ProSpec specification reference

Martín Eduardo Gutiérrez Pescarmona

April 17, 2017

The way of specifying simulations has been adapted to respond to a new paradigm in which structure and interactions of genetic circuits are stated (and behavior emerges). Other features and parameters that drive the simulations such as actions, variables and functions may also be part of this new specification. The new specification of GRO simulations is also compatible with the guarded command paradigm. In this document, the new syntax for simulation specifications will be described. Also, the new features will be identified to fully define the capabilities of the system.

1 A GRO specification (with its new syntax)

A GRO specification is usually written in four parts:

1. Library inclusion, parameter setting and definition of global variables and values
2. Genetic entities and their relationships
3. Actions associated to protein conditions
4. Top level programs (main function and other functions) and global control

Let's take a look into how these parts should be written.

1.1 Library inclusion, parameter setting, definition of global variables and values

Generally, library inclusion is standard and is covered by:

```
include gro
```

That instruction is the first one that should be written in any specification. All of the instructions of the specification language may be used once this instruction has been added to the specification. Other libraries may be included by using the keyword `include`, however not many libraries have been developed for our new specification yet.

Once the libraries have been included in the specification, the parameters for the simulation must be set. To set the parameter values, the keyword `set` is used as in the following example, in which the duration of the timestep is set to 0.1 minutes:

```
set("dt", 0.1);
```

In the context of parameters, the specification syntax has two new groups of parameters. The first group is the group of parameters that control the nutrients in the simulator. The other group are the parameters and settings relative to environmental signal control in the simulation.

It is important to note that not all parameters have to be set explicitly. All of the parameters have a default value. Parameters must only be set when a specific value is needed for the parameter and that parameter will be used. In all of the examples in this document, an arbitrary value is set, just for the sake of exemplification.

The parameters for the nutrient module are the following ones. The description for each parameter is in a comment at the end of each line:

```
set("nutrients", 0.0);  // 0.0: nutrient module is off,
                        // 1.0: nutrient module is on
set("nutrient_amount", 200.0);  // Amount of nutrient units per
                                // grid cell
set("nutrient_consumption_rate", 0.033); // Units/dt per bacterium
                                         // that are taken from
                                         // a grid cell
set("nutrient_grid_length", 10.0);      // Number of cells
                                        // per square side
                                        // in the grid
set("nutrient_grid_cell_size", 30.0);   // Length of each side
                                        // of a grid cell [pixels]
set("nutrient_consumption_mode", 0.0);  // 0: Homogeneous,
                                        // 1: Gradient,
                                        // 2: Random
```

In the case of environmental signal control, the following parameters can be set:

```
set("signals", 0.0);    // 0.0: signals module is off,
                        // 1.0: signals module is on
set("signals_draw", 1.0);       // 0.0: do not draw signals,
                                // 1.0: draw signals
```

Any signal definition should be placed after setting all the parameter values. In the following example, a signal identified by the symbol `s1` is defined and its parameters (diffusion coefficient, degradation coefficient and maximum concentration, respectively) are set:

```
// Grid settings (type of grid, diffusion method,
// grid length, grid cell size and neighborhood size)
// and signal parameters: diffusion, degradation.

grid("continuous", "gro_original", 10, 10, 8);
s1 := s_signal([kdiff := 0.02, kdeg := 0.005]);
```

Further along the specification, the symbol `s1` will always be used to refer to this signal. Examples of operations on these signals will be shown when we reach the action section of the specification.

Any other global variable may be defined at this stage also (such as a global time counter, strings that represent a route or filename, flags, for example), although it is not mandatory to do it in this section.

Once this section is defined, the next part of the specification consists of the genetic elements that will participate in the simulation.

1.2 Genetic entities and their relationships

The next section defines the core of the simulation. It specifies all of the genetic elements participating in the simulation, their dynamics and the relationships between them. Since the goal is to achieve a configurable environment, the vocabulary for names of genetic entities is open. This means that the programmer may choose a name for the elements, and that name shall act as an identifier throughout the specification. Therefore, the first step in this section is to identify the genetic elements, and define their dynamics and interactions.

