Getting Started With SIBILLA and Population Models

Michele Loreti

April 7, 2022

Contents

1 Introduction

SIBILLA is a Java framework designed to support analysis of Collective Adaptive Systems. These are systems composed by a large set of interactive agents that cooperate and compete to reach local and global goals. The framework is structured in three main packages:

- *Models*, describing processes together with some utility classes for their analysis;
- *Simulation*, supporting system simulation;
- *Tools*, providing a set of tools that can be used to analyse the data collected from SIBILLA simulator;
- *Runtime*, using classes that permit access to all the features provided by our tool.

SIBILLA is not based on a specific formalism. Indeed, thanks to the use of recurrent patterns, all the packages can be easily extended with new features.

1.1 How to install

SIBILLA is available at the GitHub repository accessible via the following link:

https://github.com/quasylab/sibilla

The simplest way to download SIBILLA is to clone the repository locally by executing in a shell

```
git clone https://github.com/quasylab/sibilla.git
```

After that a directory named `sibilla`, that contains all the source files, is created. You can run the available gradle script to build the tool (replace `./gradlew` with `.\gradlew.bat` in Windows):

```
cd sibilla
./gradlew build
./gradlew installDist
```

The compiled tool is then available in the folder `shell/build/install/sshell`.

2 Population Models

In this section we will show how population models can be defined in SIBILLA.

2.1 Formal definition

We briefly recall here that a Population Continuous Time Markov Chain (PCTMC) model is a tuple $M = (\mathbf{X}, \mathcal{D}, \mathcal{T}, \mathbf{d}_0)$ where:

- $\mathbf{X} = (X_1, \ldots, X_n)$ is a vector of variables;
- each X_i takes values in a *finite* or *countable* domain $\mathcal{D}_i \subset \mathbb{R}$;
- $\mathcal{D} = \mathcal{D}_0 \times \cdots \times \mathcal{D}_n = \Pi_i \mathcal{D}_i$;
- $\mathbf{d}_0 \in \mathcal{D}$ is the *initial state* of the model;
- $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$ is the set of *transitions* $\tau_i = (\ell, \mathbf{s}, \mathbf{t}, r)$ where:
  - ℓ is the *label* of the transition;
  - $\mathbf{s} \in \mathbb{R}^n_{\geq 0}$, is the *pre-vector*;
  - $\mathbf{t} \in \mathbb{R}^n_{\geq 0}$, is the *post-vector*;

– $r : \mathcal{D} \to \mathbb{R}_{\geq 0}$ is a *rate function* such that for any $\mathbf{d} \in \mathcal{D}$, if $\mathbf{d} - \mathbf{s} + \mathbf{t} \notin \mathcal{D}$ then $r(\mathbf{d}) = \mathbf{0}$.

Let $M = (\mathbf{X}, \mathcal{D}, \mathcal{T}, \mathbf{d}_0)$, $\mathbf{d}_1, \mathbf{d}_2 \in \mathcal{D}$ and $\tau_i = (\ell, \mathbf{s}, \mathbf{t}, r) \in \mathcal{D}$. We let $\to_{\tau_i} \subseteq \mathcal{D} \times \mathbb{R}_{>0} \times \mathcal{D}$ denote the transition relation induced by transition τ_i:

$$\frac{r(\mathbf{d}_1) = \lambda \neq 0 \quad \mathbf{d}_2 = \mathbf{d}_1 - \mathbf{s} + \mathbf{t}}{\mathbf{d}_1 \xrightarrow{\lambda}_{\tau_i} \mathbf{d}_2}$$

We say that τ_i is *enabled* in $\mathbf{d}_1$ if and only if $r(\mathbf{d}_1) > 0$. Finally, function $\rho_{\tau_i} : \mathcal{D} \times \mathcal{D} \to \mathbb{R}_{\geq 0}$ is used to denote the rate of a transition τ_i from $\mathbf{d}_1$ to $\mathbf{d}_2$:

$$\rho_{\tau_i}(\mathbf{d}_1, \mathbf{d}_2) = \begin{cases} r(\mathbf{d}_1) & \mathbf{d}_2 = \mathbf{d}_1 - \mathbf{s} + \mathbf{t} \\ 0 & \text{otherwise} \end{cases}$$

Let $M = (\mathbf{X}, \mathcal{D}, \mathcal{T}, \mathbf{d}_0)$, $\mathbf{d}_1, \mathbf{d}_2 \in \mathcal{D}$ and $\tau_i = (\ell, \mathbf{s}, \mathbf{t}, r) \in \mathcal{D}$.

