Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, P. Lendermann, eds.

py2PowerDEVS: CONSTRUCTION AND MANIPULATION OF LARGE COMPLEX STRUCTURES FOR PowerDEVS MODELS VIA PYTHON SCRIPTING

Ezequiel Pecker-Marcosig
Matías Bonaventura
Esteban Lanzarotti
Lucio Santi
Rodrigo Castro

Departamento de Computación
FCEyN, UBA and ICC, CONICET
Ciudad Universitaria, Pabellón 1
C1428EGA, Buenos Aires, ARGENTINA

ABSTRACT

As the disciplines of modeling and simulation evolve and become more efficient, the complexity of the scientific applications that can be tackled by simulation modeling continue to increase. The approach of building complex models through the composition and interconnection of modular units of behavior has been a key factor in this success. However, very often complexity entails a significant growth in the size and intricacy on the structure of simulation models. In this paper we confer new capabilities for building models with large complex structures to PowerDEVS, an established C++-based simulation toolkit for the DEVS formalism, that has typically based its modular modeling experience on a Graphical User Interface. We present py2PowerDEVS, a Python framework that seamlessly integrates pre-built modular PowerDEVS components into the powerful and growing ecosystem of Python scripting, enabling the algorithmic design of large complex PowerDEVS model structures. We demonstrate the use of py2PowerDEVS in three scientific application domains: data networks, geographical virus spread and mechanical systems.

1 INTRODUCTION

Modeling and simulation (M&S) are two different, yet fundamentally interrelated disciplines for the study of systems. Both are of vital importance in the design of efficient development cycles of products in M&S-based design techniques (Castro et al. 2019). Modeling concerns primarily with the extraction of knowledge from a real system of interest to build a model which is an abstract and limited representation of the reality (Cellier 1991). Models can be described resorting to a wide range of mathematical formalisms. The specific formalism to be used will depend on the type of questions to be answered.

Diverse software tools help modelers build their components by eliminating unnecessary overheads, allowing them to focus on the core activities of extracting knowledge and representing it into the mathematical formalism of choice (rather than on specific details and nuances of the software platforms).

In several contexts, the agility, flexibility and ease of use of the chosen modeling technique and software can become crucial. One such context is complex large-scale systems modeling, where the model development workflow requires special support from software tools to increase productivity and reduce errors. In this paper we confer new capabilities for large-scale model building to PowerDEVS (Bergero and Kofman 2011), a well established C++-based *simulation* toolkit for the Discrete EVent System specification (DEVS) formalism, that relies mainly on a GUI-based *modeling* experience.

Specifically, we present py2PowerDEVS, a Python framework that seamlessly integrates pre-built PowerDEVS modular components into the rich and growing Python scripting ecosystem, thus opening up the possibility of constructing large and/or complex PowerDEVS model structures in an algorithmic way. py2PowerDEVS is publicly available from our gitlab repository (SEDLab 2022).

DEVS is, by definition, a modular and hierarchical mathematical formalism for modeling and simulation of a variety of dynamical systems (Zeigler et al. 2018). It is able to represent any discrete-event system provided that it performs a finite number of transitions in any finite interval of time. An attractive feature of DEVS is that it has proven to be a common denominator for a variety of other mathematical formalisms (Vangheluwe 2000) which can then be mapped to DEVS, allowing to construct multiformalism models under a unified tool. Under DEVS, a model is strictly split in terms of behavior and structure. Thanks to the modular approach, DEVS fosters the reusability of pre-built components that can be orchestrated and rewired as required for each new project.

A number of DEVS simulation tools have been developed and are now available, covering a wide variety of programming languages either compiled or interpreted, pursuing different design goals and ranging from low-level interfaces (where a DEVS model is programmed using a strongly typed language) to high-level tools (relying on GUIs and the drag-and-drop paradigm). The decision as to which tool is best suited to an application depends strongly on the requirements.

PowerDEVS is mainly intended to design hybrid dynamical systems, where the interaction of discrete and continuous parts of a system is a relevant feature. A distinctive advantage is its performance compared to other DEVS simulators (Van Tendeloo and Vangheluwe 2017), a direct consequence of the underlying C++ backbone. This fact justifies preserving the definition of component behavior in its original language, while enriching the structure design capabilities by means of scripting techniques.

This paper is structured as follows: Section 2 presents the background and motivation, Section 3 is devoted to py2PowerDEVS and the details of its implementation. The use of the tool is shown in Section 4 for three case studies of increasing complexity involving different domains. The simulation results and their performance analysis are discussed in Section 5 and finally Section 6 presents the conclusions.

