eBCSgen: Tutorial

Matej Troják, David Šafránek, Lukrécia Mertová, Lucia Helmeciová, and Luboš Brim

Contents

1 Introduction **3**

2 Getting Started **4**
- 2.1 Galaxy UI basics . . . 4
- 2.2 BioChemical Space Language models . . . 7
  - 2.2.1 Language syntax . . . 7
  - 2.2.2 Editor . . . 8
  - 2.2.3 Example model . . . 9

3 Analytical Tools **11**
- 3.1 Transition System generator . . . 11
  - 3.1.1 Input specification . . . 11
  - 3.1.2 Output specification . . . 11
  - 3.1.3 Results visualisation . . . 12
- 3.2 Simulation . . . 13
  - 3.2.1 Input specification . . . 14
  - 3.2.2 Output specification . . . 14
  - 3.2.3 Results visualisation . . . 14
- 3.3 CTL Model Checking . . . 14
  - 3.3.1 Input specification . . . 14
  - 3.3.2 Output specification . . . 15
- 3.4 PCTL Model Checking . . . 15
  - 3.4.1 Input specification . . . 15
  - 3.4.2 Output specification . . . 15
- 3.5 PCTL Parameter Synthesis . . . 15
  - 3.5.1 Input specification . . . 16
  - 3.5.2 Output specification . . . 16
  - 3.5.3 Results visualisation . . . 16
- 3.6 Probability Sampling . . . 16
  - 3.6.1 Input specification . . . 17
  - 3.6.2 Output specification . . . 17
- 3.7 Static Analysis . . . 17
  - 3.7.1 Input specification . . . 18
  - 3.7.2 Output specification . . . 18
- 3.8 SBML export . . . 19
  - 3.8.1 Input specification . . . 19
  - 3.8.2 Output specification . . . 19

4 Regulations **20**
4.1 Example of regulated model . . . 21
4.2 Types of regulations . . . 22
4.2.1 Regular rewriting . . . 23
4.2.2 Ordered rewriting . . . 24
4.2.3 Programmed rewriting . . . 25
4.2.4 Conditional rewriting . . . 26
4.2.5 Concurrent-free rewriting . . . 27
4.3 Regulation exercises . . . 29
4.3.1 Task 1 . . . 30
4.3.2 Task 2 . . . 31
4.3.3 Task 3 . . . 32
4.3.4 Correct answers . . . 33

5 BCSL tutorial **34**
5.1 Operators . . . 34
5.2 Example model . . . 34
5.3 Complex aliases . . . 36
5.4 Rates . . . 36
5.5 Definitions . . . 38
5.6 Exercise . . . 39
5.6.1 Correct answers . . . 40

Chapter 1

Introduction

This tutorial is dedicated to assist users in learning the basic usage of the tool eBCSgen. eBCSgen is used to develop and maintain models written in BioChemical Space language (BCSL) and to analyse them. The eBCSgen is implemented as a part of the Galaxy tool and is available online on the address `https://biodivine-vm.fi.muni.cz/galaxy/`.

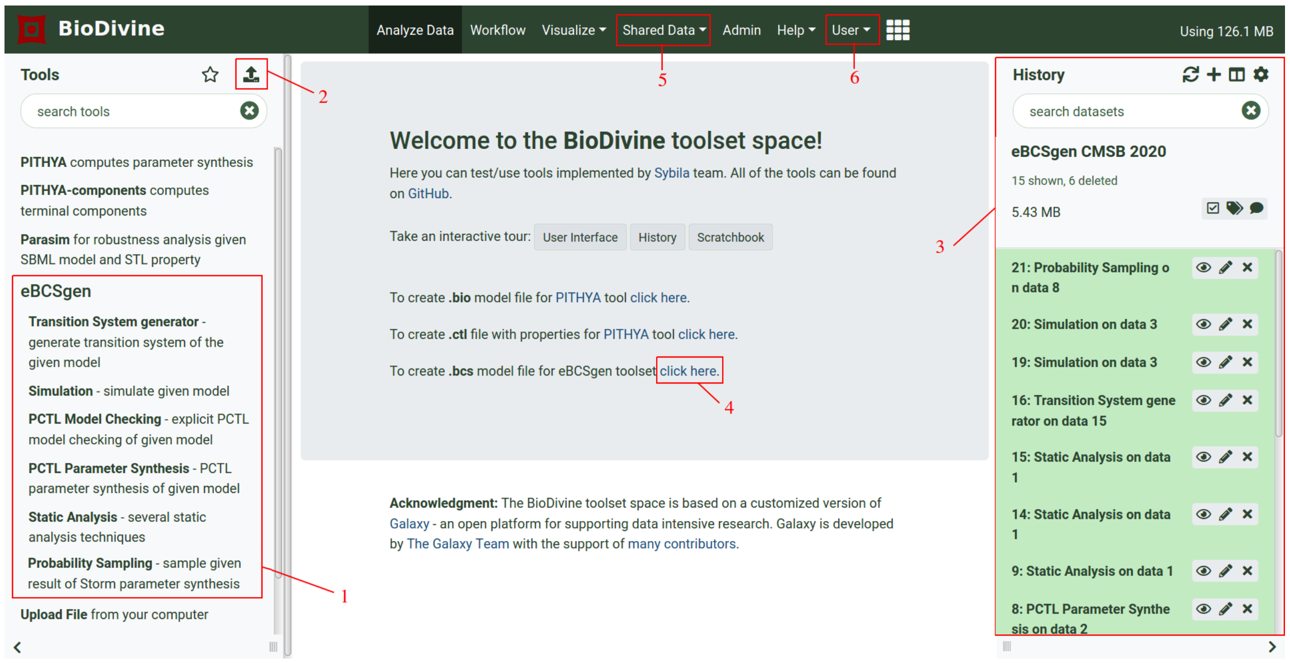
In order to be able to use the tool, it is necessary to learn basics of the Galaxy user interface and syntax of BCSL. With this knowledge, it is then possible to proceed to Analytical tools provided by eBCSgen accompanied by specific visualisations.

Chapter 2

Getting Started

To learn how to use the Galaxy UI for the purposes of this tutorial, it should be sufficient to read and follow the steps described in the following section. Besides this text, Galaxy interactive tours are available to walk the user through Galaxy features and individual tools. However, if you need more information, we recommend the official Galaxy project homepage, where a constantly growing set of tutorials is available.

2.1 Galaxy UI basics



PITHYA computes parameter synthesis

PITHYA-components computes terminal components

Parasim for robustness analysis given SBML model and STL property

eBCSgen

Transition System generator - generate transition system of the given model

Simulation - simulate given model

PCTL Model Checking - explicit PCTL model checking of given model

PCTL Parameter Synthesis - PCTL parameter synthesis of given model

Static Analysis - several static analysis techniques

Probability Sampling - sample given result of Storm parameter synthesis

Upload File from your computer

Welcome to the BioDivine toolset space!

Here you can test/use tools implemented by Sybila team. All of the tools can be found on GitHub.

Take an interactive tour: User Interface History Scratchbook

To create .bio model file for PITHYA tool click here.

To create .ctl file with properties for PITHYA tool click here.

To create .bcs model file for eBCSgen toolset click here.

Acknowledgment: The BioDivine toolset space is based on a customized version of Galaxy - an open platform for supporting data intensive research. Galaxy is developed by The Galaxy Team with the support of many contributors.

eBCSgen CMSB 2020

15 shown, 6 deleted

5.43 MB

21: Probability Sampling on data 8

20: Simulation on data 3

19: Simulation on data 3

16: Transition System generator on data 15

15: Static Analysis on data 1

14: Static Analysis on data 1

9: Static Analysis on data 1

8: PCTL Parameter Synthesis on data 2

Figure 2.1: *A screenshot of the main page of Galaxy tool. The key components of UI are highlighted by red frames: (1) available eBCSgen Analytical tools; (2) uploading utility; (3) history of your computations; (4) create new BCSL model; (5) available shared data; (6) userspace.*

The Galaxy tool is composed of several components. Figure 2.1 shows the homepage of Galaxy tool with highlighted key components:

1. *eBCSgen Analytical tools* – the main functionality of eBCSgen can be found here. Individual tools and how to use them is described in section Analytical tools.

2. *uploading utility* – upload local files and download data from the web by entering URLs.

   To learn how to use uploading utility, we recommend Galaxy UI tour.

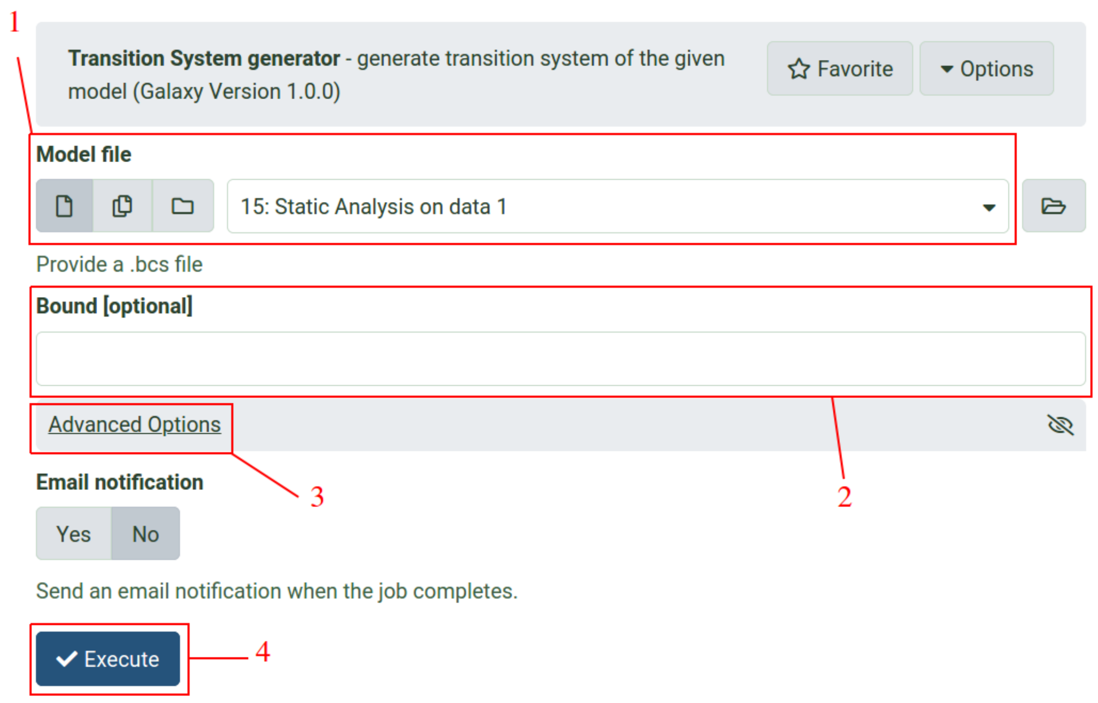
3. *history of your computations* – data space where all files and analysis results are available. This helps to keep track of analysis results origin, allows to easily chain tools, and share reproducible data.

   To learn how to work with history, we recommend History guide tour.

4. *create new BCSL model* – open an empty file in BCSL editor which allows to create a new model from scratch.

5. *shared data* – published data libraries and histories available as reference data. There are also available example files for this tutorial.

6. *userspace* – it is recommended to register and stay logged in for all times. This enables some features such analysis of results and saves history for other sessions.


Transition System generator - generate transition system of the given model (Galaxy Version 1.0.0)

Model file: 15: Static Analysis on data 1

Provide a .bcs file

Bound [optional]

Advanced Options

Email notification: Yes [✓ No]

Send an email notification when the job completes.

Figure 2.2: *A screenshot of a tool in Galaxy UI. The highlighted components in red frames are: (1) input file to the tool; (2) argument of the tool; (3) hidden advanced options; (4) execution button.*

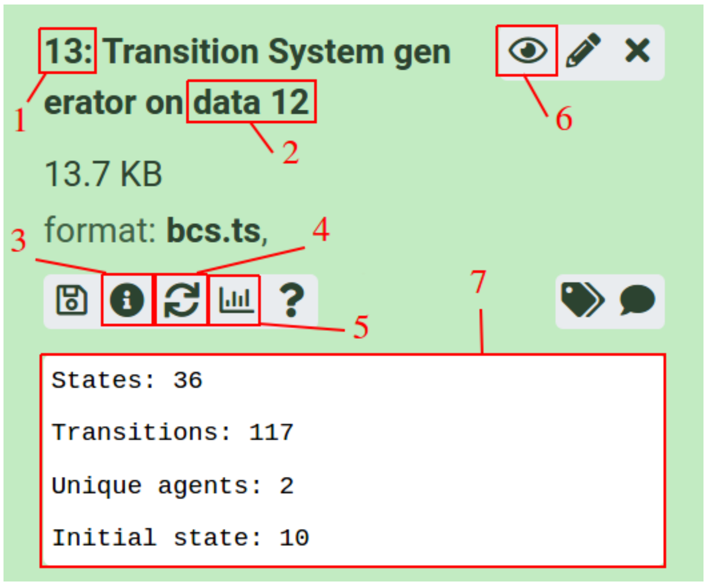
Every tool in the Galaxy UI has its interface, which allows the user to comfortably fill all required arguments for the tool in order to execute it successfully. In Figure 2.2, there are highlighted some of the key components:

1. *tool input files* – Galaxy is based on file transformation principle, i.e. each tool takes an input a file and produces an output file (can be multiple on both ends).