An operon is a set of genes under the control of a single promoter. In this context, the operon is an intermediate stage of grouping, whereas the highest level of grouping comes in the form of plasmids. Both types of grouping are necessary to the specification and will be shown in this document.

Operon definition is done through the keyword `genes`. Each operon definition may hold a lot of information related to the operon, however, it is important to note that it is not mandatory to fill in all of the information. Whatever information is important should be filled in, and what is not specified takes a default value. Mandatory fields are indicated below.

An example of operon specification is now shown and will be explained:

```
genes([ name := "OperonGfp",
        proteins := {"gfp"},
        promoter := [   function := "YES",
                        transcription_factors := {"-LacI"},
                        noise := [      toOn := 0.001,
                                        toOff := 0.002,
                                        noise_time:= 100.0]],
        prot_act_times := [     times := {10.0},
                                variabilities := {2.0}],
        prot_deg_times := [     times := {5.0},
                                variabilities := {1.0}]]);
```

This is a complete example of how an operon is defined. The information held by an operon is the following:

- `name`: A name is assigned to the operon as an id. This id will later be used for setting up plasmids containing these operons. MANDATORY.
- `proteins`: This field lists the output proteins that the operon expresses. Usually, the operon expresses at least one protein. The proteins expressed by the operon are listed by their name. MANDATORY.
- `promoter`: This is a "superfield", in the form of a record, that includes several configurable features of the promoter controlling the operon. The superfield itself is MANDATORY. The subfields composing `promoter` are described here:
  - `function`: Sets the logic function according to which the promoter will operate. This function may be set to `YES`, `NOT`, `TRUE`, `FALSE`, `OR`, `XOR`, `AND` or `NAND`. Any logic function may take any number of `transcription_factor` as input to the promoter. If this subfield is skipped, it is set to the default value `YES`.
  - `transcription_factors`: It is a list of proteins that acts as the inputs for the promoter. These proteins are referred to by their name. The list may be of any size. Generally, this subfield is MANDATORY, however, when a `TRUE` or `FALSE` gate are defined for the `function` subfield, no `transcription_factors` are required. Additionally, it is possible to specify the need for the absence of a protein by adding a `-` in front of the protein in question, like in the example (`"-LacI"`).
  - `noise`: This subfield is a list of probabilities of failure of normal behavior. This is: the necessary conditions for the output proteins to be expressed at the current operon may be met, but still these proteins could fail to be expressed due to noise. Two failure cases may be set. The promoter is permanently set to off or it is permanently set to on. `toOff` and `toOn` are the respective probabilities of occurrence over a time span of `noise_time`. By default, these three subfield values are set to `0`. Note that `toOff` and `toOn` may go from `0.0` to `1.0`.

- `prot_act_times`: This field specifies a list of protein expression `times` for the operon and their `variabilities` (in minutes). MANDATORY. In the example, activation times are specified for each of the proteins stated in the instructions. The first part (`times`) specifies protein activation times and summarizes to the following: “Protein `gfp` has an activation mean time of `10.0` minutes and a standard deviation of `2.0` minutes”. It is possible to skip the variability definition altogether and have genetic elements activate always exactly at their average activation time.

- `prot_deg_times`: Very much like in the case of `prot_act_times`, this field specifies a list of protein degradation `times` for the `proteins` and the `variabilities` (in minutes) of the `times`. MANDATORY. **Note**: In case several different `prot_deg_times` are specified for the same protein, the lowest one will be the used one for all of the same `proteins`.

The user may define as many operons as necessary. Once these operons are defined, they should be organized in `plasmids`. A plasmid may include any number of operons. The association between `plasmids` and operons (`genes`) is done through an instruction called `plasmids_genes`. This instruction defines a list of operons associated to each `plasmid`. The plasmids will be identified by their names, and so will the operons.