2 BACKGROUND AND MOTIVATION

A remarkable feature of PowerDEVS is its GUI, which hides away the internal aspects of DEVS models facilitating its use by modelers unfamiliar with DEVS. However, when working with large-scale systems the GUI can become impractical, for it requires to handle manually the creation and interconnection of an enormous number of models.

In PowerDEVS this issue was partially targeted in the past by supporting a DEVS extension called VectorialDEVS (Bergero and Kofman 2014) by which a single graphical model can represent multiple instances of a same atomic model, each of which accepts different parameters of operation. However, models involving non-regular structures, such as graphs with random adjacency matrices with varied number of edges per node (e.g. contact networks of spread processes in models of epidemics) are intractable with VectorialDEVS. Also, VectorialDEVS does not support vectors of coupled models.

Our approach with py2PowerDEVS is completely flexible allowing to seemingly interconnect both coupled and atomic DEVS models in a programmatic way with Python scripts. The advantage of scripting is the use of interpreted weakly-typed languages designed for gluing predefined components, reducing complexity, avoiding compilation-related issues and enabling the rapid development of simulation models. Python has boomed in recent years, leading popularity indices for programming languages (e.g. PYPL and Tiobe). Surprisingly, there is a small number of Python-based DEVS simulation tools. Simulation engines written in Python include xDEVS (Risco-Martín, J.L. 2014), PythonPDEVS (Van Tendeloo and Vangheluwe 2015), PyVLE (Quesnel, Gauthier and Ramat, Eric 2022) and DEVSSimPy (Capocchi et al. 2011). In the case of PyVLE it also generates C++ code for simulation. The main claim against interpreted simulators compared to compiled simulators is in terms of performance.

py2PowerDEVS allows PowerDEVS classes defined in C++ to be accessed directly from Python code, while execution always occurs in C++. The main goal is to boost the capabilities when constructing large topological models by using Python as an alternative (and a complement) to the standard PowerDEVS GUI.

2.1 The PowerDEVS Simulation Toolkit

PowerDEVS (Bergero and Kofman 2011) comprises four independent programs. The *Model Editor* is the graphical user interface (GUI) which provides a canvas to construct and configure the simulation model. The *Atomic Editor* is intended to describe the behavior of atomic models. The *Preprocessor* generates C++ code from the graphical model representation which is then compiled to produce a standalone executive that includes the model specification and the simulation engine. Finally, the *Simulation Interface* is a graphical simulation control interface that allows interactive simulation execution.

In the *Design View* a model is built by interconnecting graphical representations of DEVS atomic and DEVS coupled models (blocks with input/output ports that are dragged, dropped and wired) including the setting of their model parameters. The *Simulation View* to set the simulation parameters (e.g. final simulation time, number of executions, etc.) and check the evolution of the simulation time is provided by the *Simulation Interface* or through command line with the executive produced by the Preprocessor.

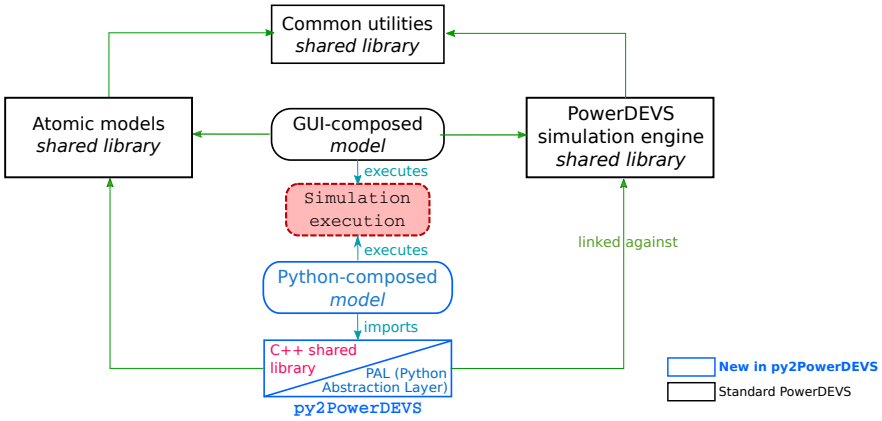


Figure 1: py2PowerDEVS architecture and components. Black border: Standard PowerDEVS. Blue border: New py2PowerDEVS scripting framework.

3 THE py2PowerDEVS INTERFACE FOR COMPOSING LARGE DEVS SYSTEMS

This section describes py2PowerDEVS (SEDLab 2022), an open source Python-to-PowerDEVS interface allowing the PowerDEVS C++ simulation engine to be directly accessed from the Python language.

3.1 The py2PowerDEVS High Level Architecture

We refactored PowerDEVS to generate separated shared libraries for common utilities, the simulation engine and atomic models. When a model is defined using the PowerDEVS GUI, C++ code is automatically generated with the model description and then is linked against the shared libraries to produce a compiled executable.

