whole part (caution with big numbers like cell numbers) although whole numbers are printed without any decimals.

Timing By default, the OutFile is executed every single step, something that may slow down the simulation and clutter the disc. Any custom `Timer` can be assigned via the `"timer"` parameter to execute the element periodically or conditionally.

5.5.3 Plot

Keywords: `plot`
Recommended prefixes: `plot_ pt_`

Table 5.19: Parameters of the OutFile element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the object	_
timer	Timer	ref.	The Timer that triggers the output. An entry is appended to the file every time the Timer goes off.	"_t_every_step" = every time step
pop_fields	array of Pop Stat, individual Pop Stat, Pop Function and Pop Gate	ref.	Global population-level fields to include in the output file. For Pop Stats, all of their individual stats are included; to include a specific one, use the name of the individual stat.	{} = no elements
fields / cell_fields	array of any cellular element	ref.	Cellular-level fields to include in the output file. As they have to be aggregated for population files, a Pop Stat with the stats specified by the "stats" parameter is automatically created.	{} = no elements
gate / filter_gate	any cellular element, usually Gate	ref.	Applies to the cellular fields. Only the cells where this condition evaluates to true are included in the individual file and aggregated for the population file.	TRUE BGate
include_elems	string in {"none", "classic", "all"}	_	Automatically adds cellular elements to the "fields" parameter. "classic" = CellType, Plasmid and Molecule. "all" = every type excluding Randomness, Pilus, Mutation and MutationProcess	"none"
stats	array of strings in {"sum", "avg", "stdedv", "min", "max"} or string in {"avg", "classic", "range", "all"}	—	The aggregation statistics to compute: summation, average, standard deviation, minimum, maximum. Applies to the cellular fields. "classic" = {"sum","avg","stddev"}; "range" = {"min","max"}	"avg" = the average
auto_vol	string in {"none","product","division"}	—	Used to automatically multiply or divide all the "fields" by the celllular volume to convert between amounts and concentrations.	"none" = nothing done
mode	string in {"individual", "population", "both"}	—	"individual" = the state of every cell, a separate file every time with the cells identified by their unique ID; "population" = a single file with aggregation statistics, a line for each time; "both" = the two of them"	"both"
ancestry	Boolean	_	If true, ancestry data is included in individual files. This includes the IDs of original, mother and sibling cells and the division history.	false
path	string	_	The path to the directory where the files is saved	"C:"
folder	string	_	Name of the directory where the files are saved. The total path is the concatenation of "path" and "folder". If it does not exist, it is created.	"gro_output"
file_name	string	_	Base name of the files. This is the name of population files. The name of individual files is created by adding the step number.	the name of the element
format	string	_	File format	".csv"
precision	positive integer	—	Number of decimal places for formatting real numbers	8

Linked to (direct): any cell-level or population-level element, `Timer`
Linked to (reverse): _

Description This output element produces real time line plots of population-level metrics. The evolution in time of any element can be reported.