An example is shown for clarity:

```
plasmids_genes ([       p1 := {"operon0","operon3"},
                        p2 := {"operon2"},
                        p6 := {"operon1","operon2","operon3"},
                        p3 := {"operon5"},
                        p5 := {"operon2","operon3","operon1"}]);
```

In this example, five plasmids are assigned different operons. Note that the same operon may be assigned to several plasmids.

As the last feature of genetic elements, environmental signals will be mentioned. Signals are transferred to a genetic element in the form of a `transcription_factor`. Hence, a signal may be a trigger for the function of an operon. How the transfer is carried out will be discussed further along this document in the next section, related to `actions`.

1.3 Actions associated to protein conditions

Once the core of the elements and their interactions are in place, the actions that are executed by bacteria must be specified. The way in which this is done is by linking an `action` to a specific protein condition. The condition is expressed as a set composed of boolean states corresponding to protein expression. This represents the fact that whenever the condition is met, it will fire a certain `action` with its parameters. The keyword `action` is used to define the instruction linked to the condition. An example will now be shown and explained, and then all currently implemented `actions` will be listed:

```
action({"-ara","tetR"},"lose_plasmid",{"p5"});
```

This clause states that whenever the protein `ara` is absent (`-ara`) and protein `tetR` is present in a bacterium, plasmid `p5` is removed from the list of `plasmids` of the bacterium. The list containing `p5` is the list of parameters associated to the `action lose_plasmid`. This parameter list is always passed in the form of a list of strings. Whichever `protein` that is NOT explicitly mentioned in the condition, is assumed to be of status "don't care". This means, it does not affect the evaluation of the logical condition. `Protein` ids should come from the glossary and the specific `actions` to be executed will be listed now along with the parameters they take in:

1. `paint (green, red, yellow, cyan)`: This `action` paints the bacterium according to the color combination given by the four color channels `green`, `red`, `yellow` and `cyan`. Valid values for these channels are integer numbers ranging from `0` to `50` (this is defined in `gro.gro`, these thresholds can be changed). Note that this instruction is absolute and imposes the color combination without regard for the previous color state in the bacterium.

2. `d_paint (green, red, yellow, cyan)`: Unlike the previous `action`, this one adds (or subtracts) a certain amount of "channel concentration". This is, it modifies the previous color distribution in the bacterium by adding or subtracting a difference in each of the channels. As in the previous case, the channels may take values ranging from `0` to `50` (integer).

3. `die`: This `action` kills the bacterium and makes it disappear from the simulation space. At this time, it is the only `action` that does not take a parameter list in.

4. `conjugate (plasmid, rate)`: `action` that copies a `plasmid` from a source bacterium (the one executing the `action`) to a random neighboring bacterium. This copy is done at an average `rate`, given by the average number of conjugations that occur during the lifetime of a bacterium.

5. `conjugate_directed (plasmid, rate)`: Similar to the previous `action`, except that it considers a mechanism (called eex) in the destination bacterium that may not allow `plasmid` entry. Whenever entry is closed to the `plasmid` in the destination bacterium, this destination bacterium is not considered as a viable neighbor and therefore, the destination selection is directed toward all other viable neighbors. The `rate` parameter in this `action` has the same meaning as in the `conjugate action`.

6. `lose_plasmid (plasmid)`: `action` that removes `plasmid` from the list of `plasmids` residing in the bacterium.

7. `set_eex (plasmid)`: Sets a constraint against `plasmid` entering the current bacterium (by conjugation).

8. `remove_eex (plasmid)`: Removes the constraint placed upon `plasmid` that forbids its entry to the current bacterium (by conjugation).

9. `set_growth_rate (rate)`: Sets the rate at which a bacterium will grow. This rate is given in μm/min.

10. `emit_cf (signal_id, concentration)`: `action` that commands the current bacterium to emit `concentration` amount of the cross-feeding signal identified by `signal_id`.

11. `get_cf (signal_id, benefit)`: This `action` reads the concentration of cross-feeding signal `signal_id` in the medium. Depending on the value of the `benefit` parameter (`1` simbolizes a positive interaction -growth increases- with the signal, `-1` simbolizes a negative interaction -growth decreases-), the growth rate of the bacterium will be changed.