2. *additional arguments* – along with files, the tool might require some additional arguments, e.g. textual field, numerical value, a variant choice.

3. *hidden options* – tool can have some optional or repetitive arguments which can be collapsed/hidden.

4. *execute button* – once all arguments have been filled, the tool can be executed and its results will appear in the history.



```
States: 36
Transitions: 117
Unique agents: 2
Initial state: 10
```

Figure 2.3: *A screenshot of a history item in Galaxy UI. The highlighted components in red frames are: (1) ID of the data; (2) ID of the source data; (3) complete information about the data; (4) run the job again; (5) visualise the data; (6) view file content; (7) peek of the file content.*

The history is a key component of Galaxy UI. It is a time-line of tool results and allows the user to reproduce the tool execution completely. The background colour of the file indicates its state: green – execution is finished successfully, orange – execution in progress, and red – execution failed (check the bug report). In figure 2.3, there are highlighted some of the key components of an item in the history:

1. *ID of the data* – unique identification in the history.

2. *ID of the source data* – the ID of file used as an input file for the tool.

3. *information* – complete information about the file (e.g. size, format) and job it was created by (e.g. input arguments, execution time).

4. *re-run the job* – possibility to run the job again with the same or modified arguments.

5. *visualise* – displays visualisation options for the data. All displayed visualisations can be used to produce a graphical output for the data. However, we recommend particular visualisation tailored for the data format for each individual tool in section Analytical tools.

6. *view file content* – show entire file content.

7. *peek of the file content* – just a peek of the file content or file statistics.

2.2 BioChemical Space Language models

2.2.1 Language syntax

The BCSL model is composed of several sections: *rules* contains the set of rules defining the behaviour of the model, *inits* defines the initial state of the model, *definitions* allows to assign parameter values, and (optional) *complexes* defines aliases for complexes. In Figure 2.4, the context-free grammar of the BCSL is provided. It is defined in extended Backus–Naur form (EBNF), recognised by context-free grammar parser Lark, a parsing library for Python. An example of the model is available in section Example model. To understand the semantics of the language in more details, we recommend studying paper containing formal definition or language tutorial for a more elemental description.

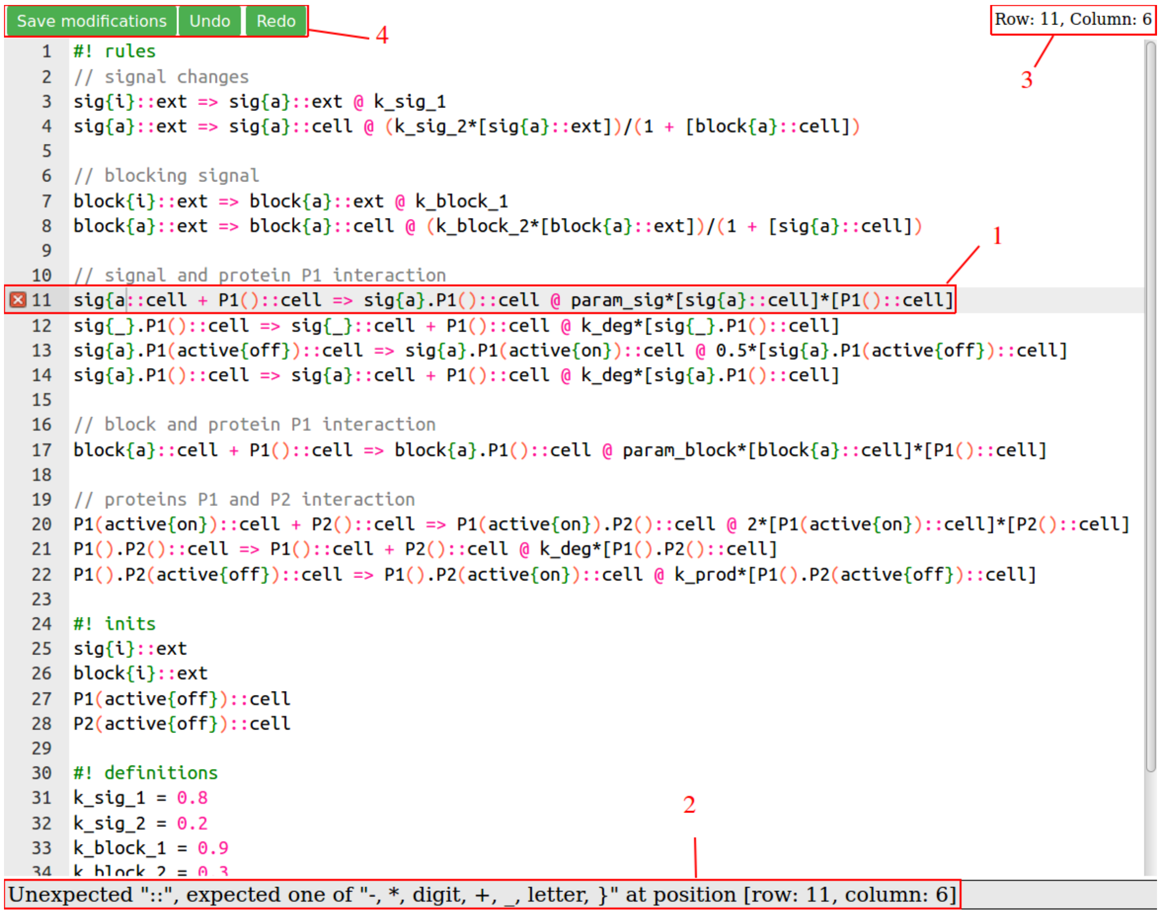
```
model: rules inits? definitions? complexes?

rules: "#! rules" (rule|COMMENT)+
inits: "#! inits" (init|COMMENT)+
definitions: "#! definitions" (definition|COMMENT)+
complexes: "#! complexes" (complex_def|COMMENT)+

init: const? rate_complex (COMMENT)?
definition: param "=" const (COMMENT)?
rule: LABEL? side "=>" side ("@" rate)? (";" variable)? (COMMENT)?
complex_def: complex_name "=" sequence (COMMENT)?

side: (const? complex "+")* (const? complex)?
complex: (abstract_sequence|sequence) "::" compartment

rate_complex: sequence "::" compartment
sequence: (agent ".")* agent
agent: atomic | structure
structure: s_name "(" composition ")"
composition: (atomic ",")* atomic?
atomic: a_name "{" state "}"

!state: (digit|letter|"+"|"-"|"*"|"_")+

!rate: fun "/" fun | fun
!fun: const | param | rate_agent | fun "+" fun |
      | fun "-" fun | fun "*" fun | fun "**" const | "(" fun ")"

!rate_agent: "[" rate_complex "]"

abstract_sequence: atomic_complex | atomic_structure_complex | structure_complex
atomic_complex: atomic ":" (complex_name|"?")
atomic_structure_complex: atomic ":" structure ":" (complex_name|"?")
structure_complex: structure ":" (complex_name|"?")

variable: "?" "=" "{" complex_name ("," complex_name)+ "}"

COMMENT: "//" /[^\n]/*
```

Figure 2.4: *The context-free grammar of BCSL in EBNF notation. The grammar is simplified – obvious terminals such as names, constants and numbers are omitted.*

2.2.2 Editor

BCSL editor is technically an interactive visualisation of a BCSL model. Therefore it can be accessed the same way as all the other visualisations (see Galaxy UI basics) by visualising a model. The key features of the editor are highlighted in Figure 2.5:

1. *a rule containing syntax error* – a single rule containing syntax error, which is indicated by the red cross on the left to the line number.
2. *error message* – details of the syntax error detected by syntax checker. The message identifies an unexpected symbol, its exact position in the model and list of expected symbols.
3. *position of cursor* – information about the current position of the cursor in the model file.
4. *action buttons* – there are three buttons available: `Save modifications` to save the current content as a new BCSL model, `Undo` and `Redo` to reverse your last modification (resp. to reverse your last undo).



Row: 11, Column: 6

```
1  #! rules
2  // signal changes
3  sig{i}::ext => sig{a}::ext @ k_sig_1
4  sig{a}::ext => sig{a}::cell @ (k_sig_2*[sig{a}::ext])/(1 + [block{a}::cell])
5
6  // blocking signal
7  block{i}::ext => block{a}::ext @ k_block_1
8  block{a}::ext => block{a}::cell @ (k_block_2*[block{a}::ext])/(1 + [sig{a}::cell])
9
10 // signal and protein P1 interaction
11 sig{a::cell + P1()::cell => sig{a}.P1()::cell @ param_sig*[sig{a}::cell]*[P1()::cell]
12 sig{_}.P1()::cell => sig{_}::cell + P1()::cell @ k_deg*[sig{_}.P1()::cell]
13 sig{a}.P1(active{off})::cell => sig{a}.P1(active{on})::cell @ 0.5*[sig{a}.P1(active{off})::cell]
14 sig{a}.P1()::cell => sig{a}::cell + P1()::cell @ k_deg*[sig{a}.P1()::cell]
15
16 // block and protein P1 interaction
17 block{a}::cell + P1()::cell => block{a}.P1()::cell @ param_block*[block{a}::cell]*[P1()::cell]
18
19 // proteins P1 and P2 interaction
20 P1(active{on})::cell + P2()::cell => P1(active{on}).P2()::cell @ 2*[P1(active{on})::cell]*[P2()::cell]
21 P1().P2()::cell => P1()::cell + P2()::cell @ k_deg*[P1().P2()::cell]
22 P1().P2(active{off})::cell => P1().P2(active{on})::cell @ k_prod*[P1().P2(active{off})::cell]
23
24 #! inits
25 sig{i}::ext
26 block{i}::ext
27 P1(active{off})::cell
28 P2(active{off})::cell
29
30 #! definitions
31 k_sig_1 = 0.8
32 k_sig_2 = 0.2
33 k_block_1 = 0.9
34 k_block_2 = 0.3
```

Unexpected "::", expected one of "-, *, digit, +, _, letter, }" at position [row: 11, column: 6]

Figure 2.5: *A screenshot of a BCSL editor in Galaxy UI. The highlighted components in a red frame are: (1) a rule containing syntax error; (2) console containing error message; (3) position of cursor; (4) action buttons.*

2.2.3 Example model

As example model in this tutorial, we use an artificial signalling pathway. This model is available in Shared Data Library or as Shared History including all analysis results demonstrated in Analytical Tools section. The data from both sources can be imported to user workspace. There are actually two separate models – `model.bcs` and `model_param.bcs`, both representing the same system with the exception of parameter values – the latter model has two unknown parameter values. There is actually a third model called `repressilator.bcs`, but this model is only used for demonstration by Simulation tool for its interesting qualitative properties. For details about the syntax of the model, see Language syntax section.

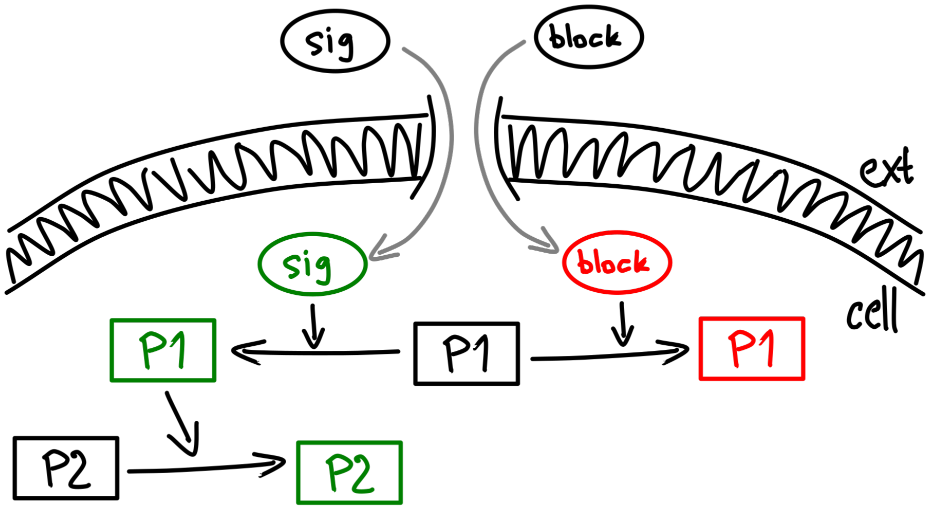


Figure 2.6: *A simplified scheme depicting the signalling pathway represented by `model.bcs`. The signal transduction starts in extracellular space `ext` and continues inside the `cell`. There are two signals – one with a positive effect (`sig`) on the pathway and the other one with a negative (`block`).*