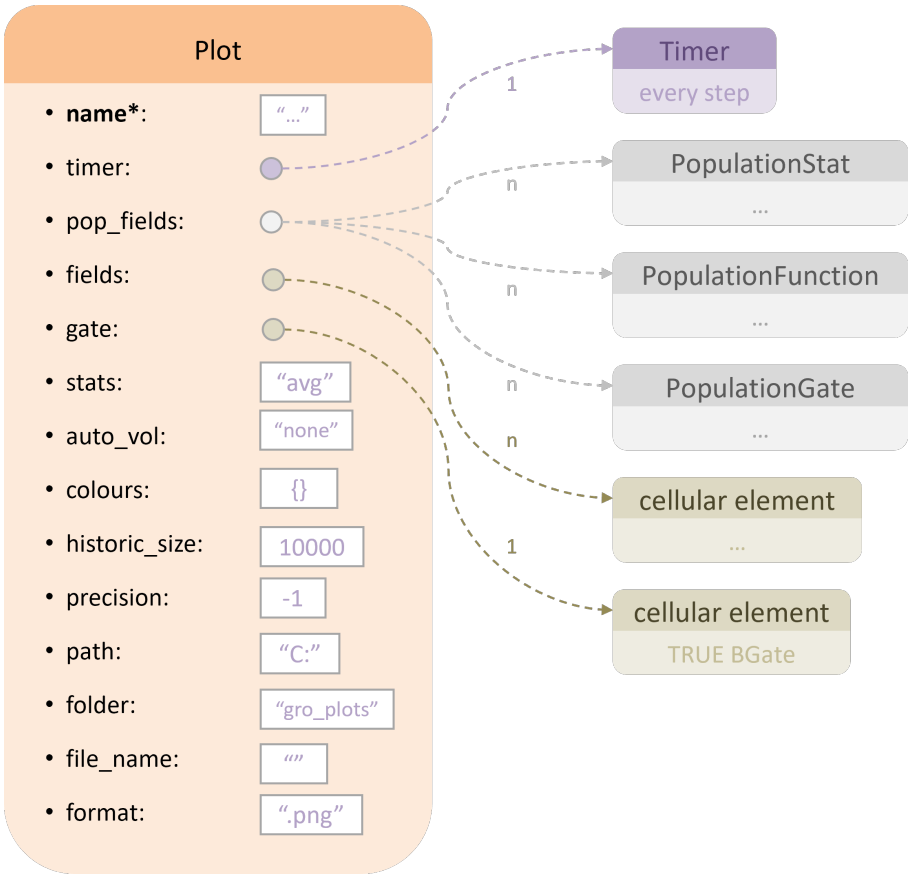

Figure 5.18: **Plot element**

In common with OutFile The Plot element is for the biggest part equivalent to the OutFile element described in the previous section (Subsection 5.5.2) so that only the differential parameters will be covered here in detail. If a OutFile can include several fields as columns of the same file, a Plot can include several fields as lines plotted in the same chart.

Example 5.29: Alternative descriptions of an the same Plot

```
global_params([ seed := 1234, show_plots := true ]);

cell_type([ name := "cell_A" ]);
cell_type([ name := "cell_B" ]);
molecule([ name := "mol_A" ]);

pop_stat([ name := "stat_A", gate := "cell_A", input := "
   mol_A" ]);

plot([ name := "plot_1"
  , pop_fields := {"stat_A"}
]);

plot([ name := "plot_1"
  , gate := "cell_A"
  , fields := {"mol_A"}
]);
```

The simulations admit a maximum of four Plot elements. A maximum of four charts are shown in a separate window that can be shown and hidden using the GUI. The initial visibility state of that window can be established using the `show_plots` global parameter. By default, the plots window is hidden. If more than four Plot elements are created, the first four are selected alphabetically and the rest are ignored.

Exactly as in OutFile, both population-level and cell-level elements can be assigned, via the `"pop_fields"` and `"fields"` parameters respectively. The difference with OutFile is that the mode of a Plot is always `"population"` so that automatic PopulationStats are always created for the cell-level elements. Again, as in OutFile, the special fields for position, size and growth can be used. The behaviour of `gate` and `stats` is identical to their homonyms at OutFile. The user should be cautious with automatically creating fields and statistics (a complete factorial is performed) as the maximum number of plotted lines per Plot element is 8. Any extra element will be ignored. The `precision` parameter in this case is set to an unbound number of decimal places (no rounding). In some cases, rounding the values may increase the clarity of the plot as it gets rid of small uninformative oscillations (some times caused by floating point error).

The Plot can be assigned a `Timer` but this does not rule the refresh time of the plots, as they are always updated every time step to offer real time data. The `Timer` controls when images of the assigned chart are stored. If no `Timer` is assigned, no images are stored by default. The exact time when the pictures are taken is approximated rather

than the exact time stated by the timer due to the parallel and asynchronous update of the plots for efficiency. It is guaranteed that the saving time is not before the timer (only exactly at the time step or a few time steps after). Precision here is not required as these plots are not for formal purposes.

The `path`, `folder`, `file_name` and `format` are equivalent to those from the Snapshot element and are only used when a `Timer` is assigned.