Function $\rho_{\mathcal{T}} : \mathcal{D} \times \mathcal{D} \to \mathbb{R}_{\geq 0}$ is used to denote the rate of a transition in M from $\mathbf{d}_1$ to $\mathbf{d}_2$:

$$\rho_{\mathcal{T}}(\mathbf{d}_1, \mathbf{d}_2) = \sum_{\tau_i \in \mathcal{T}} \rho_{\tau_i}(\mathbf{d}_1, \mathbf{d}_2)$$

We let $\to_{\mathcal{T}} \subseteq \mathcal{D} \times \mathbb{R}_{>0} \times \mathcal{D}$ denote the transition relation induced by transitions $\mathcal{T}$:

$$\frac{\rho_{\mathcal{T}}(\mathbf{d}_1, \mathbf{d}_2) = \lambda \neq 0}{\mathbf{d}_1 \xrightarrow{\lambda}_{\mathcal{T}} \mathbf{d}_2}$$

2.2 Population Models in Sibilla

Sibilla provides a module that simplifies the specification of population models. In this section we will use a simple scenario, based on a classical epidemic model, to introduce the basic features of our specification language.

The *SIR model* is one of the simplest compartmental models. The goal of the model is to represent the spread of an infection disease. The model consists of three species (or compartments):

S The number of susceptible individuals. When a susceptible and an infectious individual come into "infectious contact", the susceptible individual contracts the disease and transitions to the infectious compartment.

I The number of infectious individuals. These are individuals who have been infected and are capable of infecting susceptible individuals.

R The number of removed (and immune) or deceased individuals. These are individuals who have been infected and have either recovered from the disease and entered the removed compartment, or died. It is assumed that the number of deaths is negligible with respect to the total population. This compartment may also be called "recovered" or "resistant".

The SIR model can be described in Sibilla as follows:

```
param lambdaMeet = 1.0;      /* Meeting rate */
param probInfection = 0.25;  /* Probability of Infection */
param recoverRate = 0.05;    /* Recovering rate */

const startS = 90;           /* Initial number of S agents */
```

```
const startI = 10;           /* Initial number of I agents */

species S;
species I;
species R;

rule infection {
    S|I -[ #S*%I*lambdaMeet*probInfection ]-> I|I
}

rule recovered {
    I -[ #I*recoverRate ]-> R
}

system init = S<startS>|I<startI>;
```

Model parameters. A specification can contain a set of *parameters*. This are *real values* that can be changed after the model is loaded. In the SIR model above, we have three parameters:

```
param lambdaMeet = 1.0;      /* Meeting rate */
param probInfection = 0.25;  /* Probability of Infection */
param recoverRate = 0.05;    /* Recovering rate */
```

`lambdaMeet` is the *meeting rate* while `probInfection` is the probability that an agent `S` get infected when it meets an agent `I`. Finally, `recoverRate` is the recovering rate.

Constants. Constants are used to associate specific values to names. In our example, for instance, we use `startS` and `startI` to indicate the initial number of agents susceptibles and infected (90 and 10, respectively).