The example model is also written in Figure 2.7, with its simplified scheme shown in Figure 2.6. It represents a signalling pathway where two external signals, `sig` and `block`, can transfer inside the cell. There, positive signal `sig` can react with protein `P1()` and activate it (denoted by changing its feature `active` from state `off` to `on`). On the other hand, the negative signal `block` can disrupt the activation of `P1()` by blocking its interaction with signal `sig`. Activated protein `P1()` can consequently activate key target protein `P2()`. Its activity is modelled by the same mechanism as in the case of protein `P1()`.

```
#! rules
// positive signal
sig{i}::ext ⇒ sig{a}::ext @ k_sig_1
sig{a}::ext ⇒ sig{a}::cell @ (k_sig_2×[sig{a}::ext])/(1+[block{a}::cell])

// blocking signal
block{i}::ext ⇒ block{a}::ext @ k_block_1
block{a}::ext ⇒ block{a}::cell @ (k_block_2×[block{a}::ext])/(1+[sig{a}::cell])

// signal and protein P1 interaction
sig{a}::cell + P1()::cell ⇒ sig{a}.P1()::cell @ param_sig×[sig{a}::cell]×[P1()::cell]
sig{_}.P1()::cell ⇒ sig{_}::cell + P1()::cell @ k_deg×[sig{_}.P1()::cell]
sig{a}.P1(active{off})::cell ⇒ sig{a}.P1(active{on})::cell @ 0.5×[sig{a}.P1(active{off})::cell]
sig{a}.P1()::cell ⇒ sig{a}::cell + P1()::cell @ k_deg×[sig{a}.P1()::cell]

// block and protein P1 interaction
block{a}::cell + P1()::cell ⇒ block{a}.P1()::cell @ param_block×[block{a}::cell]×[P1()::cell]

// proteins P1 and P2 interaction
P1(active{on})::cell + P2()::cell ⇒ P1(active{on}).P2()::cell @ 0.4×[P1(active{on})::cell]×[P2()::cell]
P1().P2()::cell ⇒ P1()::cell + P2()::cell @ k_deg×[P1().P2()::cell]
P1().P2(active{off})::cell ⇒ P1().P2(active{on})::cell @ k_prod×[P1().P2(active{off})::cell]

#! inits
sig{i}::ext
block{i}::ext
P1(active{off})::cell
P2(active{off})::cell

#! definitions
k_sig_1 = 0.8
k_sig_2 = 0.2
k_block_1 = 0.9
k_block_2 = 0.3
k_deg = 0.3
k_prod = 0.6
param_sig = 0.3
param_block = 0.4
```

Figure 2.7: *The example model named* `model.bcs` *used in this tutorial. It consists of 12 rules forming a signalling pathway, initial state, and definition of parameter constants. Note that example model* `model_param.bcs` *is identical except the values for the last two parameters* `param_sig` *and* `param_block` *are not given.*

Chapter 3

Analytical Tools

eBCSgen is equipped by several analytical tools: Transition system generator, Simulation, CTL Model checking, PCTL Model checking, PCTL Parameter synthesis, several Static analysis, and Probability Sampling. This chapter describes all of them in details and how to use them in Galaxy platform. Moreover, the Example model is used for their demonstration.

3.1 Transition System generator

In this section, we describe how to generate a transition system from given BCSL model. Following the idea of using approximate models with discrete-time semantics, the obtained transition system is a Discrete Time Markov Chain (DTMC) or a parametric Markov Chain (pMC) [7, 14], depending on whether there are some parameters used in rule rates which do not have defined value in definitions. A step by step Galaxy tour showing how to use the tool can be found here.

3.1.1 Input specification

- **Model file**: Selected model in BCSL language (see BCSL model syntax for details).
- **Bound** (optional): Bound represents the maximal multiplicity of any agent in any state. If not given, an optional bound is computed automatically for potentially infinite systems. It implies the construction of special "*hell*" state aggregating all states beyond the bound.
- **Advanced Options** (all optional)
  - **Maximal computation time**: Possibility to specify the time (in seconds) boundary for the computation.
  - **Maximal TS size**: Represents approximately the maximal number of nodes in the final transition system.
  - **Precomputed TS file**: Possibility to enter pre-generated transition system for the same model and continue in its generating. Usually connected with the usage of two previous arguments.

Example. To demonstrate the usage of the tool, we select `model.bcs` (see Example model) as Model file. All the other optional arguments are unspecified.

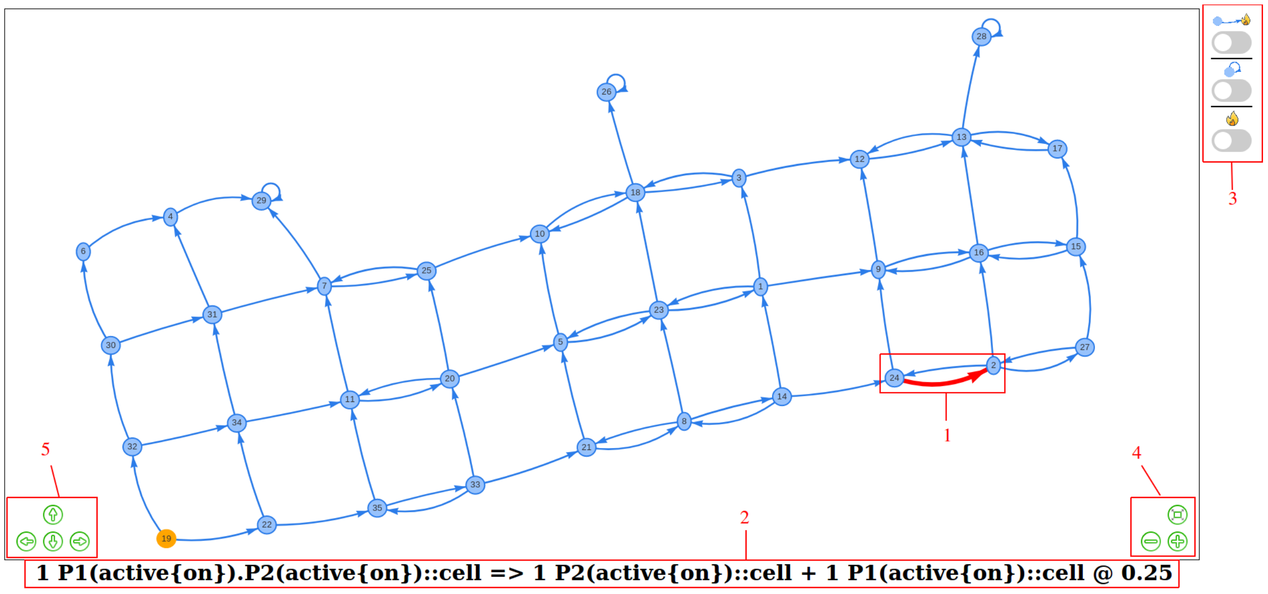
3.1.2 Output specification

Transition system generator tool generates a file format `.bcs.ts`, which is JSON with the following structure (order not given):

- **ordering**: Ordered list of all distinct agents.
- **nodes**: Set of nodes representing states of the transition system. Each of them contains a unique *ID* and a tuple of numbers, representing the multiplicity of the agent on the respective position from the ordering.
- **edges**: Set of edges representing transitions. Each edge contains ID of the *source* node (s), and ID of the *target* node (t), and the probability of the transition (p). In the case of the parametrised model, the probability is replaced by a probability function of parameters.
- **initial**: ID of the initial state.

3.1.3 Results visualisation

The transition system can be visualised as a network in an interactive chart called *Transition system visualisation* (see Figure 2.3 point 5 to learn how to visualise data). The nodes represent the states of the transition system, while the edges represent transitions between the particular states. It is possible to adjust the position of nodes by dragging them, highlight their content by clicking on them, and also display a particular transition by clicking on edge. The initial state from which was the transition system generated is shown in the orange colour.



1 P1(active{on}).P2(active{on})::cell => 1 P2(active{on})::cell + 1 P1(active{on})::cell @ 0.25

Figure 3.1: *Visualisation of the transition system computed for* `model.bcs`*. The key features are highlighted in red frames: (1) highlighted edge; (2) contents of the highlighted object; (3) options for graph details; (4) control panel for zooming options; (5) navigation panel.*

Figure 3.1 shows the visualisation for Example. The highlighted features are:

1. *highlighted edge* – edges and nodes can be highlighted by left-clicking on them.
2. *contents of the highlighted object* – the contents of the clicked node or edge are displayed. An edge contains particular interaction responsible for the transition with evaluated probability (or a probability function of parameters). A node contains agent counts present in the state.
3. *options for graph details* – it is possible to adjust three features of the graph by switch buttons: *top* – nodes with a transition to special state "*hell*" (if any) are highlighted by a squared border, *middle* – self-loops are hidden, *bottom* – special state "*hell*" (if any) is hidden.

4. *zooming options* – zooming and centring buttons. Zooming can also be achieved by mouse wheel.
5. *navigation panel* – vertical and horizontal navigation in the chart. It can also be achieved by left-clicking of mouse followed its dragging on an empty space in the chart.

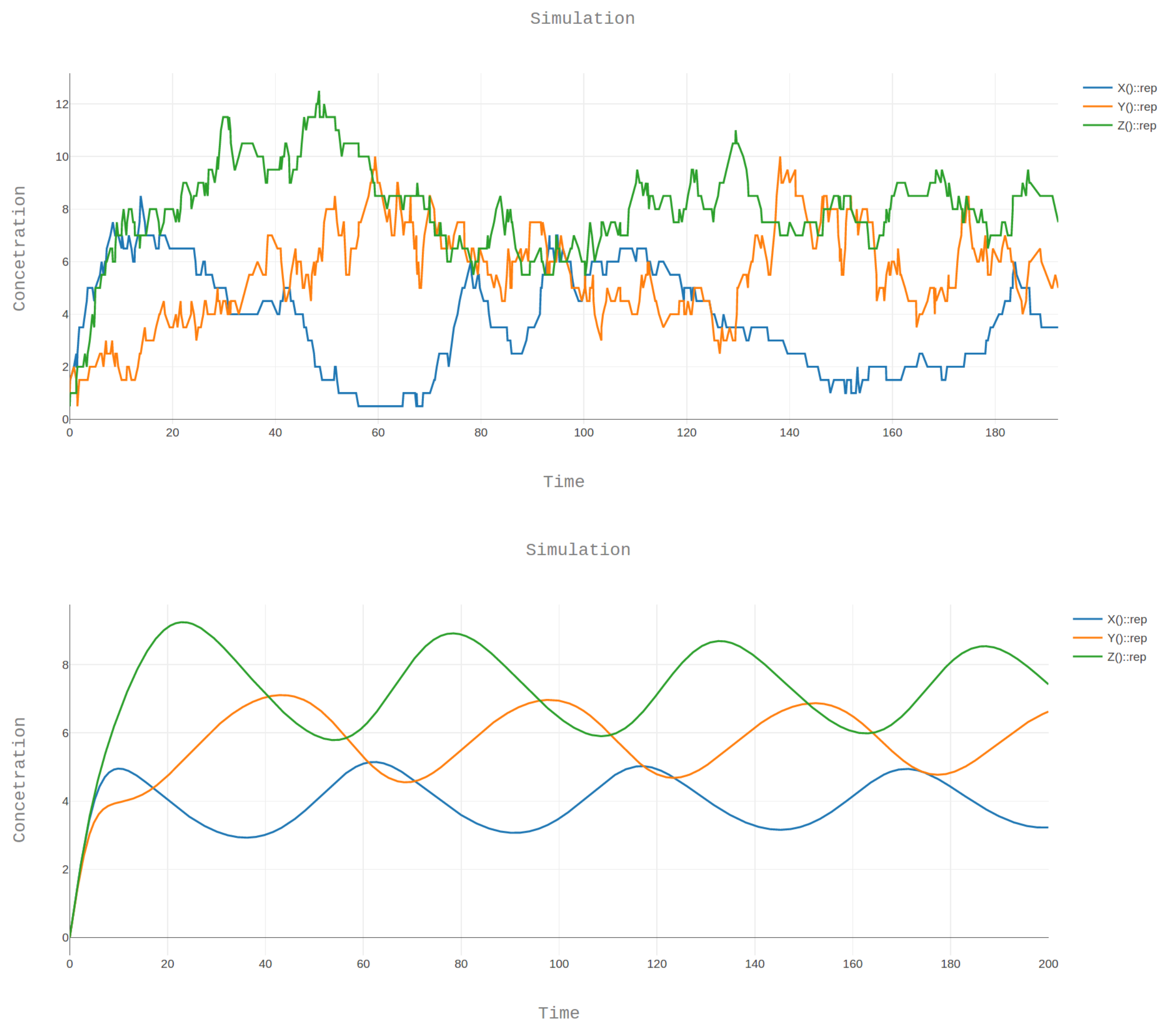


Figure 3.2: *Visualisation of time series data as a result of simulation tool. The top picture shows result of stochastic simulation from Example 1 and the bottom picture shows result of deterministic simulation from Example 2.*

3.2 Simulation

In this section, we describe how to simulate given BCSL model. Simulation tool provides a *stochastic simulation* of the model using an adapted variant of the standard Gillespie algorithm. Despite the natural probabilistic semantics, the *deterministic simulation* of the model is also possible. There are available step by step Galaxy tours showing how to use stochastic and deterministic variants of the tool.

3.2.1 Input specification

- **Model file**: Selected model in BCSL language (see BCSL model syntax for details).
- **Choose simulation method:**
  - For the **stochastic** simulation of the system, the **Number of runs** can be specified (default 1) to average individual runs (due to the stochastic nature of the method).
  - For the **deterministic** simulation of the system, a set of constructed ODEs is simulated. In this case, the **Volume** argument has to be filled in to deal with molecular counts in the initial state and **Time step** argument defining distance between two time points.
- **Maximum simulation time**: specification of simulation time.

Example 1. To demonstrate the usage of the tool for the case of *stochastic* simulation, we select `repressilator.bcs` (see Example model) as Model file. Then choose `2` runs and `200` for the simulation time.

Example 2. To demonstrate the usage of the tool for the case of *deterministic* simulation, we select `repressilator.bcs` (see Example model) as Model file. Then choose value `1` for the volume, `0.01` for the time step, and `200` for the simulation time.