Appearance The plots are interactive. The user can zoom in/out and drag them whenever the simulation is paused. The number of parameters for the appearance is minimal as the axes are automatically adjusted to fit the data as new points are added. A legend is also automatically generated. The title of the chart matches the name of the Plot element (and not `file_name`). The typical appearance of the plots windows is shown in Figure 5.20

The `colours` parameter expects a list of hexadecimal codes for colours, corresponding to the colours of each of the lines (maximum of 8). The default colour palette is shown in Figure 5.19. They are assigned from left to right. In case the number of lines to plot is higher than the number of colours, the list is completed with colours from the default palette.

As the computational cost increases with the number of data points in the plots, they are periodically cleared. The `historic_size` parameters determines the maximum number of data points per plot (per line). When that number is reached, the plot is cleared and a new plot is started with the next data point. The default number is 10000, meaning that the plots are cleared every 10000 steps or (assuming a time step size of 0.1 min) 1000 minutes. The lower this number, the higher the performance.

The aim of the plots is to provide fast visual information to the user. To create high quality publication-level charts, the user must create them once the simulation is finished using the text files saved by OutFile instead. A high number of Plot elements and total plotted lines can seriously impact the performance.

Figure 5.19: **Default palette.** Used for the plots

Table 5.20: Parameters of the Plot element

FIELD	DATA TYPE	UNITS	DESCRIPTION	DEFAULT
name*	string	_	ID name for referencing the object	_
timer	Timer	ref.	The Timer that triggers the output. An entry is appended to the file every time the Timer goes off.	"_t_every_step" = every time step
pop_fields	array of PopStat, individual PopStat, PopFunction and PopGate	ref.	Global population-level fields to include in the output file. For Pop Stats, all of their individual stats are included; to include a specific one, use the name of the individual stat. The maximum fields ("pop_fields" + "fields") per chart is 8; extra fields are ignored.	{} = no elements
fields / cell_fields	array of any cellular element	ref.	Cellular-level fields to include in the output file. As they have to be aggregated for population files, a Pop Stat with the stats specified by the "stats" parameter is automatically created. The maximum fields ("pop_fields" + "fields") per chart is 8; extra fields are ignored.	{} = no elements
gate / filter_gate	any cellular element, usually Gate	ref.	Applies to the cellular fields. Only the cells where this condition evaluates to true are included in the individual file and aggregated for the population file.	TRUE BGate
stats	array of strings in {"sum", "avg", "stdedv", "min", "max"} or string in {"avg", "classic", "range", "all"}	—	The aggregation statistics to compute: summation, average, standard deviation, minimum, maximum. Applies to the cellular fields. "classic" = {"sum","avg","stddev"};"range" = {"min","max"}	"avg" = the average
auto_vol	string in {"none","prod	—	Used to automatically multiply or divide all the "fields" by the celllular volume to convert between amounts and concentrations.	"none" = nothing done
colours / colors	Array of strings (hexadecimal colours)	_	Colours used to plot the fields. If more colours than fields are provided, any extra colours are ignored; if not enough provided, the palette is completed with default colours.	default palette
historic_size	positive integer	number of data points	Maximum number of data points to show in the chart. Once reached, the chart is cleared and new plots are created starting at the current time. Big sizes can impact the performance and produce lag.	10000
folder	string	_	Name of the directory where the chart files are saved. The total path is the concatenation of "path" and "folder". If it does not exist, it is created.	"gro_plots"
file_name	string	_	Base name of the files. The total image name is made by adding a number that indicates the ordering.	the name of the element
format	string in {".png", ".jpg", ".bmp"}	_	Image format	".png"
precision	integer	decimal places	Number of decimal places to round the output	-1 = no rounding (may introduce float point rounding errors)