Figure 1 shows the architecture of py2PowerDEVS integrated into the PowerDEVS ecosystem. A new Python Abstraction Layer (PAL) allows the PowerDEVS simulation engine and the models to be accessed from Python. The Python-PowerDEVS binding is implemented using the Boost.Python library (BoostLib 2022) and also linked against the PowerDEVS shared libraries. At compilation time, the PAL automatically generates a Python-accessible class for every PowerDEVS atomic model defined in C++. Having atomic

models accessible from Python code allows for the definition of Python classes that mimic DEVS coupled models. Additionally, the PAL generates Python classes for all models found in the PowerDEVS library, making coupled models also available to Python code. By using the PAL, Python scripts can access the PowerDEVS simulation engine to execute models, set parameters, configure logging and select global options available to the C++ compiled models.

As it will be shown in Section 5.1, the execution of simulation models defined in py2PowerDEVS barely imposes performance penalties.

py2PowerDEVS is provided as a new Python module composed of five submodules, namely: `core`, `library`, `params`, `model`, and `plot`. In the following sections we will overview each submodule. The class diagrams corresponding to py2PowerDEVS can be found in (SEDLab 2022).

3.2 Making PowerDEVS C++ Models Available From Python: the `core` submodule

The module `py2powerdevs.core` provides low-level access to PowerDEVS functionality. Entirely written in C++, it exposes to Python a well defined interface for interacting with selected C++ functions and classes of the PowerDEVS engine.

The class *Py2PowerDEVSModel* is the main class in the `py2powerdevs.core` submodule that allows the interaction with the simulation engine and atomic model library. To create new atomic model instances, this class is populated with methods `create_<atomic_name>()` for every PowerDEVS atomic model defined in C++, made visible by the PAL. These methods receive the parameters needed by the corresponding atomic model. Furthermore, the *Py2PowerDEVSModel* class is augmented with methods to manipulate the DEVS model hierarchy and to interact with the simulation engine. This module is mainly used by other py2PowerDEVS modules, as will be seen shortly, but should not be used by a standard user.

3.3 Building PowerDEVS Models in Python: the `model` submodule

Unlike the previous module, `py2powerdevs.model` is entirely written in Python. The main class in this module is *RootModel*. This class is more general than *Py2PowerDEVSModel*, provided that it has a data member `interface` of the same type. In addition, the *RootModel* class includes methods for: interconnecting models (`connect()`, `connect_input()` and `connect_output()`), managing parameters, working with loggers (`log()`), and running the final model (`run()`). Moreover, the individual atomic and coupled models make up an ordered dictionary that can be accessed under the `submodels` data member (dictionary's key values correspond to the names of the coupled and atomic submodels).

The `py2powerdevs.model` submodule defines a function called `new_model()` which returns an object *RootModel*. In addition, this module defines several submodules with utilities such as generation of random numbers and read/write data collected during simulation.

3.4 Python Libraries of PowerDEVS Models: the `Library` submodule

The `py2powerdevs.library` module is a collection of Python DEVS models readily available and tested for modelers to reuse. Users can manually create new models using the infrastructure in `py2powerdevs.core` described earlier, while the PowerDEVS distribution comes with a set of extensible model libraries (that are constantly updated and extended) for use with the GUI. The py2PowerDEVS PAL features a code generator that automatically creates Python classes for every model in the PowerDEVS GUI library, placing them in the `py2powerdevs.library` module. The generated Python classes implement, within py2PowerDEVS, the same behavior and interconnections as those implemented in the library for GUI models. An example of generated py2PowerDEVS code based on a GUI model can be found in Fig. 2b.

As mentioned earlier, models defined in the GUI lack flexibility to define a configurable structure, and this is also the case for the generated Python models. Nevertheless, generated code is designed so that Python models can be extended using class inheritance and method overload. This becomes useful

when models in the PowerDEVS GUI Library need to be adapted to build more complex structures and interconnections. An example of extending a PAL generated structure is described in Section 4.3 for *network router* models that require a dynamic number of ports.

New py2PowerDEVS models can be added to the library either manually, using the PAL code generator, or a combination of both, exploiting the strengths of each approach. A typical model development workflow involves the following sequence of steps: 1) build a single model with the GUI, and add it to the PowerDEVS library 2) let the PAL code generator create the py2PowerDEVS counterpart, and 3) augment this single model extending its structure and/or creating multiple instances with Python algorithms. This workflow will be further explored through the examples in Section 4.