3.2.2 Output specification

Simulation tool generates time series with respect to each agent. The result is stored in a `.csv` file, where the first column represents time, and the rest of the columns store values of for individual agents. Note that specified simulation time might not be achieved in the case of stochastic simulation when there are multiple runs due to averaging the runs.

3.2.3 Results visualisation

The simulation time series can be visualised in an interactive chart called *Simulation Plot* (see Galaxy UI basics to learn how to visualise data). Please note there are many available visualisations for this type of output (due to general `.csv` format) sorted alphabetically. The plot provides basic functionality such as zooming, curves filtering, and exporting `png` picture. The visualisation of both examples is available in Figure 3.2.

3.3 CTL Model Checking

CTL model Checking can be performed on *non*-parametric models with respect to the given CTL formula. The Model Checking procedure is performed using a package called `pyModelChecking`. A step by step Galaxy tour showing how to use the tool can be found here.

3.3.1 Input specification

- **TS file**: Generated transition system for the desired model (computed using Transition system generator).
- **CTL formula**: Given formula expressing property to be checked on the transition system.

Example. To demonstrate the usage of the tool, we select transition system computed for `model.bcs` (see Example model) as TS file. For CTL formula we choose `E(F([ P2(active{on})::cell > 0]))`.

3.3.2 Output specification

The textual result is stored in a `.ctl.check` file. The text contains number of states satisfying the formula and the boolean value for the initial state.

For our Example, we obtain boolean result `true`, as the formula is satisfied in the initial state.

3.4 PCTL Model Checking

PCTL model Checking can be performed on *non*-parametric models with respect to the given PCTL formula expressing a property regarding the probability of an event to occur. The tool allows checking whether a given probability threshold is satisfied or find the probability of satisfaction for the given path formula. The Model Checking procedure is performed using an external tool called `Storm`. A step by step Galaxy tour showing how to use the tool can be found here.

3.4.1 Input specification

- **TS file**: Generated transition system for the desired model (computed using Transition system generator).
- **PCTL formula**: Given formula expressing property to be checked on the transition system.

Example 1. To demonstrate the usage of the tool when the probability threshold of PCTL formula *is* given, we select transition system computed for `model.bcs` (see Example model) as TS file. For PCTL formula we choose `P <= 0.2 [F P2(active{on})::cell > 0]`.

Example 2. To demonstrate the usage of the tool when the probability threshold of PCTL formula is *not* given, we select transition system computed for `model.bcs` (see Example model) as TS file. For PCTL formula we choose `P =? [F P2(active{on})::cell > 0]`.

3.4.2 Output specification

The textual result is stored in a `.storm.check` file. The text contains some details about `Storm` model checker performance. Finally, the boolean or numerical result can be found as `Result (for initial states)`. Additionally, any errors `Storm` encountered are also shown in this file.

For our examples, we obtain boolean result `true` for Example 1 and probability `0.1304038484` for Example 2.

3.5 PCTL Parameter Synthesis

PCTL parameter Synthesis can be performed on *parametric* models with respect to the given PCTL formula expressing a property regarding the probability of an event to occur. If the formula has defined probability threshold, then the partitioning of the given parameter space (defined by the user) to regions which satisfy (resp. violate) the property is computed. If the threshold is not given, a probability function of parameters is computed instead, which evaluates to the probability of satisfaction for particular parametrisation. PCTL Parameter Synthesis procedure is performed using an external tool called `Storm`. A step by step Galaxy tour showing how to use the tool can be found here.

3.5.1 Input specification

- **TS file**: Generated transition system for the desired model (computed using Transition system generator).
- **PCTL formula**: Given formula expressing property to be used to perform parameter synthesis on the transition system.
- **Intervals**: Relevant only in the case when PCTL formula *has* defined probability threshold. Then, an interval of allowed values has to be specified for each unknown parameter in the model, together forming parameter space. If PCTL formula does *not* have defined probability threshold (i.e. starts with `P` =?), the intervals should *not* be defined.

Example 1. To demonstrate the usage of the tool when the probability threshold of PCTL formula *is* given, we select transition system computed for `model_param.bcs` (see Example model) as TS file. For PCTL formula we choose `P` <= 0.2 [`F P2(active{on})`::`cell` > `0`] and enter the following intervals – `param_sig`: $[0.1, 0.6]$ and `param_block`: $[0.05, 1.0]$.

Example 2. To demonstrate the usage of the tool when the probability threshold of PCTL formula is *not* given, we select transition system computed for `model_param.bcs` (see Example model) as TS file. For PCTL formula we choose `P` =? [`F P2(active{on})`::`cell` > `0`], and intervals are not specified.

3.5.2 Output specification

The textual result contains some details about `Storm` model checker performance. It is stored in a `.storm.sample` or `.storm.regions` file, for *with* and *without* threshold respectively.

`.storm.sample` file contains computed probability function of parameters, which can be evaluated to the probability of satisfaction for particular parametrisation. This result can be further analysed in Probability Sampling tool (demonstrated using Example 2).

`.storm.regions` file contains segmentation of specified parameter space to regions where the given property is satisfied and violated. For some of the regions, the satisfiability might not be decided due to efficiency reasons.

3.5.3 Results visualisation

The Parameter Synthesis results stored in `.storm.regions` file can be visualised in *Parameter synthesis result visualisation* (see Galaxy UI basics to learn how to visualise data). The visualisation shows green, red, and grey regions where the satisfiability of PCTL formula is True, False, and unknown, respectively. The visualisation allows user to change chosen parameters on X and Y axis, which is particularly useful when there are more than two parameters. In that case, slices of parameter space in other dimensions can be chosen. The visualisation of Example 1 is available in Figure 3.3 *left*.

3.6 Probability Sampling

Probability Sampling is a tool dedicated for the sampling of computed probability function of parameters as a result of Parameter Synthesis. A step by step Galaxy tour showing how to use the tool can be found here.

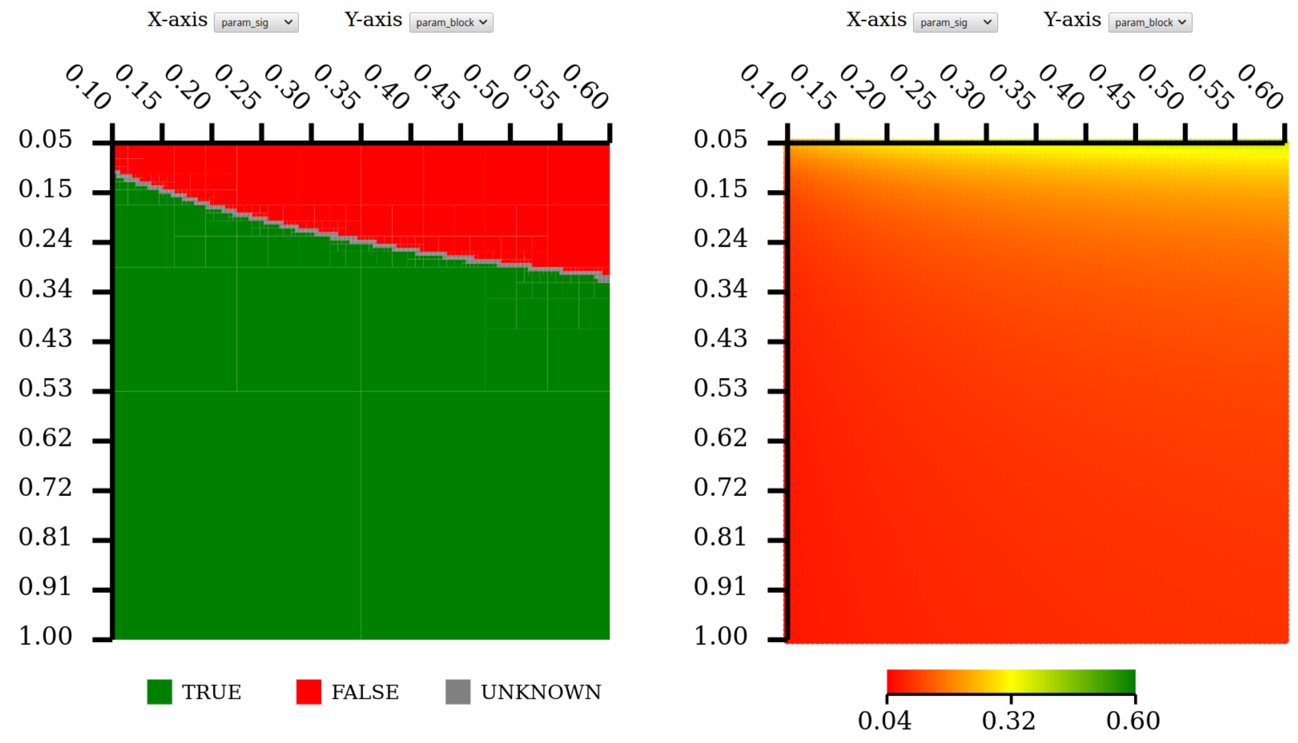


Figure 3.3: *Visualisation of results of parameter synthesis. The* left *picture depicts partitioning of parameter space as a result of parameter synthesis* with *a defined probability threshold (0.2 in this case). The* right *picture depicts the sampling of the probability function of parameters as a result of parameter synthesis* without *a defined probability threshold. Both pictures display the same parameter space for the* `model_param.bcs`

3.6.1 Input specification

- **Storm results file**: Selected PCTL Parameter Synthesis result in `.storm.sample` format.
- **Intervals**: An interval of allowed values have to be specified for each unknown parameter in the function, together forming parameter space to be sampled. Moreover, for each interval, it has to be specified how many samples should be taken.

Example. To demonstrate the usage of the tool, we select the output file of Example 2 from Parameter Synthesis) as Storm results file. We enter the following intervals – `param_sig`: $[0.1, 0.6]$ with 101 samples and `param_block`: $[0.05, 1.0]$ with 101 samples.

3.6.2 Output specification

The result of this tool is an `html` file containing a graphical output of the sampling. The output is similar to visualisation mentioned in Parameter Synthesis, but the legend of the visualisation is different – there is a scale assigning a colour to probability values. It can be viewed directly (see Figure 2.3, item 6). The visualisation of Example is available in Figure 3.3 *right*.

3.7 Static Analysis

Static analysis techniques are performed directly on the model syntax and are therefore extremely fast by definition.

In particular, we have developed *unreachability* static analysis method which can be used to check whether a single agent is unreachable before enumerating the entire transition system of the model. This analysis is based on the idea that in order to reach an agent, there has to be a rule which creates it or a lesser specified agent; in other words, the particular states inside of the agent are changed at some point. A step by step Galaxy tour showing how to use the tool can be found here.

The second method aims to *detect* (potentially) *redundant* rules in the model. This can be useful in large models to detect potentially concurrent rules and to make the model more compact. The redundant rules are denoted by adding comments at the end of such potentially redundant rules and connecting them by an integer identifier. A step by step Galaxy tour showing how to use the tool can be found here.

The third static analysis technique is used to *reduce the context* of the model to a minimal level in order to produce a smaller and more abstract model. The resulting model still preserves some properties while making the analysis of the model computationally simpler. A step by step Galaxy tour showing how to use the tool can be found here.

3.7.1 Input specification

- **Model file**: Selected model in BCSL language (see BCSL model syntax for details).
- **Choose static analysis method**:
  - **Static unreachability**: This option computes unreachability for the particular agent given in **Complex agent**.
  - **Rule redundancy elimination**: This option marks pairs of rules which might be redundant in the model.
  - **Context based reduction**: This option removes all states from all agents in the model, reducing context to a minimum. Rule with no sense (e.g. rules where left- and right-hand sides are equal) are removed.

Example 1. To demonstrate the Static unreachability method, we select `model.bcs` (see Example model) as Model file and enter the expression `sig{i}.P1()::cell` as Complex agent.

Example 2. To demonstrate the Rule redundancy elimination method, we select `model.bcs` (see Example model) as Model file.

Example 3. To demonstrate the Context based reduction method, we select `model.bcs` (see Example model) as Model file.

3.7.2 Output specification

The results of the tool depend on the particular method which was chosen.

For Static unreachability method, the output is textual and states whether the agent of interest *cannot be reached* and *can possibly be reached* in the model. For Example 1, the output states:

```
The given agent
        sig{i}.P1()::cell
cannot be reached in the model.
```

For Rule redundancy elimination method, the output is a `.bcs` model. In the first case, just some comments are added to the file indicating potential (if any) redundant rules. For Example 2, the output model contains two rules which are potentially redundant:

```
sig{a}.P1()::cell ⇒ sig{a}::cell + P1()::cell @ k_deg×[sig{a}.P1()::cell] // redundant #{1}
sig{_}.P1()::cell ⇒ sig{_}::cell + P1()::cell @ k_deg×[sig{_}.P1()::cell] // redundant #{1}
```

For Context based reduction method, the output is a `.bcs` model. In the output model, all states are absent, leaving only rules for complex formation and dissociation, production, and degradation. For Example 3, we recommend to run Transition system generator and compare visualisations for the original and reduced model. The transition system of the reduced model is significantly smaller but also preserves fewer details.

