Sibilla: A Tool for Reasoning about Collective Systems

Nicola Del Giudice, Lorenzo Matteucci, Michela Quadrini, Aniqa Rehman, and Michele Loreti(✉)

University of Camerino, Camerino, Italy
{nicola.delgiudice,lorenzo.matteucci, michela.quadrini,aniqa.rehman, michele.loreti}@unicam.it

Abstract. Sibilla is a Java framework designed to support the 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. Sibilla is thought of container where different tools supporting specification and analysis of concurrent and distributed large scaled systems can be integrated. In this paper, a brief overview of Sibilla features is provided together with a simple example showing some of the tool's practical capabilities.

Keywords: Collective systems · Specification languages · Property specification and verification

1 Sibilla in a Nutshell

Sibilla[1] is a modular tool, developed in Java, to support quantitative analysis of Collective Adaptive Systems (CAS). This tool is thought of as a container where new components can be easily added to integrate new analysis techniques and specification languages. In this section, we first provide an overview of Sibilla back end. Then the specification languages that are currently available in our tool are presented. Finally, the user interfaces provided by Sibilla are described.

1.1 Sibilla Back End

Sibilla back end consists of four components: *Models*, *Simulation*, *Tools* and *Runtime*.

[1] The tool is available on GitHub at https://github.com/quasylab/sibilla and on *Software Heritage* with id swh:1:dir:fe015fb0a6fb6f5ee7cd6c58d446ab14168f39d4.

This research has been partially supported by Italian PRIN project "IT-MaTTerS" n, 2017FTXR7S, and by POR MARCHE FESR 2014–2020, project "MIRACLE", CUP B28I19000330007.

M. H. ter Beek and M. Sirjani (Eds.): COORDINATION 2022, LNCS 13271, pp. 92–98, 2022.
https://doi.org/10.1007/978-3-031-08143-9_6

Models. This component provides interfaces and classes that can be used to describe a *Stochastic Process* [14]. This is a collection of *random variables* $\{X(t)|t \in T \subseteq \mathbb{R}_{\geq 0}\}$. These random variables take values on a *measurable set* S representing the *state space* of the stochastic process. Random variable $X(t)$ describes the state of the process at time unit $t \in T$. Different types of stochastic processes can be considered depending on the index set T (*discrete time* vs *continuous time*) or the properties of the considered process (such as *Markovian Processes*). The classes provided in the *Models* permits describing these processes together with some utility classes for their analysis (such as *transient analysis* of *Markov Chains*). These classes provide the base on top of which different *specification languages* can be implemented (see Sect. 1.2).

Simulation. When one considers a system composed of a large number of entities, exact numerical analysis of stochastic processes is often hard or even impossible to be used. This is mainly due to the problem of *state space explosion*. For this reason, Sibilla provides a set of classes that permits supporting simulation of stochastic processes. First of all, these classes permit sampling a *path* from a model. This represents a possible computation/behaviour that can be experienced in the model. Classes to extract *measures* from a *path* and to collect statistical information are also provided. The classes for simulation samplings and statistical analyses rely on the *The Apache Commons Mathematics*[2].

Sometimes, even simulation can require a remarkable computational effort. For this reason, in Sibilla a framework has been integrated that permits supporting simulation to follow a *multi-threading* approach. The framework, based on *Java Concurrency API*, supports the execution of *simulation tasks* according to different *scheduling policies* that can be tailored to fit with the *parallelism* of the hosting architecture. Currently, a framework that permits dispatching simulation tasks over multiple hosts is under development.

Tools. This component provides a set of tools that can be used to analyse the data collected from Sibilla simulator. Currently the following tools are available: computation of *first-passage-time* and *reachability analysis*.

The *first-passage-time* is used to estimate the average amount of time needed by a model to reach a given *condition*. The latter consists of a *predicate* on the states of the considered process. *Reachability analysis* permits estimating the probability that a given set of states (identified by a *condition*) can be reached within a given amount of time by passing through states satisfying a given predicate. We will see in Sect. 1.2 how the specific syntax and format of used conditions depend on the used *specification language*.

The above mentioned tools strongly rely on *statistical inference techniques*. Note that, thanks to the Sibilla modularity, other tools can be easily integrated.