3.5 Post-processing, Plotting and Parameters Management: the `plot` and `param` submodules

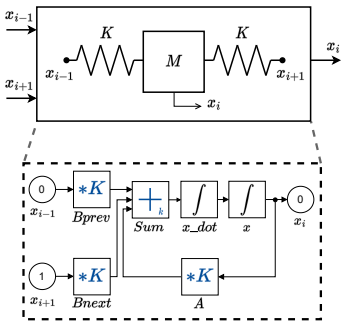
The `py2powerdevs.params` module manages parameters for models and simulations in the Python environment. It provides utilities to read parameters either from command line, multiple configuration files, and/or setting them directly with Python code. Before running a model, the PAL uses the `py2powerdevs.params` module to serialize parameters in the format required by the simulation engine. This allows for greater flexibility in models requiring a large amount of interdependent parameters. An example of this feature is described in Section 4.3.

The `py2powerdevs.plot` module provides post-simulation utilities to load, manipulate, transform, and plot simulation results. Time-series of different metrics for each model are collected during simulation and stored in standard HDF5 format (Folk et al. 2011). Also parameters and general simulation performance statistics (such as initialization and execution times) are stored for analysis. *Plotter* is the main class in the `py2powerdevs.plot` module, which provides methods to print simulation times, standard methods to manipulate time-series, plotting and retrieve used parameters.

4 CASE STUDIES

In this section we present three examples from different fields to showcase some advantages offered by py2PowerDEVS for modeling and simulation.

4.1 Multiple Interconnected Mass-Spring Models to Study Oscillatory Systems



(a) Schematic view (above) and PowerDEVS GUI model (below).

```
class MassSpringExt(BaseModel):
  [...]
  def _do_create_model(self):
    self._pd_model = self.interface.create_coupled(self.parent.pd_model(), self.name)
    # Child models
    __x = self.interface.create_qss_integrator(self._pd_model,"x",["QSS3", "1e-6", "1e-3", self.x0])
    __xdot = self.interface.create_qss_integrator(self._pd_model,"xdot",["QSS3", "1e-6", "1e-3", self.v0])
    __A = self.interface.create_qss_gain(self._pd_model,"2K/M",[self.A])
    __Bprev = self.interface.create_qss_gain(self._pd_model,"Bprev",[self.Bprev])
    __Bnext = self.interface.create_qss_gain(self._pd_model,"Bnext",[self.Bnext])
    __Sum = self.interface.create_qss_wsum(self._pd_model,"Sum",["1", "1", "-1", "0", "0",
                                                                 "0", "0", "0", "3"])
    # connections
    self.interface.connect_models(__Sum, 0, __xdot, 0)
    self.interface.connect_models(__xdot, 0, __x, 0)
    self.interface.connect_models(__x, 0, __A, 0)
    self.interface.connect_models(__Bprev, 0, __Sum, 0)
    self.interface.connect_models(__Bnext, 0, __Sum, 1)
    self.interface.connect_models(__A, 0, __Sum, 2)
    # connection between father and childrens
    self.interface.connect_input_to_model(__Bprev, 0, 0) # left end
    self.interface.connect_input_to_model(__Bnext, 0, 1) # right end
    self.interface.connect_model_to_output(__x, 0, 0) # mass position

    return self._pd_model
```

Internal Couplings

External Input/Output Couplings

(b) py2PowerDEVS code automatically generated.

Figure 2: Mass-spring coupled model comprised of two integrators, three gains and a summation.

A mass-spring system is a simple and representative mechanical oscillatory system composed of masses attached to springs. Its dynamical behavior (depicted in Fig. 2a) is derived by applying the Newton equations: $M\ddot{x}(t) = -Kx(t)$, where $x(t)$ and $\ddot{x}(t)$ are the mass position and acceleration, M is the mass value and K the spring's elastic constant. Since there is no damping, the mass undergoes harmonic oscillations around its equilibrium position at a frequency given by M and K. This system has a single degree-of-freedom (DOF) and one natural mode of oscillation. To study vibrating systems more masses and springs can be attached in series combination leading to a set of ordinary differential equations. The Equation 1 corresponds to the case of N masses and $N+1$ springs.

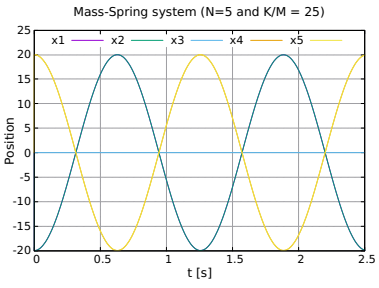
$$\begin{cases} \ddot{x}_1(t) & = -2K/M\,x_1(t) + K/M\,x_2(t) \\ \ddot{x}_i(t) & = -2K/M\,x_i(t) + K/M\,x_{i-1}(t) + K/M\,x_{i+1}(t) \quad \forall\, 2 \leq i \leq N-1 \\ \ddot{x}_N(t) & = -2K/M\,x_N(t) + K/M\,x_{N-1}(t) \end{cases} \tag{1}$$

We can have as many degrees-of-freedom and modes of oscillation as masses are in the system. By choosing suitable initial conditions for position and speed of all masses we can choose to excite either a single mode of oscillation or a composition of modes.