3.8 SBML export

SBML standard is a key medium for exchanging models in systems biology. However, encoding rule-based, domain-detailed models using the concepts of Species and Compartment in SBML can be difficult for models with complex rules that lead to large numbers of particular reactions. For that, SBML Level 3 package: Multistate, Multicomponent and Multicompartment Species extends the SBML Level 3 core with the “type” concept in the Species and Compartment classes and therefore reaction rules may contain species that can be patterns and be in multiple locations in reaction rules. It allows SBML standard for encoding rule-based models using their “native” concepts for describing reactions instead of having to apply the rules and unfold the networks prior to encoding in an SBML format.

This tool allows to export a BCSL model in SBML format supporting the `multi` package. A step by step Galaxy tour showing how to use the tool can be found here.

3.8.1 Input specification

- **Model file**: Selected model in BCSL language (see BCSL model syntax for details).

3.8.2 Output specification

The results of the tool is an XML file containing equivalent model in SBML format.

Chapter 4

Regulations

Regulation mechanisms are a form of an extension to a model. In general, regulations influence conditions when a rewriting rule can be used, which in consequence affects the nondeterministic behaviour of the model. The regulations allow compactly modelling the additional knowledge about the system, otherwise too laborious or even impossible to express. We support five types of regulations – regular, programmed, ordered, concurrent-free, and conditional. They allow expressing mutual relationships and dependencies among rules. To comfortably use the regulations, it is necessary to specify `LABEL` for rules (at least for those which will be used in a regulation). One regulation type per model is allowed.

We are not able to provide a comprehensive guide on what regulation should be used to achieve the desired behaviour. We do not have a particular recipe since it is very tightly coupled to the particular case. Instead, we explain individual regulations and their application case by case in section Types of regulations. On a running example, we show effects of regulation on the respective transition system (which can be inspected visually due to their minimalistic nature) and using CTL model checking we confirm that desired behaviour was achieved. All CTL model checking analyses performed on transition system of the respective regulated model are available in the Shared History. Moreover, we provide several Regulation exercises to let the reader drill the regulation effects on additional examples (with the correct answers on the last page).

In Figure 4.1, a grammar for each type of regulation is given. The usage of regulation is optional. Moreover, in Figure 4.2 we provide an illustrative example in the form of regulated version of model from Figure 2.7 with usage of a *conditional* regulation. We can see that the model becomes more readable and compact because some modelling details can be abstracted out using the regulation.

```
model: rules inits? definitions? complexes? regulation?
...
regulation: "#! regulation" regulation_def
regulation_def: "type" ( regular | programmed | ordered | concurrent_free | conditional )

regular: "regular" expression
expression: LABEL | expression "|" expression | expression "." expression | expression"*"

programmed: "programmed" successors+
successors: LABEL ":" "{" LABEL ("," LABEL)* "}"

ordered: "ordered" order ("," order)*
order: ("(" LABEL "," LABEL ")")

concurrent_free: "concurrent-free" order ("," order)*

conditional: "conditional" context+
context: LABEL ":" "{" rate_complex ("," rate_complex)* "}"
```

Figure 4.1: *Extension of BCSL grammar by regulations.*

4.1 Example of regulated model

In Figure 4.2 we provide a variant of the model from Figure 2.7 with the usage of regulations. A *conditional* regulation was applied to substitute laborious mechanism of `block` signal with a simpler and more straightforward approach. Such an approach is especially useful when the particular mechanism is not known in detail. The regulation ensures that rule label `r1` cannot be used when there is agent `block{a}::cell` present in the current state.

```
#! rules
// positive signal
sig{i}::ext ⇒ sig{a}::ext @ k_sig_1
sig{a}::ext ⇒ sig{a}::cell @ (k_sig_2×[sig{a}::ext])/(1+[block{a}::cell])

// activation blocking signal
block{i}::cell ⇒ block{a}::cell @ param_block

// signal and protein P1 interaction
sig{a}::cell + P1()::cell ⇒ sig{a}.P1()::cell @ param_sig×[sig{a}::cell]×[P1()::cell]
sig{_}.P1()::cell ⇒ sig{_}::cell + P1()::cell @ k_deg×[sig{_}.P1()::cell]
sig{a}.P1(active{off})::cell ⇒ sig{a}.P1(active{on})::cell @ 0.5×[sig{a}.P1(active{off})::cell]
sig{a}.P1()::cell ⇒ sig{a}::cell + P1()::cell @ k_deg×[sig{a}.P1()::cell]

// proteins P1 and P2 interaction
r1 ~ P1(active{on})::cell + P2()::cell ⇒ P1(active{on}).P2()::cell @ 0.4×[P1(active{on})::cell]×[P2()::cell]
P1().P2()::cell ⇒ P1()::cell + P2()::cell @ k_deg×[P1().P2()::cell]
P1().P2(active{off})::cell ⇒ P1().P2(active{on})::cell @ k_prod×[P1().P2(active{off})::cell]

#! inits
sig{i}::ext
block{i}::ext
P1(active{off})::cell
P2(active{off})::cell

#! definitions
k_sig_1 = 0.8
k_sig_2 = 0.2
k_deg = 0.3
k_prod = 0.6
param_sig = 0.3
param_block = 0.4

#! regulation
type conditional
r1: {block{a}::cell}
```

Figure 4.2: *The example model from Figure 2.7 with applied conditional regulation. The regulation ensures that rule `r1` (interaction of `P1` and `P2`) is allowed only when the blocking signal is* not *activated. This allowed us to eliminate other rules related to the handling of `block` molecule. Additionally, this step enables the parameter `param_block` to directly influence the effects of `block` molecule on a quantitative level (with no extra steps introduced by parameters from intermediate rules).*

4.2 Types of regulations

In this section of the tutorial, we explain individual regulation classes and on a running example (Figure 4.3), we demonstrate their applicability. While the minimalistic nature of the model allows validating the effects of regulation by visual exploration of the transition system, we also support the claims about its behaviour using CTL model checking.

The model consists of a single molecule `P` and three rules which can modify it. The molecule has two domains `S` and `T` that can be independently activated by the respective rule. Third rule can export the molecule from compartment `cell` to compartment `out`. Initially, there is a single `P` molecule present with both domains inactive. Assuming the initial state, the transition system corresponding to its behaviour is available in Figure 4.4.

```
#! rules
r1_S ~ P(S{i})::cell ⇒ P(S{a})::cell
r1_T ~ P(T{i})::cell ⇒ P(T{a})::cell
r2   ~ P()::cell     ⇒ P()::out

#! inits
1 P(S{i},T{i})::cell
```

Figure 4.3: *Running BCSL model for demonstration of regulations.*

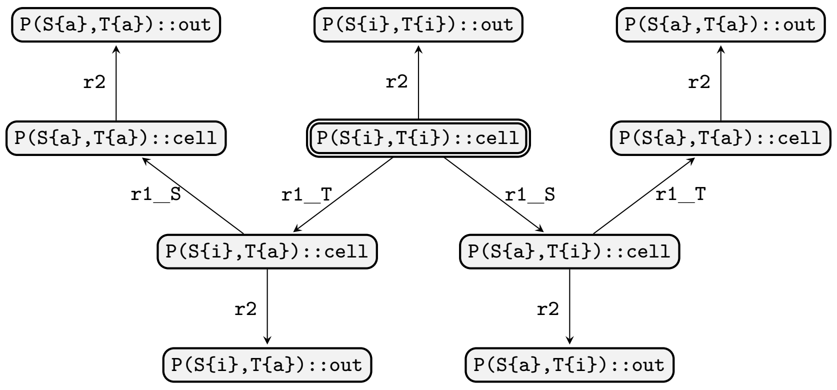

$$\begin{tikzcd}
\texttt{P(S\{a\},T\{a\})::out} & \texttt{P(S\{i\},T\{i\})::out} & \texttt{P(S\{a\},T\{a\})::out} \\
\texttt{P(S\{a\},T\{a\})::cell} \arrow[u, "r2"] & \boxed{\texttt{P(S\{i\},T\{i\})::cell}} \arrow[u, "r2"] \arrow[dl, "r1\_T"] \arrow[dr, "r1\_S"] & \texttt{P(S\{a\},T\{a\})::cell} \arrow[u, "r2"] \\
\texttt{P(S\{i\},T\{a\})::cell} \arrow[u, "r1\_S"] \arrow[d, "r2"] & & \texttt{P(S\{a\},T\{i\})::cell} \arrow[u, "r1\_T"] \arrow[d, "r2"] \\
\texttt{P(S\{i\},T\{a\})::out} & & \texttt{P(S\{a\},T\{i\})::out}
\end{tikzcd}$$

Figure 4.4: *Transition system of the model from Figure 4.3. The double circled state is the initial state. The transitions are labelled by a rule which was used to enable the transition. Individual model runs correspond to the particular paths starting in the initial state.*

4.2.1 Regular rewriting

Let us assume the model from Figure 4.3 does not meet our modelling intentions. We want to tweak conditions when the molecule `P` can be exported outside of the `cell` using rule `r2`. The goal is to make this rule dependent on the previously used rules and allow it to be executed *only* after rules `r1_S`, `r1_T` were used in this particular sequence.

To express such special conditions, we can use a regular expression (RE). RE is a pattern that defines a set of sequences over a given domain. Such an approach can define sequences of rules that are allowed explicitly. These, in practice, describe particular executions of the model. We can achieve that by defining regular expressions over rules.

The most general RE has form $\{\mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_3, \dots\}^*$, which allows any possible sequence of rules, corresponding to unregulated behaviour. On the other hand, the least permissive REs have form $\mathtt{r}_1\mathtt{r}_2\mathtt{r}_3$ which allows only this particular sequence of rules to be used. Then, some more variable options can include for example choice from a set of candidates on some fixed positions $\mathtt{r}_1\{\mathtt{r}_2, \mathtt{r}_4\}^*\mathtt{r}_3$ or specification of several alternatives using boolean "or" $\mathtt{r}_1\mathtt{r}_2|\mathtt{r}_3\mathtt{r}_4$.

To achieve the behaviour described above, we can introduce the following section to our model from Figure 4.3:

```
#! regulation
type regular
r1_S r1_T r2 | r1_T r1_S
```

It allows either a sequence of three rules `r1_S`, `r1_T`, and `r2` or a sequence of two rules `r1_T` and `r1_S`. This RE ensures that both activation rules are first used and then the molecule is exported out of the cell, depending on the order of activation. The effect of regulation on the transition system is depicted in Figure 4.5.

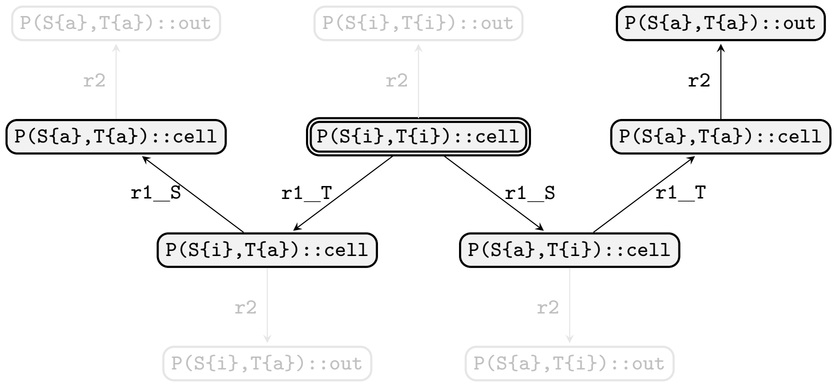


Figure 4.5: *Transition system of the model from Figure 4.3 with applied regular regulation. The double circled state is the initial state, the grey states and transition are absent due to the effects of the regulation. The regulation allows introducing an asymmetry to an otherwise symmetric transition system.*

To support the claims made by inspecting the transition system from Figure 4.5, we run CTL model checking on it. First, we want to make sure that desired behaviour was preserved, i.e. that a fully activated molecule `P` can be exported outside of the cell. This can be expressed using CTL formula from Equation 4.1. This formula is evaluated to `True` value in the respective transition system.

$$\mathtt{EF}\ [\ \mathtt{P(S\{a\},T\{a\})::out} > 0\ ] \tag{4.1}$$

We need to make sure that undesired behaviour was eliminated from the model. In particular, no other form of molecule `P` (i.e. partially inactivate) should be reachable. The respective positive reachability property is expressed using CTL formula in Equation 4.2. The formula is evaluated to `False` value (i.e. such form is not reachable), which corresponds to the expected behaviour.

$$\texttt{EF}\ [\ \texttt{P(S\{i\})}::\texttt{out} > 0 \mid \texttt{P(T\{i\})}::\texttt{out} > 0\ ] \tag{4.2}$$

Finally, in the unregulated case of the model, two branches are leading to the export of the active form of molecule `P`. However, our regulation was supposed to eliminate one of them. We need to make sure this was also achieved. The CTL formula from Equation 4.3 captures this property, by conditioning reachability of the fully active form by first reaching `P(S{i},T{a})::cell` form in the next state from the initial one. This property is evaluated to `False` value, which again corresponds to the expected behaviour and concludes our analysis.

$$\texttt{EX}\ [\ \texttt{P(S\{i\},T\{a\})}::\texttt{cell} > 0\ \&\ \texttt{EF}\ [\ \texttt{P(S\{a\},T\{a\})}::\texttt{out} > 0\ ]\ ] \tag{4.3}$$

4.2.2 Ordered rewriting

Let us assume the model from Figure 4.3 does not meet our modelling intentions. We want to tweak conditions when the molecule `P` can be exported outside of the `cell` using rule `r2`. The goal is to allow such export only before the molecule was modified by any rules `r1_S`, `r1_T`. Once the molecule is partially activated, it cannot leave the cell.