12. `s_emit_signal (signal_id, concentration, emission_type)`: This `action` is associated to the emission of a signal (identified by `signal_id`) `concentration` into the environment by a bacterium. It is emitted according to an `emission_type`, being `"exact"`, `"area"` or `"average"`. The first type emits the `exact concentration` only at the cell under the center of the bacterium, the second one emits the given `concentration` to all grid cells in the `area` under the bacterium and the third mode emits to this same area, but the amount is an `average concentration` over the number of cells. These modes always have the same range meaning for all signal actions, but of course refer to the specific `action`.

13. `s_absorb_signal(signal_id, concentration, absorption_type)`: `action` that prompts the absorption of a given signal (identified by `signal_id`) `concentration` from the environment. Whenever absorption is executed, the given `concentration` amount is absorbed, but if there is not enough signal at the specified location (or locations), whatever present `concentration` is absorbed. In addition, a fourth `type` is included in this case: `random`. This `absorption_type` takes signal `concentration` from a randomly chosen cell in the area under the bacterium.

14. `s_get_signal(signal_id)`: senses the concentration of a given signal (identified by `signal_id`) from the environment.

15. `s_set_signal (signal_id, concentration, x, y)`: Emission of a determinate `concentration` of a given signal(identified by `signal_id`) into the environment (at location (x,y)).

16. `s_emit_cf(signal_id, concentration, emission_type)`: Refers to the emission of a signal (identified by `signal_id`) into the environment by

a given `concentration`. This directive is related to cross-feeding experiments, meaning the emitted concentration is scaled by a detrimental/beneficial coefficient that is associated to the bacterium's metabolism. The `emission_types` can be: `"exact"`, `"area"` or `"average"`.

17. `s_absorb_cf(signal_id, concentration, benefit, absorption_type)`: Absorbs a certain `concentration` of signal (identified by `signal_id`) from the environment. This kind of absorption affects cross-feeding experiments. The absorbed concentration triggers a response (represented by `benefit`: `-1` is a detrimental effect and `1` is a beneficial effect) in the metabolism of the bacterium that absorbs. The absorbed amount of signal modulates the effect. Signal can be absorbed in four `types`: `"exact"`, `"area"`, `"average"` or `"random"`.

18. `s_get_cf(signal_id, concentration, benefit)`: Reads a certain concentration of signal (identified by `signal_id`) from the environment. This kind of absorption affects cross-feeding experiments. The read concentration triggers a response (represented by `benefit`: `-1` is a detrimental effect and `1` is a beneficial effect) in the metabolism of the bacterium that is sensing. The read amount of signal modulates the effect.

19. `s_absorb_QS(signal_id, comparison, threshold, protein)`: Absorbs a certain amount of signal (identified by `signal_id`) from the environment. If the absorbed amount satisfies a condition given by a `comparison` sign (`"<"` or `">"`) and a `threshold`, the genetic element `protein` is activated. Otherwise, `protein` is deactivated. **VERY IMPORTANT NOTE**: any `protein` defined in an action of this kind CANNOT be used in action conditions.

20. `s_get_QS(signal_id, comparison, threshold, protein)`: Is the same `action` as the above, except that it does not absorb signal from the environment and only senses. The **VERY IMPORTANT NOTE** holds for `proteins` defined by this action as well.

Several other `actions` may be implemented in the future, such as CRISPR related functions or phage control. It is very important to note that despite defining a new specification paradigm for controlling the bacteria in the simulations, this paradigm is still compatible with the previous one defined through guarded commands (meaning all (but one, due to constraints of the physical engine - chemostat mode) of the instructions that come from the original GRO still work within the scope of programs.

1.4 Top-level programs and global control

This section will simply be a reminder, since the definition of top-level programs and global control come from the original GRO specification. Programs are code spaces where conditions are evaluated and instructions are executed in response to these conditions (this is the guarded command paradigm). Most of the time, programs are assigned to a bacterium as their behavior.