A single mass-spring model is implemented in PowerDEVS following the procedure described in Section 3.4 using the GUI and automatically exported to py2PowerDEVS libraries. A snippet of py2PowerDEVS generated code is shown in Figure 2b, where all submodels and interconnections in Figure 2a are coded. Once a single mass-spring model is available in Python, multiple instances of the model can be combined with simple loops as depicted in Figure 3b. In the GUI this would only be possible for a fixed value of N.

To analyze simulation results, let's consider the response to initial conditions of a mass-spring system composed by 5 masses and 6 springs ($N = 5$). The initial speed is zero, and in order to excite a single mode, the initial position for the two rightmost masses is $x = 20$, the leftmost masses start at $x = -20$ and the middle mass starts at $x = 0$. In Figure 3a we see the resulting simulation where the middle mass remains stationary during the experiment, while the others oscillate at a frequency proportional to $\sqrt{K/M}$.

This simple example shows how py2PowerDEVS can be a programmatic equivalent for a block-based model developed with the GUI, and how easily this model can be replicated with a Python script. The number of mass-springs can be changed by simply modifying a parameter (N), whereas with the GUI it requires to drag-and-drop individual mass-spring models (along with all their corresponding interconnections).



(a) Response to initial conditions.

```
import py2powerdevs.model as model                                  } Import
import py2powerdevs.library.sources as sources                        Py2PowerDEVS
from py2powerdevs.library.mechanics_ext import MassSpringExt          libraries

model = model.new_model('Mass-Spring0') # top model

massspring_prev = MassSpringExt(model,'MassSpring_0', x0=params["x1_0"], v0=params["v1_0"],
                                M=params["M1"], Kprev="25", Knext="25")
x_0 = sources.Constant(model,"x_0",K="0") # left end spring is tied to wall
model.connect((x_0,0),(massspring_prev,0))

for n in range(1, N):
  massspring_n = MassSpringExt(model,'MassSpring_%d' % n, x0=str(x0), v0=params["v1_0"],
                               M=params["M1"], Kprev="25", Knext="25")
  # connections
  model.connect((massspring_n,0),(massspring_prev,1)) # spring_{n-1} <-> spring_{n}
  model.connect((massspring_prev,0),(massspring_n,0)) # spring_{n} <-> spring_{n-1}
  if n == N-1: # right end of last spring in fixed position
    x_n = sources.Constant(model,"x_%d" % n,K="0") #leftmost spring tied to wall
    model.connect((x_n,0),(massspring_n,1))
  massspring_prev =  massspring_n
```

(b) py2PowerDEVS snippet for a system composed of N mass-spring modules.

Figure 3: Response for a system comprising $N = 5$ mass-spring in series connection. Due to the initial conditions, mass positions x_1 and x_2 (in green) and x_4 and x_5 (in yellow) evolve exactly overlapped.

4.2 Multiple Interconnected SIR Models to Analyze Virus Propagation

In this section, we show an example dedicated to stress the connectivity capacity of py2PowerDEVS. In this model we first define with the GUI a classic SIR (Susceptible-Infected-Recovered) model (Kermack and McKendrick 1927) using a set of non-linear differential equations as shown in Figure 4 (blue box). Furthermore, the green box represents the influence of neighbor SIR models. The connections between SIR models represent infections between cities or towns. Note that the total population remains constant in each SIR model. The rate of cross-district infections is defined by an adjacency matrix Λ. Each element i, j: $[\Lambda]_{i,j} = \lambda_{i,j}$ is a parameter of the model and represents persons in node i getting infected by persons in node j. The dynamics for node i are shown in Equation 2, where I_i, S_i and R_i are the infected, susceptible and recovered populations, respectively (ranging from 0 to 1), while β_i and γ_i are parameters that represent the infection and recovery rates.

$$\begin{cases} \dot{S}_i(t) & = -\beta_i S_i(t) I_i(t) - \sum_{j=1}^{N} \lambda_{i,j} \beta_i I_j(t) S_i(t) \\ \dot{I}_i(t) & = \beta_i S_i(t) I_i(t) - \gamma_i I_i(t) + \sum_{j=1}^{N} \lambda_{i,j} \beta_i I_j(t) S_i(t) \quad \forall 1 \leq i, j \leq N \\ \dot{R}_i(t) & = \gamma_i I_i(t) \end{cases} \tag{2}$$

Based on this, we used py2PowerDEVS to create a large network of interconnected SIR models, replicated in each node to represent independent sets of S, I and R compartments for several geographic regions. The models were set up using known populations for census radiuses of the City of Buenos Aires and its Metropolitan Area. The geographic locations were used to build an adjacency matrix for all node. In this way it is possible to define an irregular connectivity structure, in contrast to what can be defined with VectorialDEVS (limited to regular structures). This model represents a clear example of diffusion in time and space over a structure of SIR models.