Ordering of arbitrary elements allows assuming only sequences that respect this order in some manner. We use such an approach to define an order on rules and then require that any rule sequence has to respect it in a pair-wise fashion. In other words, the latter rule is greater than the former in any pair of subsequent rules. Moreover, we assume so-called partial order, which allows some rules to be incomparable. Incomparable rules can be applied in an arbitrary order.

To achieve the behaviour described above, we can introduce the following section to our model from Figure 4.3:

```
#! regulation
type ordered
(r1_S, r2), (r1_T, r2)
```

It consists of an explicit enumeration of pairs of rules, where the first member of the pair is higher in the order than the second one. It can be interpreted as `r1_S` > `r2` and `r1_T` > `r2` – both rules `r1_S` and `r1_T` are greater than `r2` while rules `r1_S` and `r1_T` are incomparable. This order makes sure that rule `r2` is never used immediately after neither `r1_S` nor `r1_T`. In other words, it can be only used before them. The effect of regulation on the transition system is depicted in Figure 4.6.

To support the claims made by inspecting the transition system from Figure 4.6, we run CTL model checking on it. First, we need to express that the inactive form of molecule `P` can be actually exported outside of the cell. This is captured in CTL formula from Equation 4.4, which is evaluated to `True` value in the transition system.

$$\texttt{EF}\ [\ \texttt{P(S\{i\},T\{i\})}::\texttt{out} > 0\ ] \tag{4.4}$$

Finally, we also need to make sure that all the other exports are not allowed. The CTL formula from Equation 4.5 capture the property that all forms with an active domain can be exported from the cell. This, as expected, is evaluated to `False`, validating that the transition system corresponds to the modelling intentions.

$$\texttt{EF}\ [\ \texttt{P(S\{a\})}::\texttt{out} > 0 \mid \texttt{P(T\{a\})}::\texttt{out} > 0\ ] \tag{4.5}$$

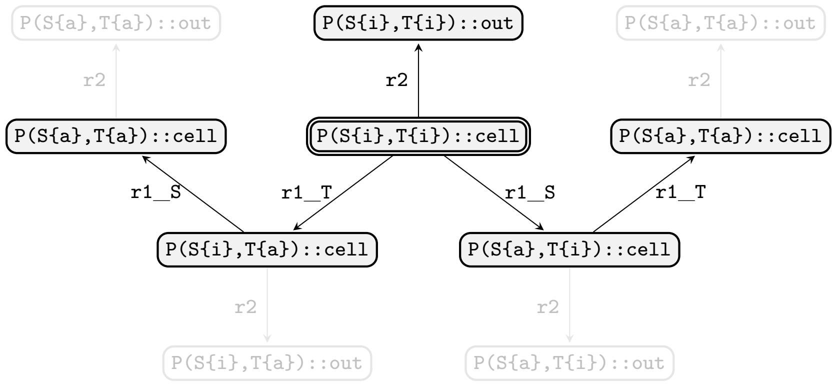


Figure 4.6: *Transition system of the model from Figure 4.3 with applied ordered regulation, ensuring the rule* `r2` *is never used immediately after rule* `r1_S` *or* `r1_T`*. The double circled state is the initial state, the grey states and transition are absent due to the effects of the regulation.*

4.2.3 Programmed rewriting

Let us assume the model from Figure 4.3 does not meet our modelling intentions. We want to tweak conditions when the molecule `P` can be exported outside of the `cell` using rule `r2`. The goal is to allow exporting the molecule only immediately after it was activated on its `S` domain. In other words, it *cannot* be exported immediately after it was activated on its `T` domain.

A successor function of a rule defines the rules which can be used in the next step. This allows limiting the set of possible successors to an admissible subset. When a particular rule is used, only its successors are allowed to be used in the next step.

For a particular rule, it is possible to define as successors the complete set of rules (no restrictions), a proper subset (some of the rules cannot be used immediately after), or an empty set, which makes the rule terminal (no other action can be done after).

To achieve the behaviour described above, we can introduce the following section to our model from Figure 4.3:

```
#! regulation
type programmed
r1_S: {r1_T, r2}
r1_T: {r1_S}
```

It consists of possibly multiple lines containing a set of successor rules for one rule per line. We can see that for rule `r2` there are no successors, which is implicitly interpreted as an empty set. This function ensures that rule `r2` is used only after rule `r1_S`, never after rule `r1_T`. The effect of regulation on the transition system is depicted in Figure 4.7.

To support the claims made by inspecting the transition system from Figure 4.6, we run CTL model checking on it. From the visual inspection, we can see that there is an asymmetry in the export of partially versus fully active molecule `P`. Therefore we need to check all four cases individually. First, we consider the "left" branch, starting with `P(S{i},T{a})::cell` form. Since this form was achieved by activating domain `T`, the molecule cannot be exported directly in this form. The respective property is expressed in CTL formula from Equation 4.6, which is evaluated to `False` as expected.

$$\texttt{EX}\ [\ \texttt{P(S\{i\},T\{a\})}::\texttt{cell} > 0\ \&\ \texttt{EF}\ [\ \texttt{P(S\{i\},T\{a\})}::\texttt{out} > 0\ ]\ ] \tag{4.6}$$

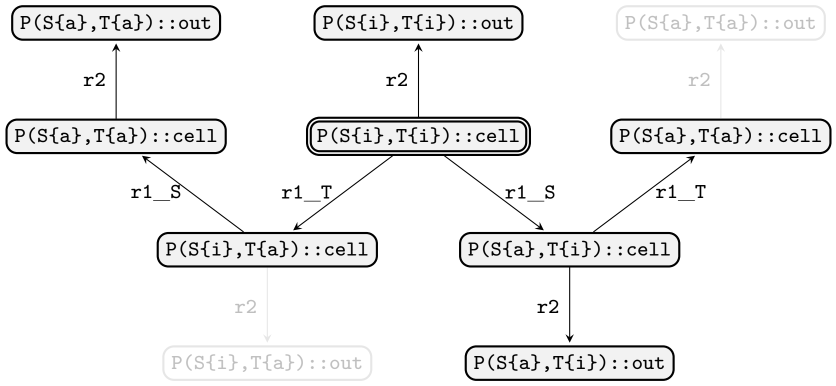


Figure 4.7: *Transition system of the model from Figure 4.3 with applied programmed regulation, making sure the rule* `r2` *is never used after rule* `r1_T`*. The double circled state is the initial state, the grey states and transition are absent due to the effects of the regulation.*

Continuing in this branch, the domain `S` can be activated and followed by the export from the cell. The formula from Equation 4.7 is evaluated to `True`, validating that such a form can be reached.

$$\texttt{EX}\ [\ \texttt{P(S\{i\},T\{a\})}::\texttt{cell} > 0\ \&\ \texttt{EF}\ [\ \texttt{P(S\{a\},T\{a\})}::\texttt{out} > 0\ ]\ ] \tag{4.7}$$

Next, we consider the "right" branch, starting with `P(S{a},T{i})::cell` form of the molecule. State with this molecule variant achieved by activating domain `S`, which allows it to be exported immediately. This property is expressed in CTL formula from Equation 4.8, which is evaluated to `True`.

$$\texttt{EX}\ [\ \texttt{P(S\{a\},T\{i\}\})}::\texttt{cell} > 0\ \&\ \texttt{EF}\ [\ \texttt{P(S\{a\},T\{i\})}::\texttt{out} > 0\ ]\ ] \tag{4.8}$$

Finally, the other option is to continue by activating domain `T`, which disables the export from the cell. The respective property is expressed in CTL formula from Equation 4.9, which is evaluated to `False`. The regulated behaviour corresponds to our modelling intentions.

$$\texttt{EX}\ [\ \texttt{P(S\{a\},T\{i\})}::\texttt{cell} > 0\ \&\ \texttt{EF}\ [\ \texttt{P(S\{a\},T\{a\})}::\texttt{out} > 0\ ]\ ] \tag{4.9}$$

4.2.4 Conditional rewriting

Let us assume the model from Figure 4.3 does not meet our modelling intentions. We want to tweak conditions when the molecule `P` can be exported outside of the `cell` using rule `r2`. The goal is to disable the exporting of the molecule only in a particular context (i.e. contents of the current state). In particular, we do not want it to be exported when `S` domain is activated, and `T` domain is deactivated.

In general, a rule can be used if there are enough molecules in a state where it is being applied. A natural extension to this mechanism is to define molecules, which, when present in the state, prohibit the usage of the rule.

Conditional rewriting defines a *prohibited* context to each rule, that is, a set of molecules that cannot be present in the current state. If present, the rule cannot be used. If the defined context is an empty set, rule applicability has no limitations.

To achieve the behaviour described above, we can introduce the following section to our model from Figure 4.3:

```
#! regulation
type conditional
r2: {P(S{a},T{i})::cell}
```

It consists of possibly multiple lines containing forbidden context for one rule per line. If there is no context defined for a rule (both `r1_T` and `r1_S` in this case), their prohibited context is empty by default (there are no restrictions). This regulation makes sure that rule `r2` is never used when molecule `P` has particular form `P(S{a},T{i})::cell`. The effect of regulation on the transition system is depicted in Figure 4.8.

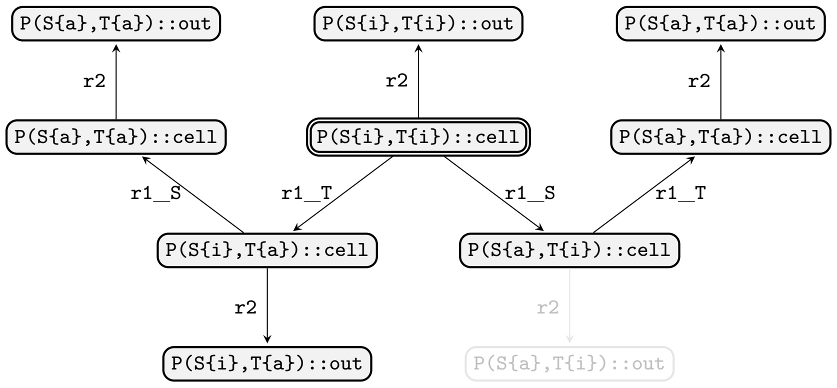


Figure 4.8: *Transition system of the model from Figure 4.3 with applied conditional regulation. The double circled state is the initial state, the grey states and transition are absent due to the effects of the regulation. The rule* `r2` *is forbidden to be used in a particular context but is still allowed in other cases.*

We will just briefly validate using CTL model checking that the specified form of `P` is actually not reachable. In Equation 4.10 there is a CTL formula expressing the reachability of such molecule, and, as expected, it is evaluated to `False`, confirming that the modelling goal was achieved.

$$\texttt{EF}\ [\ \texttt{P(S\{a\},T\{i\})::out} > 0\ ] \tag{4.10}$$

4.2.5 Concurrent-free rewriting

Let us assume the model from Figure 4.3 does not meet our modelling intentions. We want to tweak conditions when the molecule `P` can be exported outside of the `cell` using rule `r2`. The goal is to disable the exporting of the molecule whenever it can be still activated. In other words, the activation is always given priority over the export.

Concurrency is a natural process in biology, and controlling concurrent processes may be desirable. In the context of rules, such processes are expressed by rules which consume mutual molecules. We allow assigning a priority to one of them for such concurrent rules. Only the prioritised one can be used whenever multiple concurrent rules are applicable in a state.

To achieve the behaviour described above, we can introduce the following section to our model from Figure 4.3:

```
#! regulation
type concurrent-free
(r1_S, r2), (r1_T, r2)
```

It consists of an explicit enumeration of pairs of rules, where the first member of the pair has priority over the second one. Informally, this regulation can be read as `r1_S` > `r2` and `r1_T` > `r2`, meaning that rules `r1_S` and `r1_T` have always priority over rule `r2`. The effect of regulation on the transition system is depicted in Figure 4.9.