Another important program is the `main` program. The `main` program controls the environment and the top level settings of the simulator. Examples of these settings may be the data gathering, iteration control of simulations, bacteria creation and placement. The only difference in the pre-existing instructions lie in the bacterial creation. The instructions `ecoli` and `c_ecolis` have been modified to include more information to initialize bacteria in the simulation space. An example of each instruction will be shown now and will be briefly explained:

```
ecoli([y := 20, x := 50],{"p1"},program p());
ecoli([y := 20, x := 50],program p());
```

This instruction creates a single ecoli agent. It receives two or three parameters. The first parameter is a record that specifies the physical definitions of the bacterium. These variables are: `x`, `y`, `theta` and `volume` (all float values). Next (optional) is the `plasmid_list` parameter that specifies what `plasmids` are initially present in the bacterium. The `plasmids`, as always, are identified through their name (id). Finally, a `program` to run is associated to the bacterium. Note that when the new paradigm is used, this `program` is usually empty (`skip();`).

```
c_ecolis(200, 0, 0, 200, {"p1"}, program p());
```

`c_ecolis` is an instruction that creates a group of bacteria in a random circular pattern in batch. Only the four first parameters will be described, as the two last ones are the same final parameters of the `ecoli` instruction with 3 parameters. The first parameter is `n` (integer). It determines the number of `ecolis` to create. The two following parameters are `x` and `y` (both floats), and establish where the circular pattern will be centered in the simulation space. The fourth parameter is the maximum `radius` in pixels (float) of the circular pattern up to where bacteria may be placed. The rest of the parameters have already been explained for the previous instruction and have the same meaning in this instruction.

2 Basic specification examples

A couple of real examples will be appended to demonstrate how all of the syntax discussed in this document may be used.

The first example, `newLtest.gro`, is a specification in which the features of the language are tested. It was designed as the skeleton for building the new specification language. Specifically, there is a large list of parameter setting, data collection (stored in files), circuit definition (very extensive, using most of the capabilities of the specification) and a showcase of several options of actions.