To build this model we proceed as described in Section 3.4. First, the simulation model for a single SIR model is built with the GUI as shown in Figure 4. Then a graphical library is created with this model (`coronavirus.pdl`), which is then automatically exported to py2PowerDEVS (`coronavirus.py`) by means of the PAL code generator. This library is finally used with py2PowerDEVS to create a graph with a user-provided adjacency matrix. Model parameters (infection and recovery rates, initial conditions, etc.) are passed via command line with a parameters file. It is worth mentioning that they could have been defined directly in the script jointly with a strategy for parameter sweeping at no additional cost. The same Python script is used to perform post-processing tasks, such as plotting the evolution of S, I and R over time for different nodes and to build a heat map to visually summarize georeferenced infection levels of all nodes at each time instant.

The simulation results are shown in Figures 5a, 5b and 5c which showcase the infected population using interconnected georeferenced SIR models at three different times (20, 30 and 40 days). Each node represents a population cluster in the map. In the color scale, 1 represents the total population of each node. The infection spreads out from two initially infected nodes towards the borders as the simulation progresses. In this example, the py2PowerDEVS script accept several files to retrieve real values for the population of each node (census radiuses), the adjacency of nodes and the β_i and γ_i parameters. The full system couples 1431 SIR models interconnected by 9632 input/output links. The simulation took 64 s and consumed a maximum of 2397 MB of RAM in a standard desktop computer with Intel Core i5-6500 3.20GHz.

4.3 Modeling Large Configurable Data Network Topologies

In this section we describe how py2PowerDEVS allows the definition of large data networks when their structure depends on different conditions and involves a large number of interdependent configuration parameters. Such models would be practically impossible to define using the PowerDEVS GUI. In the past, these models were semi automatically defined with the help of domain-specific algorithms that tackle e.g. Software-Defined-Networks (Laurito et al. 2017) or parameter complexity (Foguelman et al. 2016).

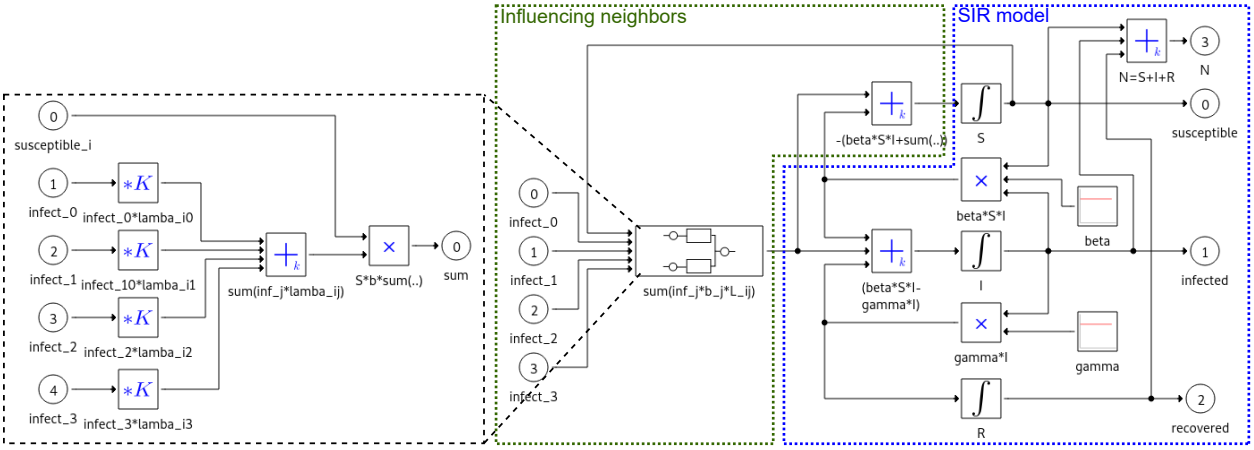
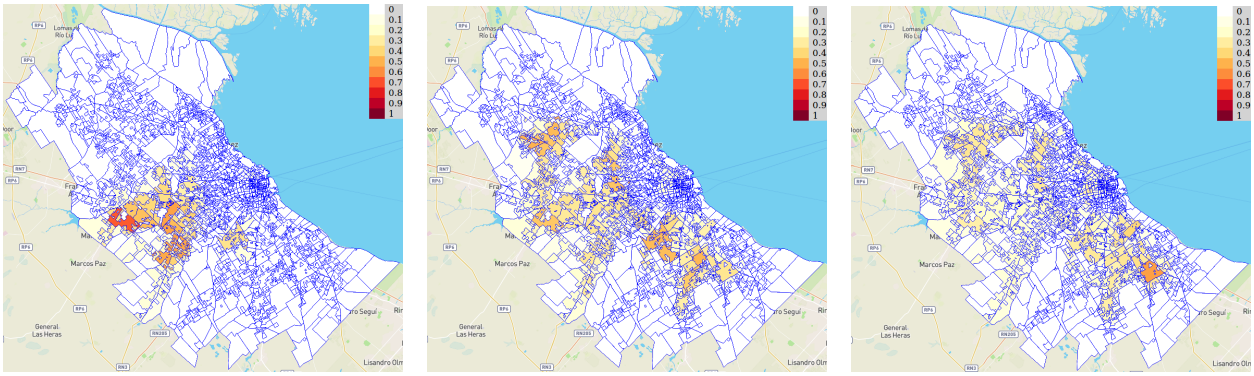