Runtime. Sibilla runtime provides the classes that permit access to all the features provided by our tool. The runtime in fact plays the role of the *controller* in

[2] https://commons.apache.org/proper/commons-math/.

the standard *Model-View-Controller* pattern [13] and it is used by the Sibilla user interfaces (see Sect. 1.3). Sibilla runtime is structured in *modules*. Each module is associated with a *specification language*. When a module is *selected*, the runtime environment will start working with the corresponding language.

1.2 Sibilla Specification languages

In this section, a brief overview of the specification languages currently included in Sibilla is provided. These languages permit describing stochastic processes with different features and using different primitives. As we have already remarked, one of the main features of Sibilla is that the tool is not focused on a specific language, but it can be extended to consider many formalism. This feature is useful whenever one is interested in the study languages that are equipped with constructs and primitives thought to model specific application domains (see for instance [2,11,12]).

Currently, in Sibilla three specification languages are fully integrated: *Language of Interactive Objects*, *Language of Population Model*, and *Simple Language for Agent Modeling.*

Language of Interactive Objects (LIO). This is a formalism introduced in [8] where a *system* consists of a population of N identical interacting objects. At any step of computation, each object can be in any of its finitely many states. Objects evolve following a *clock-synchronous* computation. Each member of the population must either execute one of the transitions that are enabled in its current state (by executing an *action*), or remain in such a state. This choice is performed according to a probability distribution that depends on the whole system state. The stochastic process associated with a LIO specification is a *Discrete Time Markov Chain* (DTMC) whose states consists of a vector of counting variables associating each state with the number of agents in this state. LIO models have been used to describe a number of case studies in different application domains [6–8].

Language of Population Models (LPM). A *Population Model* [4] consists of a set of agents belonging to given set of *species*. A system evolves by means of *reaction rules* describing how the number of elements of the different species changes. Rules are applied with a *rate*, which is a positive *real value*, that depends on the number of agents in the different species. The stochastic process associated with a LPM specification is a *Continuous Time Markov Chain.* Those models have been widely used to model different kinds of systems in different application domains ranging from ecology and epidemics to cyber-physical systems.

Simple Language for Agent Modelling (SLAM). This is an agent-oriented specification language. A system consists of a set of agents that interact in order to reach *local* and *global goals.* Each agent is equipped with a set of *attributes.* Some of these attributes are under the control of the agent and are updated during its evolution, while others depend on the environment where the agent operates, and their values are updated as the time is passing. The latter are used in particular to model the fact that an agent is able to *sense* its environment. In SLAM, agents interact via explicit message passing via *attribute based communication primitives* [1,9,10]: an agent sends messages to other agents satisfying a given predicate. The stochastic process associated with a SLAM specification is a *Timed Process* where actions and activities may have a duration sampled by generic distributions.

1.3 Sibilla front ends

To simplify the integration of Sibilla has several interpreters to allow smooth interaction between user and tool. For each specification language there is its own interpreter that allows simple creation of models so that the user can create his own model without necessarily knowing Java technicality. Then the user can easily perform analyses and interact with its own model in different ways, for example, by using the shell provided. The user can also use Sibilla by coding in Python, thus allowing the use of web-based interactive development environment like Google Colaboratory or Jupyter Notebook (see Subsect. 1.3). The current version of the tool allows to execute simulations in two ways: either via command line tool (denominated Sibilla shell), via Python scripts. Moreover, a Docker image is available to simplify deployment of the tool.

Sibilla *Shell.* This is a *command line interpreter* that can be used to interact with the Sibilla core modules and permits performing all the analysis outlined in the previous section. This front end can be either used interactively[3] or in a *batch mode* to execute saved scripts.

Python Front End. Python is one of the easiest to learn and versatile language, and it has been established as one of the most used languages, especially in the context of *data analysis*. For these reasons, Sibilla also provides a *Python front end* that permits interacting with Sibilla back end from Python programs. This front end relies on the *Pyjnius* library [15] that simplifies the interaction between Python and Java classes. The use of this front end permits using many of the available Python libraries likes, for instance, Matplotlib[4] that simplifies data visualisation. Moreover, Sibilla can be used within a web-based IDE such as Jupyter notebook/lab [5] and in Google Colaboratory [3]. This significantly increases the portability of the tool which can then be used without the need to install anything on the local machine.

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.