2.1 newLtest.gro

```
include gro

// default nutrient parameters
set ("nutrients", 0.0); //0.0: off, 1.0: on
set ("nutrient_amount", 200.0); //Arbitrary units
set ("nutrient_consumption_rate", 0.033); // Units/dt per bacterium
set ("nutrient_grid_length", 10.0); //In grid cells
set ("nutrient_grid_cell_size", 30.0); //In pixels
set ("nutrient_consumption_mode", 0.0); /*0: Homogeneous, 1:
Gradient, 2: Random (between 50% of max concentration and max
concentration)*/

// Standard parameters
set("dt", 0.1);
set("population_max", 2000000);

// Control variables
t := 0;
i := 0;
n_run := 0;
n_runs := 10;

// Routes and filenames
route1 := "/Users/a/b/c/";
route2 := "/Users/a/b/d/";
filename := "example"; //Filename to where data goes

// File opening
//fp := fopen (route1 <> filename <> tostring((i+1)) <> ".csv", "w");

// Signal grid and signal definition
grid("continuous", "gro_original", 10, 10, 8);
cf1 := s_signal([kdiff := 0.02, kdeg := 0.005]);
```

```
/* Example of cell program using selected cell for single dump (to
file fp)*/
/*program p() :=
{
        selected:
        {
                dump_single(fp);
        }
};*/

//Empty program example
program p() :=
{
        skip();
};

// Movie program example
/*program movie() :=
{
        t1 := 0;
        set ("ecoli_growth_rate", 0.0);

        true:
        {
                t1 := t1+dt
        };

        t1 > 1 :
        {
                snapshot ( route1 <> tostring(n) <> ".tif" ),
                n := n + 1,
                t1 := 0
        }
};*/

// Operon definition
genes([ name := "operon0",
                proteins := {"gfp","lacI","ara"},
                promoter := [   function := "YES",
                                transcription_factors := {"-tetR"},
                                noise:= [       toOn := 0.02,
                                                toOff := 0.2,
                                                noise_time := 200.0]],
                prot_act_times := [     times := {10.0, 3.0, 4.5},
                                        variabilities := {2.0, 1.0, 1.1}],
                prot_deg_times := [     times := {20.0, 50.0, 10.0},
                                        variabilities := {3.0, 10.0, 2.0}]]);
```

```
// Plasmid inclusion matrix (of operons)
plasmids_genes ([       p1 := {"operon0","operon3"},
                        p2 := {"operon2"},
                        p6 := {"operon1","operon2","operon3"},
                        p3 := {"operon5"},
                        p5 := {"operon2","operon3","operon1"}]);

// Example of how to use several actions.
action({"gfp"},"paint",{"500","0","0","0"});
action({"-gfp"},"d_paint",{"-1","0","0","0"});
action({"ara"},"conjugate",{"p5","1.0"});
action({"lacI","cI"},"die",{"0"});
action({"-ara","tetR"},"lose_plasmid",{"p5"});
action({"gfp"},"set_growth_rate",{"0.011"});
action({"-tetR","-ara"},"s_emit_cf",{tostring(cf1),"10.0","exact"});
action({"tetR"},"s_get_cf",{tostring(cf1),"5","-1"});

// Example of main program
/*program main() :=
{
        t = 0:
        {
                fprint(fp, "Time, LacI, not(TetR)\n");
        }

        true:
        {
                //dump_multiple(fp,{"LacI"},{"-TetR"});
                t := t + dt;
        }


        t > 300 & n_run < n_runs:
        {
                reset();
                reset_actions();
                t := 0;
                n_run := n_run + 1;
                set ( "dt", 0.1 );
                set ( "population_max", 2000000 );
                /*fp := fopen ( route1 <> filename1 <> tostring(n_run) <>
                ".csv", "w" );*/

                /* E Coli creation. Parameters: [Location, angle, volume],
                {plasmids included}, program to run*/
                ecoli([y := 20, x := 50],{"p1"},program p());
                /* E Coli mass creation. Parameters: number of E Coli to
                create, x (center of cluster), y (center of cluster),
                maximum radius of cluster, {plasmids included},
                program to run*/
                c_ecolis(200, -50, -70, 30, {"p1"}, program p());
        }

        t > 150 & n_run >= n_runs:
        {
                stop();
        }*/

        /* E Coli creation. Parameters: [Location, angle, volume],
        {plasmids included}, program to run*/
        //ecoli([y := 20, x := 50],{"p1"},program p());

        /* E Coli mass creation. Parameters: number of E Coli to
        create, x (center of cluster), y (center of cluster),
        maximum radius of cluster, {plasmids included},
        program to run*/
        c_ecolis(200, 0, 0, 200, {"p1"}, program p());
//};
```

The second example is meant to test the implementation of the promoters for a circuits that have different actions associated to them, such as glowing or dying. Data is also collected in this example, and action showcase is present, although not all actions are used.