From the visual inspection of the respective transition system, it is obvious that the regulation effectively eliminated the export of all other forms of molecule `P` except those fully activated. In terms of CTL model

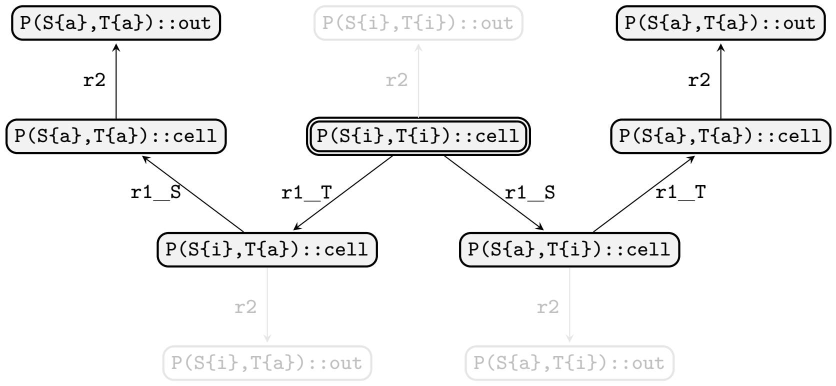


Figure 4.9: *Transition system of the model from Figure 4.3 with applied concurrent-free regulation. The double circled state is the initial state, the grey states and transition are absent due to the effects of the regulation. Whenever one of the rules* `r1_T`, `r1_S` *is enabled alongside with* `r2`*, it is given priority.*

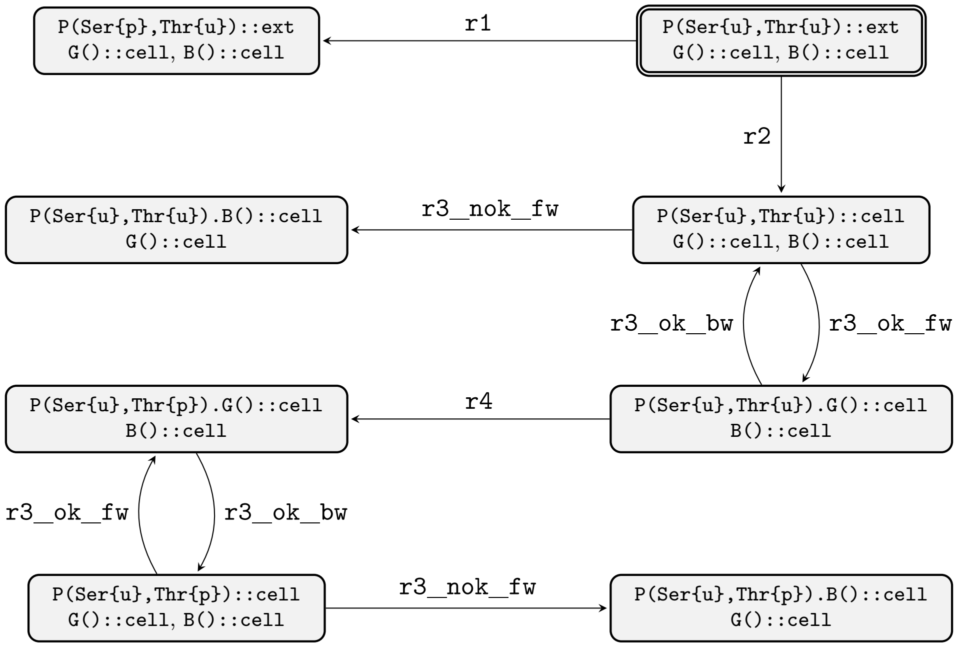
checking, this can be confirmed by a formula expressing that it always holds that eventually, the fully active form of molecule `P` will be reached. Such a property is expressed in Equation 4.11, and its evaluation to `True` value validates our modelling intentions.

$$\texttt{AF}\ [\ \texttt{P(S\{a\},T\{a\})}::\texttt{out} > 0\ ] \tag{4.11}$$

4.3 Regulation exercises

We have given the following BCSL model, with defined set of rules and initial state. Its corresponding transition system is displayed below.

```
#! rules
r1          ~ P(Ser{u})::ext ⇒ P(Ser{p})::ext
r2          ~ P(Ser{u})::ext ⇒ P(Ser{u})::cell

r3_ok_fw    ~ P()::cell + G()::cell ⇒ P().G()::cell
r3_ok_bw    ~ P().G()::cell ⇒ P()::cell + G()::cell

r3_nok_fw ~ P()::cell + B()::cell ⇒ P().B()::cell

r4          ~ P(Thr{u}).G()::cell ⇒ P(Thr{p}).G()::cell

#! inits
1 P(Ser{u},Thr{u})::ext
1 G()::cell
1 B()::cell
```



P(Ser{p},Thr{u})::ext G()::cell, B()::cell
r1
P(Ser{u},Thr{u})::ext G()::cell, B()::cell
r2
P(Ser{u},Thr{u}).B()::cell G()::cell
r3_nok_fw
P(Ser{u},Thr{u})::cell G()::cell, B()::cell
r3_ok_bw
r3_ok_fw
P(Ser{u},Thr{p}).G()::cell B()::cell
r4
P(Ser{u},Thr{u}).G()::cell B()::cell
r3_ok_fw
r3_ok_bw
P(Ser{u},Thr{p})::cell G()::cell, B()::cell
r3_nok_fw
P(Ser{u},Thr{p}).B()::cell G()::cell

Class	Regulation
regular	run labels belong to regular language
ordered	subsequent rules are restricted by partial order
programmed	successors of rules are explicitly specified
conditional	rules cannot be used in context of specified elements
concurrent-free	priority is given to one of the concurrent rules

4.3.1 Task 1

Which of the following regulations will ensure that molecule `P` will *not* get stuck in `ext` location (i.e. achieve the behaviour displayed below):

A. `conditional`

   `r1` : { `P(Ser{u},Thr{u})`::`ext` }

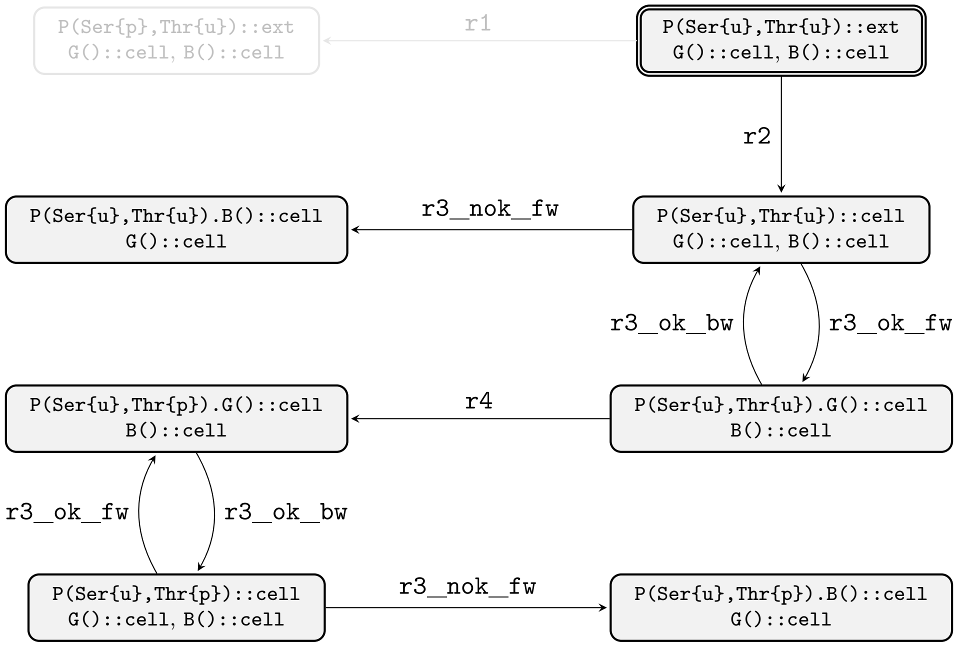
B. `ordered`

   {(`r2`, `r1`)}   (`r2` > `r1`)

C. `concurrent-free`

   {(`r2`, `r1`)}   (informally `r2` > `r1`)

There might be more than one correct answer.
Can you find any other solutions?
Is it possible to use `programmed` regulation?



P(Ser{p},Thr{u})::ext G()::cell, B()::cell
r1
P(Ser{u},Thr{u})::ext G()::cell, B()::cell
r2
P(Ser{u},Thr{u}).B()::cell G()::cell
r3_nok_fw
P(Ser{u},Thr{u})::cell G()::cell, B()::cell
r3_ok_bw
r3_ok_fw
P(Ser{u},Thr{p}).G()::cell B()::cell
r4
P(Ser{u},Thr{u}).G()::cell B()::cell
r3_ok_fw
r3_ok_bw
P(Ser{u},Thr{p})::cell G()::cell, B()::cell
r3_nok_fw
P(Ser{u},Thr{p}).B()::cell G()::cell

4.3.2 Task 2

Which of the following regulations will ensure that molecule `P` never creates complex with molecule `B` (i.e. achieve the behaviour displayed below):

A. `programmed`

   `r2` : {`r3_ok_fw`},

   `r3_ok_fw` : {`r3_ok_bw`, `r4`},

   `r4` : {`r3_ok_bw`},

   `r3_ok_bw` : {`r3_ok_fw`}

B. `regular`

   `r1` | `r2` . (`r3_ok_fw` . `r3_ok_bw`)*. `r4` . (`r3_ok_fw` . `r3_ok_bw`)*

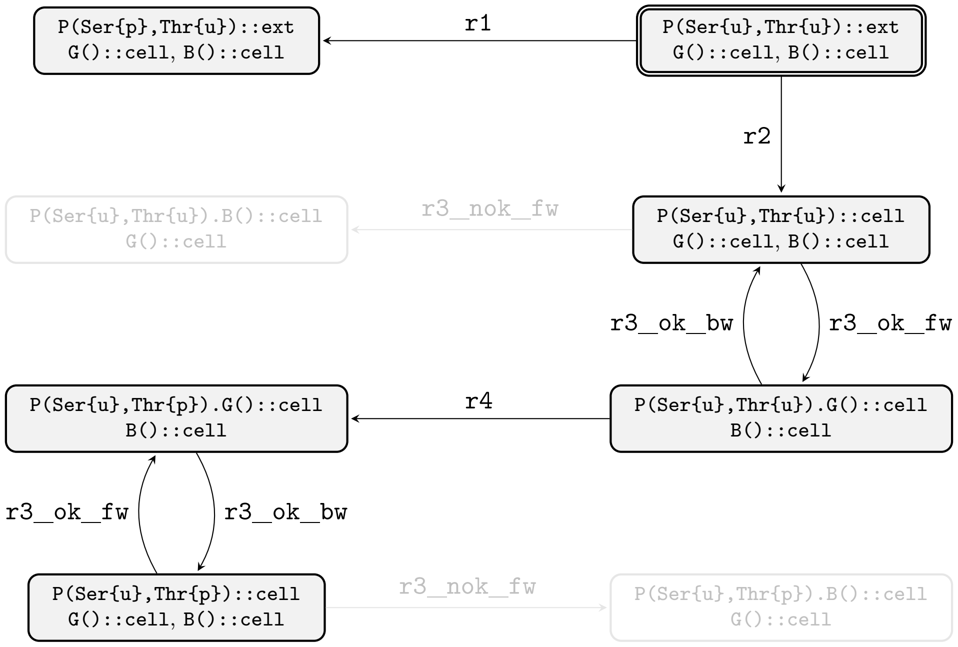
C. `conditional`

   `r3_nok_fw` : { `P(Ser{u},Thr{u})::cell` }

D. `concurrent-free`

   {(`r3_ok_fw`, `r3_nok_fw`)} (informally `r3_ok_fw` > `r3_nok_fw`)

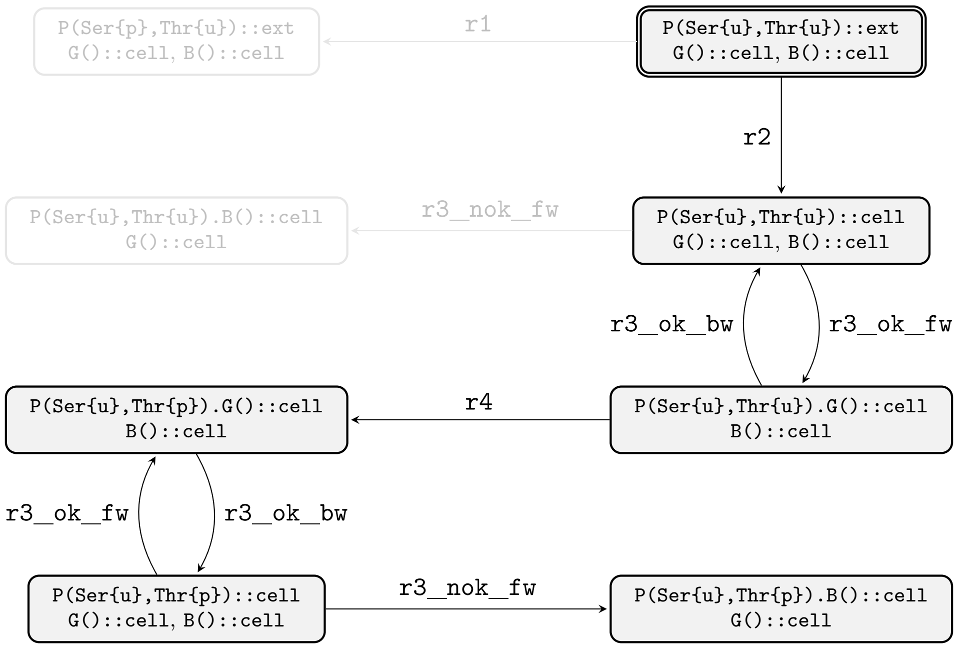
There might be more than one correct answer.
Can you find any other solutions?



P(Ser{p},Thr{u})::ext G()::cell, B()::cell
r1
P(Ser{u},Thr{u})::ext G()::cell, B()::cell
r2
P(Ser{u},Thr{u}).B()::cell G()::cell
r3_nok_fw
P(Ser{u},Thr{u})::cell G()::cell, B()::cell
r3_ok_bw
r3_ok_fw
P(Ser{u},Thr{p}).G()::cell B()::cell
r4
P(Ser{u},Thr{u}).G()::cell B()::cell
r3_ok_fw
r3_ok_bw
P(Ser{u},Thr{p})::cell G()::cell, B()::cell
r3_nok_fw
P(Ser{u},Thr{p}).B()::cell G()::cell

4.3.3 Task 3

Find a regulation which ensures that the state with a complex of threonine (`Thr`) phosphorylated molecule `P` and molecule `B` (i.e. `P(Ser{u},Thr{p}).B()::cell`) is always reachable (i.e. achieve the behaviour displayed below).