[3] The full list of shell commands is available at the Sibilla web site.
[4] https://matplotlib.org.

2 Sibilla at Work

In this section, we use a simple example to show how Sibilla can be used to support analysis of a simple scenario[5]. We consider a classical epidemic example based on the *Language of Population Model*: the *SEIR Model*. The goal of the model is to represent the spread of an infection disease. The model, whose specification is reported in Listing 1.1, consists of four species (or compartments): S, an individual that is susceptible; E, an individual that has been exposed to the virus but is not yet infected; I, an individual that is infected; R, an individual that is immune.

```
param lambdaMeet = 10.0; /* Meeting rate */
param probInfection = 1.00; /* Probability of Infection */
param incubationRate = 2.00; /* rate of Infection */
param lambdaRecovery = 0.5; /* rate of recovery */
param scale = 1.0;

const startS = 99; /* Initial number of S agents */
const startI = 1; /* Initial number of I agents */

species S;
species E;
species I;
species R;

rule exposure {
    S|I -[ #S*%I*lambdaMeet*probInfection ]-> E|I
}
rule infection {
    E -[ #E*lambdaInfection ]-> I
}
rule recovered {
    I -[ #I*lambdaRecovery ]-> R
}

system init = S<startS*scale>|I<startI*scale>;

predicate allRecovered = (#S+#E+#I==0) ;
```

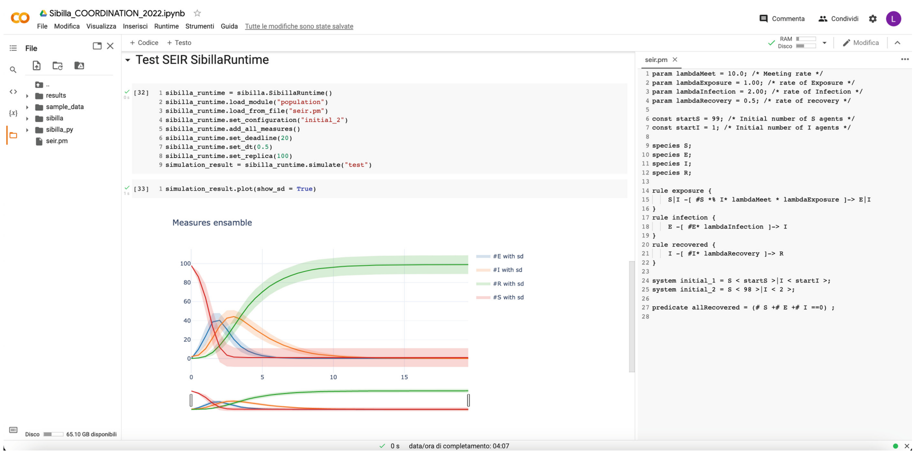
Listing 1.1. SEIR model in SIBILLA

We can observe that a specification can contain a set of *parameters*. These are *real values* that can be changed after the model is loaded. The parameters considered in our model are the agent meeting rate (lambdaMeet), the probability of infection (probInfection), the rate of incubation (incubationRate) and the recovery rate (recoverRate).

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>)*
}
```

[5] Detailed Sibilla documentation is available at https://github.com/quasylab/sibilla/wiki.


Sibilla_COORDINATION_2022.ipynb
File Modifica Visualizza Inserisci Runtime Strumenti Guida
Tutte le modifiche sono state salvate
Commenta
Condividi
+ Codice + Testo
RAM
Disco
Modifica
results
sample_data
sibilla
sibilla_py
seir.pm
Disco 65.10 GB disponibili
Measures ensamble
#E with sd
#I with sd
#R with sd
#S with sd
0 s data/ora di completamento: 04:07

Test SEIR SibillaRuntime

```
[32] 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")