Table 5.21: Summary of all the simulation elements

NAME	KEYWORDS	TYPE	DESCRIPTION	PREFIX	FRIENDS
Population Stat	pop_stat / pop_stats	logic	Aggregation statistics for any cellular element	stat_	In: Any cellular element. Out: PopFunction, PopQgate, OutFile, Plot
Population Function	pop_function	logic	Custom and chainable mathematical function. Analogous to cellular Function	pfun_	In: PopStat, PopFunction. Out: PopFunction, PopQgate, OutFile, Plot
Population Qgate	pop_qgate	logic	Condition with quantitative input and Boolean output. Analogous to cellular QGate	pqga_	In: PopStat, PopFunction. Out: PopBgate, Timer, OutFile, Plot
Population BGate	pop_bgate pop_gate	logic	Logic gate with Boolean inputs and output. Analogous to cellular BGate	pbga_ pga_	In: PopBGate, PopQgate. Out: PopBgate, Timer, OutFile, Plot
Timer	timer	timing	Determines when any of the following elements perform their tasks.	t_	In: PopBGate, PopQgate. Out: CellPlacer, CellPlating, SignalPlacer, Snapshot, OutFile, Plot
Checkpoint	checkpoint	timing	Pauses or stops the simulation when its condition holds	stop_ ckpt_	In: Timer
CellPlacer	cell_placer cells_placer	placer	Adds new cells to the world.	cp_	In: CellType, Timer
CellPlating	cell_plating cells_plating plating	placer	Removes and/or relocates existing cells that fulfil a condition	cpt_	In: cell-level Gate, Timer
SignalPlacer	signal_placer signals_placer	placer	Adds signals to the medium	sp_	In: Signal, Timer
Snapshot	snapshot	output	Snapshot	img_	In: Timer
OutFile	out_file cells_file	output	Custom output file describing the state of the cells over time	file_ out_	In: cell-level Gate, any cellular element, Timer
Plot	plot cells_plot	output	Interactive custom chart	plot_ pt_	In: cell-level Gate, any cellular element, Timer

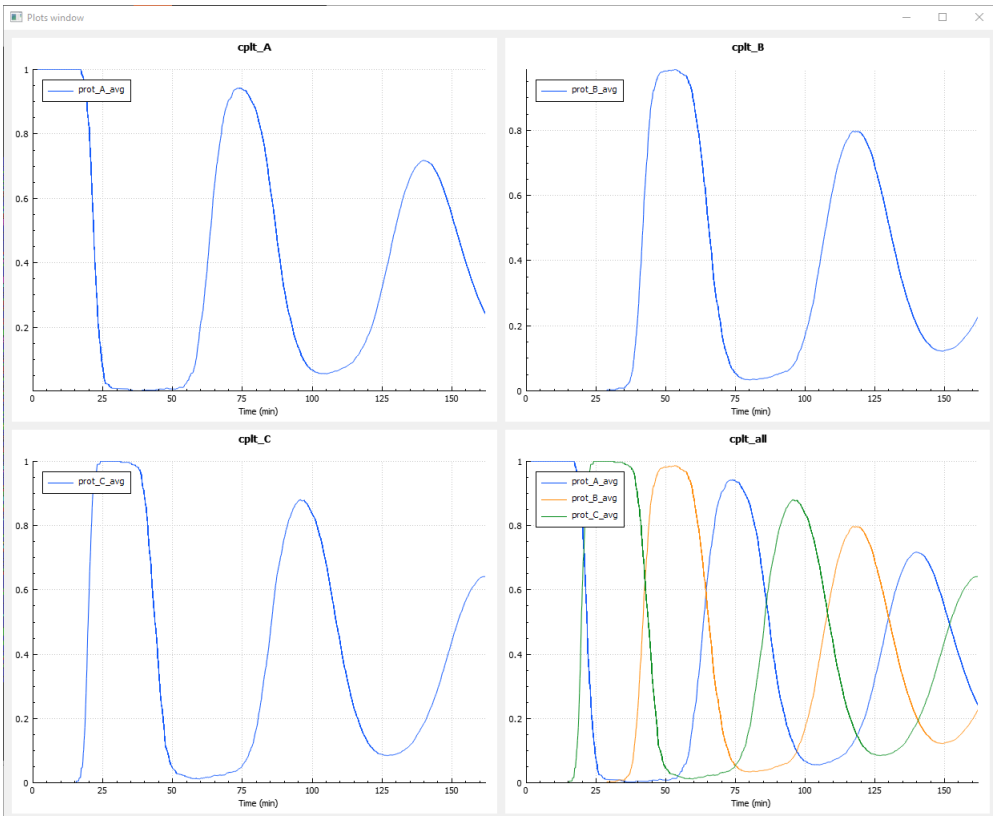

Figure 5.20: **Plots of a Repressilator circuit.** Individual plots show the oscillation of each of the three repressor proteins (A, B and C). The last chart combines three proteins.