Figure 4: Single SIR model for an *i-th* node with $N = 4$ influencing neighbors built with the PowerDEVS GUI.



(a) Infected population at day 20. (b) Infected population at day 30. (c) Infected population at day 40.

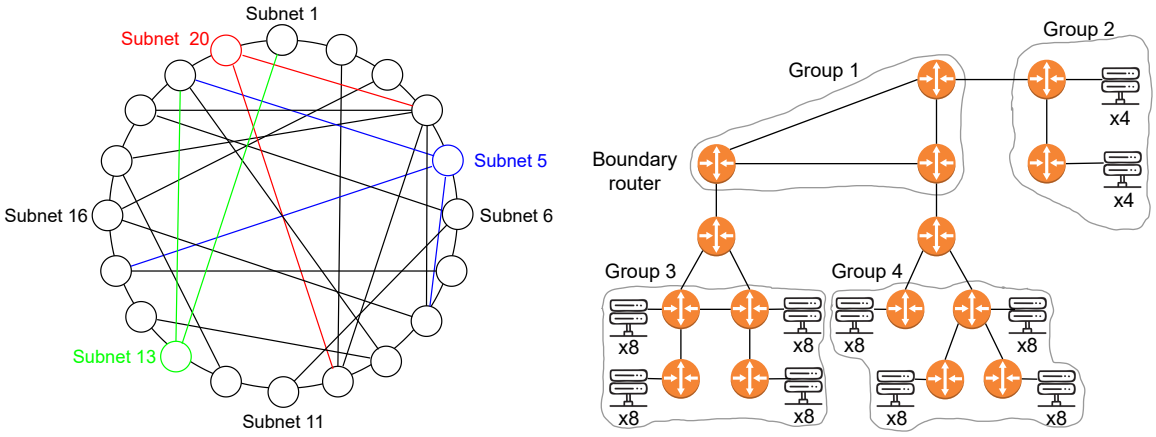
Figure 5: Multiple SIR models of neighboring population clusters interconnected with py2PowerDEVS. The spread of the virus is analyzed for Buenos Aires City and its surroundings. The proportion of infected population in each cluster is represented in shades of red (1 represents the whole population in a cluster)

py2PowerDEVS allowed all M&S tasks for this case study to be entirely developed within a Python environment.

The modeled topology in this case study is adapted from a typical ring-structure data network posed as a challenge to network simulators (Liu et al. 2003). DEVS-based network models such as TCP senders/receivers, routers, and links are part of the PowerDEVS library (Bonaventura et al. 2016; Bonaventura and Castro 2018) and as such they are now available to Python scripts through the new PAL described in Section 3.1

At a high level, the network is composed of a ring of $N \geq 3$ subnetworks interconnected to its two neighbors. In addition, each subnetwork is connected to another randomly chosen alternative subnetwork other than its neighbors (Figure 6a).

All subnetworks share the same topology consisting of 4 groups as depicted in Figure 6b. Router 0 in Group 1 is the boundary router that connects to boundary routers of other campus networks. Group 2 contains 2 routers and 4 application servers. Group 3 and Group 4 each contains 4 routers connecting to 8 servers each. Regarding the data transfer pattern, half of the servers in Groups 2 and 3 send data to servers in its neighboring subnetworks, and the other half send data to servers in the alternative subnetworks. All



(a) Ring Structure for $N = 20$.