Rules. A set of rules are used to describe system dynamics. The simper form of the rule is the following:

```
rule <rulename> {
  <species>(|<species>)* -[ <exp> ]->  <species>(|<species>)*
}
```

where `<rulename>` is the name of the rule, `<species>` is a species name and `<exp>` is the expression used to compute the rule rates. Expressions are build by using standard mathematical operators and the two special operators `%X` and `#X`: the former returns the *fraction of agents of species* `X` while the latter amounts to the *number of agents of species* `X`. In our SIR scenario, such operators are used in the rule `infection`:

```
rule infection {
    S|I -[ #S*%I*lambdaMeet*probInfection ]-> I|I
}
```

Systems. The initial configurations of a system are described in the form

```
system <name> = <species>(|<species>)*;
```

When multiple copies of the same species `X` are in the system, the form `X<n>` can be used, where n is a numerical value.

2.3 Species patters and parametric rules.

Sometime it is useful to associate a species with a set of *parameters*. This parameters can be used to represent the location of an agent of a part of its state (in numerical form).

Let us consider a system where two groups of agents, named `A` and `B`, are distributed over a *grid* having dimensions $N \times M$ for some N and M. Both the agents randomly move from one place to another. However, while agents of type A tend to move to locations that are *empty*, those of type B follow the *crowd*.

The following portion of code can be used to declare two species, namely `A` and `B`, each of which is has two parameters identifying the position of the species in the grid:

```
param N = 10;
param M = 10;

species A of [0,N]*[0,M];
species B of [0,N]*[0,M];
```

An element of kind `A` at position (i, j) will be denoted by `A[i,j]`.

Following a similar approach, a family of rules can be defined by by using rule templates:

```
param movementRate = 1.0;

rule go_up_a for i in [0,N] and j in [0,M] when (i<N-1) {
    A[i,j] -[ movementRate*(1 - (%A[i+1]+%B[i+1])) ]-> A[i+1,j]
}
```

Above we have defined a family of rules where we let the variables `i` and `j` range in the intervals `[0,N]` and `[0,M]`, respectively. Moreover, the boolean condition `(i<N-1)` is used to *filter* some of the values (in this case, we can *go up* when we are not in the last *row* of the grid).

The other rules can be defined in the following way:

```
rule go_up_b for i in [0,N] and j in [0,M] when (i<N-1) {
    B[i,j] -[ movementRate*(1 - (%A[i+1]+%B[i+1])) ]-> B[i+1,j]
}

rule go_down_a for i in [0,N] and j in [0,M] when (0<i) {
    A[i,j] -[ movementRate*(1 - (%A[i+1]+%B[i+1])) ]-> A[i+1,j]
}

rule go_down_b for i in [0,N] and j in [0,M] when (i<N-1) {
    B[i,j] -[ movementRate*(1 - (%A[i+1]+%B[i+1])) ]-> B[i+1,j]
}
```

3 Sibilla Shell

Sibilla Shell is a *command line interpreter* that can be used to interact with the Sibilla core modules and permits performing all the available analysis. This front end can be either used interactively or in a *batch mode* to execute saved scripts.

3.1 Start Sibilla Shell

To run the shell one needs to execute the script `sshell` (or `sshell.bat`) in the folder `shell/build/install/sshell/bin`.

3.2 Use Sibilla Shell

Below a session that can be used to analyse the SIR model described in the previous section:

```
> module "population"
> load "seir_path/seir.pm"
> init "initial"
> add all measures
> deadline 100
> dt 1.5
> replica 100
> simulate
> save output "./results" prefix "sir" postfix "__"
```

A detailed list of SIBILLA Shell commands is reported below:

Table 1: Table with all the sshell's commands

Command name	**Description**
`modules`	This command shows the list of available modules.
`module <name>`	This command loads the module with the given name.
`load <filename>`	This command loads the specification from the given file name. This command fails if no module has been yet selected.
`env` (or `environment`)	This command shows the current assignment of parameters.
`set "<name>" <real>`	This command sets the given parameter to the given value.
`clear`	This command resets the shell content.
`reset "<name>"`	This command resets the given parameter to the default value.
`states`	This command shows the list of initial configurations.
`init "<name>"`	This command sets initial state to the given value.
`replica <num>`	This command sets the number of simulation replicas.
`deadline <real>`	This command sets the simulation deadline.
`dt <real>`	This command sets the sampling time.
`measures`	This command shows the list of available measure.
`add measure "<name>"`	This command adds the given measure to the ones collected during the simulation.