2.2 testConstitutive.gro

```
include gro

// Standard parameters
set("dt", 0.1);
set("population_max", 2000000);

// default nutrient parameters
set("nutrients", 0.0);  // 0.0: nutrient module is off,
                        // 1.0: nutrient module is on
set("nutrient_amount", 200.0);  // Amount of nutrient units per
                                // grid cell
set("nutrient_consumption_rate", 0.033); // Units/dt per bacterium
                                          // that are taken from
                                          // a grid cell
set("nutrient_grid_length", 10.0);      // Number of cells
                                        // per square side
                                        // in the grid
set("nutrient_grid_cell_size", 30.0);   // Length of each side
                                        // of a grid cell [pixels]
set("nutrient_consumption_mode", 0.0);  // 0: Homogeneous,
                                        // 1: Gradient,
                                        // 2: Random

set ( "ecoli_growth_rate", 0.0346574);
set ( "signal_grid_width", 800 );
set ( "signal_grid_height", 800 );
set ( "signal_element_size", 5 );
set ( "ecoli_growth_rate_max", 0.0346574);

// Cross-feeding signal parameters: diffusion and degradation.
grid("continuous", "gro_original", 10, 10, 8);
cf1 := s_signal([kdiff := 1.0, kdeg := 0.001]);
cf2 := s_signal([kdiff := 1.0, kdeg := 0.001]);
```

```
// Operon definition
genes([ name := "Operon1",
                proteins := {"gfp", "rel1"},
                promoter := [   function := "OR",
                                transcription_factors := {"lacI", "-rel2"}],
                prot_act_times := [times := {0.5, 20.0}],
                prot_deg_times := [times := {10.0, 10.0}]]);

genes([ name := "Operon2",
                proteins := {"rel2", "lacI"},
                promoter := [   function := "TRUE"],
                prot_act_times := [times := {0.2, 50.0}],
                prot_deg_times := [times := {10.0, 10.0}]]);

// Plasmid inclusion matrix (of operons)
plasmids_genes ([       p1 := {"Operon1"},
                        p2 := {"Operon2"} ]);

//ACTIONS

//Express "color" with a fixed saturation. Code: {G,R,Y,C}
action({"gfp","-rel2"},"paint",{"500","0","0","0"});

//Express "color" gradually. Code: {G,R,Y,C}
//action({"-gfp"},"d_paint",{"-1","0","0","0"});

action({"rel2","-gfp"},"paint",{"0","500","0","0"});
action({"rel2","gfp"},"paint",{"0","0","500","0"});
//action({"gen2"},"paint",{"0","500","0","0"});
//action({"-gfp"},"d_paint",{"-20","0","0","0"});

//Plasmid id, conjugation rate
//action({"rel1"},"conjugate",{"p1","1.0"});
//action({"rel2"},"conjugate",{"p2","1.0"});

action({"lacI"},"die");

//Plasmid id
//action({"-ara","tetR"},"lose_plasmid",{"p1"});

//ln(2)/gt(min)[um/min]
//action({"gfp"},"set_growth_rate",{"0.011"});

//Cross-feeding emission tag, concentration, emission mode
//action({"gen1"},"s_emit_cf",{tostring(cf1),"20","exact"});
//action({"gen2"},"s_emit_cf",{tostring(cf2),"20","exact"});

//Cross-feeding reception tag, concentration, benefit: 1. good, -1. toxin
//action({"gen1"},"get_cf",{tostring(cf2),"20","-1"});
//action({"gen2"},"get_cf",{tostring(cf1),"20","-1"});
```

```
// Control variables
t := 0;
i := 0;
n_run := 0;
n_runs := 10;

// Routes and filenames
route1 := "/Users/a/b/c/";
//route2 := "/Users/a/b/d/";
filename := "constitutive"; //Filename to where data goes
//fp := fopen (route1 <> filename <> tostring((i+1)) <> ".csv", "w");

// File opening
fp := fopen ( route1 <> filename <> ".csv", "w");

///PROGRAMS
//Empty program example
program p() :=
{
        skip();
};

// Movie program example
/*program movie() :=
{
        t1 := 0;
        //The recording bacteria does not grow
        set ("ecoli_growth_rate", 0.0);

        true:
        {
                t1 := t1+dt
        };

        t1 > 1 :
        {
                snapshot ( route1 <> tostring(n) <> ".tif" ),
                n := n + 1,
                t1 := 0
        }
};*/
```

```
/* Program using selected cell for single dump (to file fp). SINGLE
MODE*/
program p() :=
{
        selected:
        {
                //dump_single(fp);
                die();
        }
};

// Main program
program main() :=
{
        /*t = 0:
        {
                fprint(fp, "Time, LacI, not(TetR)\n"); // Prints headings
        }*/

        /*true:
        {
                dump_multiple(fp,{"LacI"},{"-TetR"}); //MULTIPLE MODE
                t := t + dt;
        }*/

        c_ecolis(200, 0, 0, 80, {"p1"}, program p());
        c_ecolis(100, 0, 0, 80, {"p2"}, program p());
};
```

More examples can be found at https://github.com/liaupm/GRO-LIA in the "PaperExamples" folder.