[33] simulation_result.plot(show_sd = True)
```

seir.pm

```
param lambdaMeet = 10.0; /* Meeting rate */
param lambdaExposure = 1.00; /* rate of Exposure */
param lambdaInfection = 2.00; /* rate of Infection */
param lambdaRecovery = 0.5; /* rate of recovery */

const startS = 99; /* Initial number of S agents */
const startI = 1; /* Initial number of I agents */

species S;
species E;
species I;
species R;

rule exposure {
    S|I -[ #S *% I* lambdaMeet * lambdaExposure ]-> E|I
}
rule infection {
    E -[ #E* lambdaInfection ]-> I
}
rule recovered {
    I -[ #I* lambdaRecovery ]-> R
}

system initial_1 = S < startS >|I < startI >;
system initial_2 = S < 98 >|I < 2 >;

predicate allRecovered = (# S +# E +# I ==0) ;

```

Fig. 1. Sibilla running at Google colaboratory

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 SEIR scenario, such operators are used in the rule `exposure`:

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

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.

In Fig. 1 the analysis of the SEIR model with Sibilla is performed in Google Colaboratory. We can observe how results of the simulation are represented as a plot. Moreover, we can also compute the average time needed to eradicate the infection. This can be obtained in terms of the *first passage time* of the condition `#E+#I=0`, namely when then number agents that are either *exposed* or *infected* is 0. At the same time, reachability can be used to compute the probability that in a given time unit t 90% of the agents *recovered* while never happens that more than 10% of agents are infected.

3 Concluding Remarks

In this paper, a brief overview of the framework Sibilla has been provided. Sibilla is a Java framework designed to support the analysis of Collective Adaptive

Systems. It is thought of as a container where different tools supporting specification and analysis of concurrent and distributed large scaled systems can be integrated. A simple example has been used to show basic features of Sibilla.

As future work, we will plan to integrate in Sibilla a *Graphical User Interface*. The goal is to improve the usability of our tool for users that are not familiar with formal methods. Moreover, we plan to extend the analysis tools available in Sibilla in order to consider model checking tools in the spirit of [6]. Finally, we plan to integrate an optimization module that, by relying on machine learning-assisted inference tools, permits identifying the best parameters that maximize/minimize given goal functions.

References

1. Alrahman, Y.A., Nicola, R.D., Loreti, M.: Programming interactions in collective adaptive systems by relying on attribute-based communication. Sci. Comput. Program. **192**, 102428 (2020)
2. Bettini, L., et al.: The Klaim project: theory and practice. In: Priami, C. (ed.) GC 2003. LNCS, vol. 2874, pp. 88–150. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40042-4_4
3. Bisong, E.: Google Colaboratory, pp. 59–64. Apress, Berkeley, CA (2019)
4. Bortolussi, L., Hillston, J., Latella, D., Massink, M.: Continuous approximation of collective system behaviour: a tutorial. Perform. Eval. **70**(5), 317–349 (2013)
5. Kluyver, T., et al.: Jupyter notebooks - a publishing format for reproducible computational workflows. In: Loizides, F., Schmidt, B. (eds.), Positioning and Power in Academic Publishing: Players, Agents and Agendas, pp. 87–90. IOS Press (2016)
6. Latella, D., Loreti, M., Massink, M.: On-the-fly PCTL fast mean-field approximated model-checking for self-organising coordination. Sci. Comput. Program. **110**, 23–50 (2015)
7. Latella, D., Loreti, M., Massink, M.: FlyFast: a mean field model checker. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 303–309. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_18
8. Le Boudec, J.-Y., McDonald, D., Mundinger, J.: A generic mean field convergence result for systems of interacting objects. In: Fourth international conference on the quantitative evaluation of systems (QEST 2007), pp. 3–18. IEEE (2007)
9. Loreti, M., Hillston, J.: Modelling and analysis of collective adaptive systems with CARMA and its tools. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 83–119. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34096-8_4
10. Nicola, R.D., Duong, T., Loreti, M.: Provably correct implementation of the ABC calculus. Sci. Comput. Program. **202**, 102567 (2021)
11. Nicola, R.D., Latella, D., Loreti, M., Massink, M.: MarCaSPiS: a markovian extension of a calculus for services. Electron. Notes Theoret. Comput. Sci. **229**(4), 11–26 (2009)
12. Nicola, R.D., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming. ACM Trans. Autonom. Adapt. Syst. **9**(2), 1–29 (2014)
13. Reenskaug, T., Wold, P., Lehne, O.A., et al.: Working with objects: the OOram software engineering method, chapter 9.3.2, pp. 333–338. Citeseer (1996)
14. Ross, S.M., et al.: Stochastic processes, vol. 2. Wiley, New York (1996)
15. K. Team and other contributors. pyjnius. https://github.com/kivy/pyjnius