Continued on next page

Table 1 – *Continued from previous page*

Command name	Description
`add all measures`	This command adds all the available measures to the simulation.
`remove measure "<name>"`	This command removes the given measure from the ones collected during the simulation.
`remove all measures`	This command removes all measures from the simulation.
`save output "<output_folder>" prefix "<prefix>" postfix =<"postfix">`	This command saves the collected results in the given folder. Each measure will be saved in a `csv` file named `<prefix><measurename><postfix>.csv`. The generated file will contain three columns with: time of sampled value, mean, variance and standard deviation.
`run "<script_file>"`	This command executes the script in the given file name.
`quit`	This command terminates the execution.

4 Sibilla Python Front End

Sibilla provides a *Python front end* that permits interacting with Sibilla back end from Python programs. The use of this front end permits using many of the available Python libraries likes, for instance, Matplotlib that simplifies data visualisation. Moreover, Sibilla can be used within a web-based IDE such as Jupyter notebook/lab and in Google Colaboratory.

4.1 Start Sibilla Python

The Python front end can be easily used via Google Colaboratory or via Jupyter Notebook. To use Google Colaboratory you don't need to install anything, you just need any browser. You just need to log in to your google account and then create a new notebook. In the notebook enter these commands in a new cell and run them to install everything is needed for Sibilla.

```
    !git clone https://github.com/quasylab/sibilla
    !cd sibilla && ./gradlew build -x test && ./gradlew installDist
    !cp -a sibilla/shell/src/dist/scripts/sibilla_py .
!cd sibilla_py && pip install .
```

In a second cell enter the next lines of code and run them too.

```
    import os
    os.environ["SSHELL_PATH"]="/content/sibilla/shell/build/install/sshell/"
    import sibilla
```

and that's it, you're ready to use Sibilla, you can try yourself at the following link:

Google Colaboratory example

Another simple way to use the Python front end is to install Docker on your machine and use the provided Docker image.

Docker will provide a container image that includes everything needed to run Sibilla in Jupyterlab. (See Section 5.1)

Eventually you can install the Sibilla Python package in your own machine. To do that follow that, after the framework installation, the user should go via CLI into the folder `shell/build/install/sshell/scripts/sibilla_py` and type:

```
pip install .
```

It is important to set the environment variable **SSHELL_PATH** as follows

```
export SSHELL_PATH=/path/repository/sibilla/shell/build/install/sshell
```

for *nix Systems,

```
setx SSHELL_PATH "\path\repository\sibilla\shell\build\install\sshell" /M
```

for Windows Systems.

Below is a simple example of a python script where the seir model is simulated.

```
import sibilla

sibilla_runtime = sibilla.SibillaRuntime()
sibilla_runtime.load_module("population")
sibilla_runtime.load_from_file("seir.pm")
sibilla_runtime.set_configuration("initial_2")
sibilla_runtime.add_all_measures()
sibilla_runtime.set_deadline(20)
sibilla_runtime.set_dt(0.5)
sibilla_runtime.set_replica(100)
simulation_result = sibilla_runtime.simulate("test")
```

4.2 Use Sibilla Python

In the following a detailed list of methods that are provided:

Table 2: Table with all the Python's methods

Method name	**Description**
`get_modules()`	Return an array with the names of all enabled modules.
`load_module()`	Load the module with the given name.
`load_from_file(file_name: str)`	Load a specification from file.
`load(code: str)`	Load a specification from a string.

`info()`	Return info about the module state.
`load(code: str)`	Load a specification from a string.

`set_parameter(`
`  name: str,`
`  value: float,`
`  keep_configuration :`
`  bool = False,`
`  silent : bool = True)`
	Set a parameter to a given value. You can also keep the previous configuration by setting keep configuration keep_configuration True. If silent is False, it will print the operation, showing the previous parameter and the one just set.