P(Ser{p},Thr{u})::ext G()::cell, B()::cell
r1
P(Ser{u},Thr{u})::ext G()::cell, B()::cell
r2
P(Ser{u},Thr{u}).B()::cell G()::cell
r3_nok_fw
P(Ser{u},Thr{u})::cell G()::cell, B()::cell
r3_ok_bw
r3_ok_fw
P(Ser{u},Thr{p}).G()::cell B()::cell
r4
P(Ser{u},Thr{u}).G()::cell B()::cell
r3_ok_fw
r3_ok_bw
P(Ser{u},Thr{p})::cell G()::cell, B()::cell
r3_nok_fw
P(Ser{u},Thr{p}).B()::cell G()::cell

4.3.4 Correct answers

Task 1 - A, C

Task 2 - A, B, D

Task 3

- `regular` – `r2 . (r3_ok_fw . r3_ok_bw)*. r4 . (r3_ok_fw . r3_ok_bw)* . r3_nok_fw`
- `conditional` – `r3_nok_fw` : { `P(Ser{u},Thr{u})::cell` }, `r1` : { `P(Ser{u},Thr{u})::ext` }

Chapter 5

BCSL tutorial

5.1 Operators

Overview of operators:

- {} – atomic agent, feature placeholder
- () – structural agent, composition placeholder
- :: – compartment, specification of the place of occurrence
- . – a complex, structure made of the structure and atomic agents using . notation
- $+$ – operator for adding two or more complexes, atomic agents, structural agents together
- $\Rightarrow$ – operator for describing reaction direction
- $\Leftrightarrow$ – operator for reversible reaction direction
- `@` – rate function placeholder

Overview of model sections:

- `#! definitions` – part of the script for aliases for numeric rate values (not mandatory)
- `#! rules` – part of the script for rules (mandatory)
- `#! inits` – part of the script for structure and atomic agents that must be present before the reaction can start (not mandatory)
- `#! complexes` - part of the script where aliases for complexes are defined (not mandatory)
- `#! regulations` – and extension allowing to regulate model's behavior (not mandatory)

5.2 Example model

To better understand rule-based modelling and, specifically, BCSL, we will start by introducing a real-life example. Assume that we have three people, Anabelle (`A`), Benito (`B`), and Cecilia (`C`), and we would like to model their typical day. Their day has some rules, so we will state them first.

Anabelle hates her job, and she is always sad while being there. Each of her days starts at work. She has a colleague, Benito, who works in different departments, but they share an attitude towards their job. The only bright moments are during the lunch break. They meet at the elevator. They either go together or not because they consider all other people hard to talk to and overly enthusiastic.

Let us write that in shorter notation, using BCSL:

`A(mood{sad})::work + B(mood{sad})::work ⇒ A(mood{happy}).B(mood{happy})::lunch`

Then one of two things can happen:

1. Benito is an intern and has to make a good impression; therefore, he goes back to work after lunch, usually with Anabelle.

   `A(mood{happy}).B(mood{happy})::lunch ⇔ A(mood{sad})::work + B(mood{sad})::work`

   After work, Anabelle gets in the car and starts to love her life again.

   `A(mood{sad})::work ⇒ A(mood{happy})::car`

   Then she can finally go home.

   `A(mood{happy})::car ⇒ A(mood{happy})::home`

2. Anabelle goes for a walk after they are done with lunch.

   `A(mood{happy}).B(mood{happy})::lunch ⇒ B(mood{sad})::work + A(mood{happy})::walk`

   The walk makes her happy. If she gets near the park and meets Cecilia, they go for a walk in the park together.

   `A(mood{happy})::walk + C()::park ⇒ A(mood{happy}).C()::park`

   After they are done talking and walking, Anabelle can get into the car and go home, while Cecilia stays in the park.

   `A(mood{happy}).C()::park ⇒ A(mood{happy})::car + C()::park`
   `A(mood{happy})::car ⇒ A(mood{happy})::home`

It is very probable that one day Anabelle will finally have enough of it all and will be inspired by Cecilia's life and will stay in the park.

`A(mood{happy}).C::park ⇒ A(mood{happy})::park + C()::park`

Above we discussed rules of their normal day, but for the day to even begin, they have to be in some initial state, in BCSL model specified in `#!inits` section. Our initial state is:

```
1 A(mood{sad})::work
1 C()::park
1 B(mood{sad})::work
```

We can change the initial state and look at what will happen. Let us say that Annabelle can also begin her day in the park. Then we would have to add `A(mood{happy})::park` to the initial state and remove `A(mood{sad})::work`. By allowing Annabelle to begin her day in the park, other things about her day would change. For example, she would not have to go to work anymore or to lunch to be in a happy mood and going for a walk to the park. She would already be there.

What if we removed `C()::park` from `# !inits`? Then Annabelle would never meet Cecilia, so she would not be able to go to the park at all. She could go for a walk but never come back from the walk.

5.3 Complex aliases

Complex aliases help simplify the rules during modelling. Let us say that we want to model generalized photosynthesis reaction. There are three main requirements for photosynthesis to take place: carbon dioxide, water, and light. We will write first two in the initial state while the light is neglected:

```
#! inits
6 C().O().O()::chlp
6 H().H().O()::chlp
```

We suppose that most readers know the chemical structure of water and carbon dioxide. Therefore we can simplify the notation using complex aliases.

```
#! complexes
water =             H().O().O()
carbonDioxide = C().O().O()
H12 =               H().H().H().H().H().H().H().H().H().H().H().H()
C6 =                C().C().C().C().C().C()
O6 =                O().O().O().O().O().O()
```

The rules will then be written as follows:

```
6 carbonDioxide::chlp + 6 water::chlp ⇒ C6.H12.O6::chlp + 6 O().O()::chlp
```

while still preserving the information about chemical structure of water and carbon dioxide.

5.4 Rates

Not all reactions have the same probability of taking place. In these cases, we can use `@ rate` notation in BCSL to adjust the probability. In the case the rates are not specified, they are assigned default constant value `1`. Let us take a look at some examples:

```
#! rules
r1 ~ A()::a + B()::b ⇒ A().B()::a
r2 ~ A()::a + B()::b ⇒ A()::a + B()::a

#! inits
A()::a
B()::b
```

No specific rates were given in this case, so the rules have been assigned default rate `1`. There are two rules stating that `A` and `B` can either be joined or stay separated. The rules have the same left-hand side. The probability of the rules is then calculated as the rate of the used rule divided by the sum of all rates assigned to rules that can take place in the current state. In our case:

Used rule is `r1`, with implicit rate 1. In the current state `A()::a +B()::b`, both rules can take place. Therefore the probability of rules being executed will be $\frac{1}{1+1} = 0.5$.

```
#! rules
r1 ~ A()::a + B()::b ⇒ A().B()::a   @ 0.5
r2 ~ A()::a + B()::b ⇒ A()::a + B()::b  @ 0.8

#! inits
A()::a
B()::b
```

Here we have rates specified. The probability of the execution of the rule one will then be calculated as $\frac{0.5}{0.5+0.8} = 0.38$, i.e. the rate of the first rule divided by the sum of all rates assigned to rules that can take place at the current state.

```
#! rules
r1 ~ A()::a + B()::b => A().B()::a   @ 0.5
r2 ~ A().B()::a => A()::a + B()::b   @ 0.8

#! inits
A()::a
B()::b
```

Used rule is `r1`. There is only one rule that can be executed, therefore the rate does not matter and the probability will be `1`.

```
#! rules
r1 ~ A().B()::a => A()::a + B()::b @ 0.7×[A()::a]
r2 ~ A()::a + B()::b => A().B()::a @ 0.8

#! inits
5 A()::a
5 B()::b
```

In this case, the first rule has a rate that is not constant, so the calculation will be more difficult. In the initial state, we have `5 A()::a + 5 B()::b`. Therefore, in the first step, we can use only the second rule. The probability will be calculated as $\frac{0.8}{0.8} = 1$, i.e. the rate of the second rule divided by sum of all rates assigned to rules that can take place at the current state.

After we used the second rule we are at the state `4 A()::a + 4 B()::b + 1 A().B()::a`. In this state, we can use both rules. Probability is calculated as follows:

1. rule `r1`: $\frac{0.7*4}{0.7*4+0.8} = 0.77$, where `4` is the number of `A`s

   the output state: `5 A()::a + 5 B()::b`

2. rule `r2`: $\frac{0.8}{0.7*4+0.8} = 0.23$

   the output state: `3 A()::a + 3 B()::b + 2 A().B()::a`

With the probability of `0.77` will we end up again in the initial state. The same thing will happen as at the beginning. With the probability of `0.23` we will end up in the state from case 2.

Starting in the state from case 2, we can use both rules. Probability is calculated as follows:

1. rule `r1`: $\frac{0.7*3}{0.7*3+0.8} = 0.72$

   the output state: `4 A()::a + 4 B()::b + 1 A().B()::a`

2. rule `r2`: $\frac{0.8}{0.7*3+0.8} = 0.28$

   the output state: `2 A()::a + 2 B()::b + 3 A().B()::a`

We can observe that the rate at which the `A().B()::a` dissociates to `A()::a` and `B()::b` depends on the amount of `A()::a` present, because the rate of the first rule is *not* constant.

5.5 Definitions

It is a similar concept to complex aliases. It also introduces aliases, but in this case, for numerical values. It is helpful in case we want the model to be more self-explanatory. Rather than writing down a seemingly random number, we can use an alias to assign descriptive meaning to the constants.

```
A()::a + B()::b => A().B()::a @ Pathway_1
A().B()::b => A()::a + B()::a @ Pathway_2

#! inits
A()::a
B()::b

#! definitions
Pathway_1 = 0.5
Pathway_2 = 0.8
```

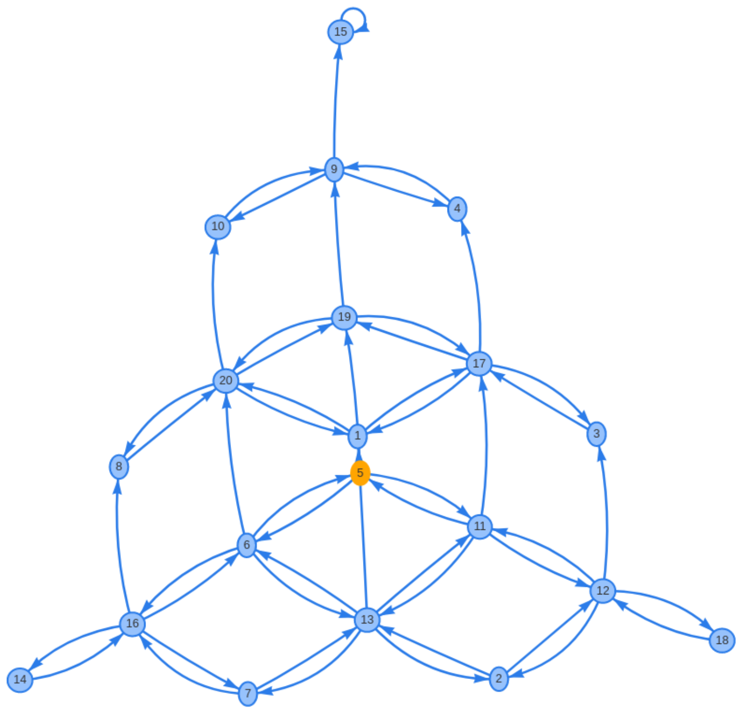
Let us say we want to model the above reverse chemical reactions. We know that `A` joins `B` in pathway 1, and `A` and `B` complex dissociates in pathway 2. Rather than writing down numbers, we can create aliases that specify pathway numbers.

5.6 Exercise

Create a model with 3 structural agents `A`, `B`, and `C`.

- In the begging, we have `3 A`s, `3 B`s, and `5 C`s.
- `A` is in compartment `a`, `B` in `b`, and `C` in `c`.
- While not in complex, `A`, `B`, and `C` have negative states that change to positive when they form complexes (use an atomic agent)
- When `A`, `B`, and `C` stand alone, they can create a complex that stays in the compartment `a`. Name the complex `D`.
- When `A` and `B` stand alone, they can join and create complex, same is true for `A` with `C`. The complexes will stay in the compartment `a`.
- Make the rate of `A().B()` complex formation 0.5 times the amount of `A`.
- Assign name `k1` to the constant 0.5.
- `A().B()` complex can dissociate back to `A` and `B` at a rate 0.5.
- `A().C()` complex can also dissociate to `A` and `C`.

Calculate the probability of `A` and `B` forming a complex starting from the initial state. Validate the result by inspecting the transition system of the model. You should see similar shape of the transition system.



5.6.1 Correct answers

```
#! rules
A(state{-})::a + B(state{-})::b ⇒ A(state{+}).B(state{+})::a @ [A()::a]×k1
A(state{+}).B(state{+})::a ⇒ A(state{-})::a + B(state{-})::b @ k1
A(state{-})::a + C(state{-})::c ⇔ A(state{+}).C(state{+})::a
A(state{-})::a + B(state{-})::b + C(state{-})::c ⇒ D()::a

#! inits
3 A(state{-})::a
3 B(state{-})::b
5 C(state{-})::c


#! definitions
k1 = 0.5

#! complexes
D = A(state{+}).B(state{+}).C(state{+})
```

The probability of `A` and `B` becoming a complex is `0.43`, calculated as follows: $\frac{3*0.5}{3*0.5+1+1} = 0.5$, three rules are applicable on the initial states, first second, and third; those that do not have explicit rates are implicitly 1.