`get_parameters()`	Get the array of current model parameter.
`get_evaluation_environment()`	Return the EvaluationEnvironment of the current model.
`clear()`	Cancels all the data loaded in the module.
`reset(name: str = None)`	Reset all the parameters to their default value. If you specify a name the method will reset to the default value only the specified parameter.
`get_initial_configurations()`	Return the array with the names of the initial configurations available in the current module.
`get_configuration_info(` `  name: str)`	Return a string providing info about the given configuration.

`set_configuration(`
`  name: str,`
`  *args: float)`
	Set initial configuration for simulations and property checking.

`get_measures()`	Return the array of measures defined in the module.
`is_enabled_measure(name: str)`	Return true if a specific measure is enabled
`set_measures(*measures: str)`	Set the measures to sample in a simulation.
`add_measures(*args)`	Add different measures to the ones collected in simulation.
`add_measures(*args)`	Add different measures to the ones collected in simulation.
`add_measure(name: str)`	Add a measure to the ones collected in simulation.
`remove_measure(name: str)`	Remove a measure from the ones collected in simulation.

`add_all_measures()`	Add all the available measures to the ones collected in simulation.
`remove_all_measures()`	Remove all measures from the ones collected from simulation.

`simulate(`
`label: str,`
`monitor:`
`SimulationMonitor = None)`
	Run a simulation and save results with the given label.

`use_descriptive_statistics()`	Use descriptive statistics.
`use_summary_statistics()`	Use summary statistics.
`is_descriptive_statistics()`	Return true if a descriptive statistics is used.
`is_summary_statistics()`	Return true if a summary statistics is used.
`get_statistics()`	Return a string that describes the kind of used statistics.
`check_dt()`	return true if the dt has been set
`set_deadline( deadline: float)`	set the deadline
`get_deadline()`	return the value of the deadline
`set_dt(dt: float)`	set the dt (samples per unit)
`get_dt()`	Return the value of the dt
`get_modes()`	Return the module modes.
`set_mode(name: str)`	Set module mode.
`get_mode()`	Return the current module mode.
`set_seed(seed: int)`	Set a seed for the random generator.
`get_seed()`	Generate and set a new seed that can be used to replicate experiments.

`save(`
`output_folder: str,`
`prefix: str,`
`postfix: str,`
`label: str = None)`
	Save last result to a given folder. Data files, in CSV format, are stored in the given output format. The name of each save file, having extension .csv, consists of the string prefix, the name of the series, and the postfix.

`set_replica(replica: int)`	Set the number of replications.
`set_replica()`	Set the number of simulation replica
`get_replica()`	Return the number of simulation replica
`print_data(label: str)`	return a string with the statistics collected by a labeled stimulation
`get_predicates(self)`	Returns an array of all the predicates of the model
`evaluate_reachability( goal: str, delta:float = 0.01, epsilon:float = 0.01, condition: str = None, monitor: SimulationMonitor=None)`	Evaluate the probability to reach a certain state

The method `simulate()` return an instance of the class `SibillaSimulationResult`, this class simplifies the management of simulation results, in particular it allows plot graphs by invoking the methods `plot(show_sd : bool = False)` and `plot_detailed()`. The `plot(show_sd : bool = False)` will show all the measures in the same graph, if `show\_sd` is set to True the standard deviation will be shown as a colored area around the single measurements. It is also possible to choose which measurement to display in the graph and zoom in on a specific range using the Range Slider

The `plot_detailed()` will show all the measures, one for each graph, with the respective standard deviations and the confidence interval.s

5 Sibilla Docker

Sibilla can be built by using Gradle. However, this needs some familiarity with the Java ecosystem. To simplify the deployment of Sibilla, a docker image is provided for creating containers that already have all the needed dependencies, and that can be easily executed in the Docker framework.

5.1 Start Sibilla Docker

To run simulation within this environment, first of all the user should download Docker from the site[1]. Commands should be run in the folder `/shell/src/dist/scripts/sibilla_docker`. To install Sibilla Docker use:

```
docker-compose up
```

After process conclusion, type on your browser "localhost:8888" and a Jupyter tab should open.

If the designated port is not available (a printed message will appear on the console tab), one should use instead the command:

```
docker build -t sibilla_docker .
```

[1]Docker Site: https://www.docker.com

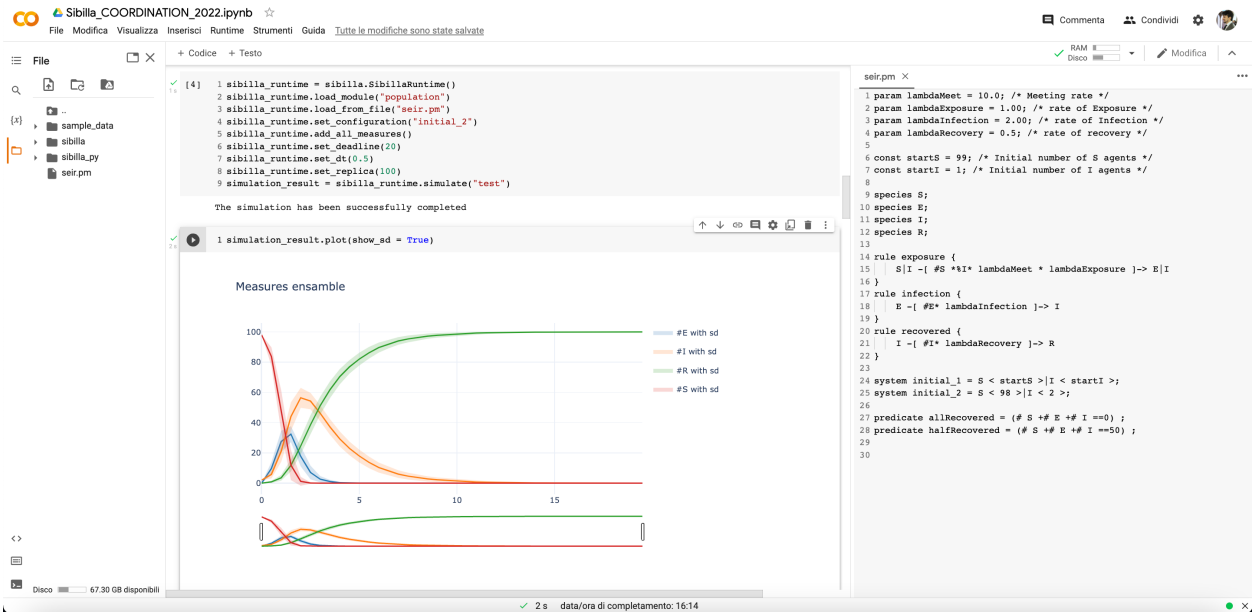
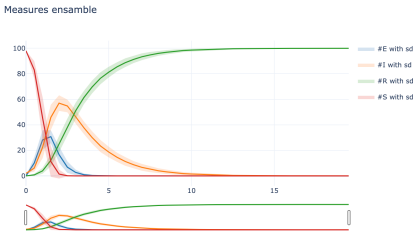
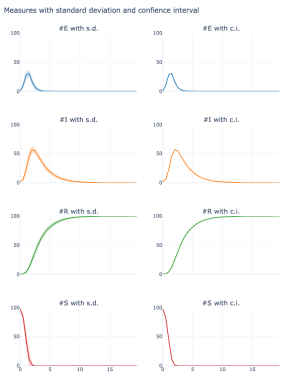


Figure 1: SIBILLA running at Google Colaboratory, the method



(a) plotted using the method `plot(show_sd : bool True)`



(b) plotted using the method `plot_detailed()`

Figure 2: Two way to plot the simulation results

At this point the image build should be successfully saved in the folder and a new container can be run via the next command:

```
docker run -p 8888:8888 -d sibilla_docker
```

As the previous approach, the Jupyter instance can be accessed via browser "localhost:8888".

5.2 Use Sibilla Docker

Because Docker uses JupyterLab as environment, the available commands are the same described in Section 4.2 of this document.