(b) Topology of Subnetworks.

Figure 6: Topology of a large ring-structured data network with configurable number of nodes and subnetworks.

servers use K persistent TCP sessions with unlimited data to send. In total, the entire network has $15 \times N$ routers, $68 \times N$ servers, $88 + 2 \times N$ bidirectional links, and $64 \times N \times K$ sender/receiver TCP sessions pairs.

Defining this type of complex topologies with drag-and-drop methods would be very time consuming and error-prone, and would only be possible for a fixed N. On the contrary, py2PowerDEVS allows to define models using algorithms as sophisticated as required, providing greater flexibility to create complex structures. The Python code describing the network ring structure with subnetworks required 204 lines of code for any chosen value of N, compared to 2368 lines of C++ code generated from the GUI model only for the Group 2 in one subnetwork. Using the same hardware as in Section 5, simulating 1 s with N=10, K=5, 10 Gbps ring links, and 2.5 Gbps campus links, execution completes in 816 s using roughly 1.8 GB of RAM ($\approx 3.5e6$ simulated network packets). Performance scales linearly with the number of packets simulated (links speeds, number of nodes, number of TCP connections, etc).

py2PowerDEVS also facilitates the parameter definition task to configure large topologies. Network topologies requires setting thousands of parameters to define packet routing policies, flow paths across the network, and network devices. In PowerDEVS, simulation parameters can be defined by command line, through configuration files or in the GUI. But in complex network models, where each data flow path requires at least 5 parameters to populate routing tables, those approaches can easily become cumbersome and error-prone. Relying on py2PowerDEVS, which provides access to DEVS network models within a Python environment, new Python modules were implemented to automate such parameterization, allowing the modeler to define the data flow paths in a single line of code.

Parameter sweeping is another aspect where py2PowerDEVS greatly simplifies experimentation. A typical exercise in network simulation is to sweep the links' bandwidth to verify if congestion control algorithms can accommodate different scenarios. Link capacity can be easily set with few parameters when the model is defined either in the GUI or in py2PowerDEVS, but parameters for control algorithms such as Random-Early-Detection (RED) and TCP depend on link speeds and need to be updated accordingly (Hollot et al. 2001). When using the GUI the modeler can not define parameters that depend on the value of other parameters, so usually scripts specific for each problem must be prepared. Conversely, with py2PowerDEVS all dependent parameters can be easily defined using Python as part of the model definition. To sweep different bandwidths in the ring-structure topology, two links speeds need to be updated in each execution which in turn affect the value of $75 \times N + 64 \times N$ other parameters.

After the network model is simulated, results are stored in a standard HDF5 format which requires post-processing for aggregating, correlating and transforming network metrics before generating different plots.

These post-processing and visualization tasks are typically performed using the `py2powerdevs.params` module, while well known Python libraries (matplotlib, numpy, pandas, scipy, etc.) allow py2PowerDEVS to perform the modeling, parameterization and simulation tasks, so the modeler uses the same Python environment throughout the end-to-end modeling and simulation workflow.

5 DISCUSSION

The case studies in Section 4 showcased how py2PowerDEVS' new features enable and simplify the full M&S process for large models with complex topologies (modeling, parameterization, simulation, post-processing, and visualization). This also contributes to a faster learning curve.

However, py2PowerDEVS does not replace the GUI, but provides an alternative workflow. In addition, different workflows can emerge from the combination of GUI-based and scripting approaches.

Once in the Python domain, we can profit from the increasing number of libraries such as signal processing, optimization, machine learning and visualization. Another feature is that the experimentation workflow becomes more powerful than the GUI-based experience. Performing parameter sweeping experiments (as in Section 4.3) becomes straightforward by creating flexible algorithms that parameterize and adapt the structure of the models using simple scripts. However, a limitation of the new PAL code generator is that currently it can not produce models that can be accessed by the GUI, which is a very desirable bidirectional modeling capability (this feature is part of future work).

In terms of simulation performance, the execution of a model in py2PowerDEVS barely imposes performance penalties, since Python is used only at the stage of defining model structure, while the simulation of behavior (functions within DEVS atomic models) is fully executed in C++.

5.1 Simulation Performance of py2PowerDEVS

Regarding the performance overhead introduced by the Python binding, a system-level approach is to compare the total execution time of a model created with the PowerDEVS GUI (compiled C++) against the same model but defined and executed with py2PowerDEVS. To provide a quick insight we return to the interconnected mass-spring model (Section 4.1). Every experiment involves the simulation of 1000 s of evolution.

Figure 7 shows that in both cases the execution time increases linearly with an increasing number of mass-spring elements. Also, the performance difference appears as negligible (with py2PowerDEVS

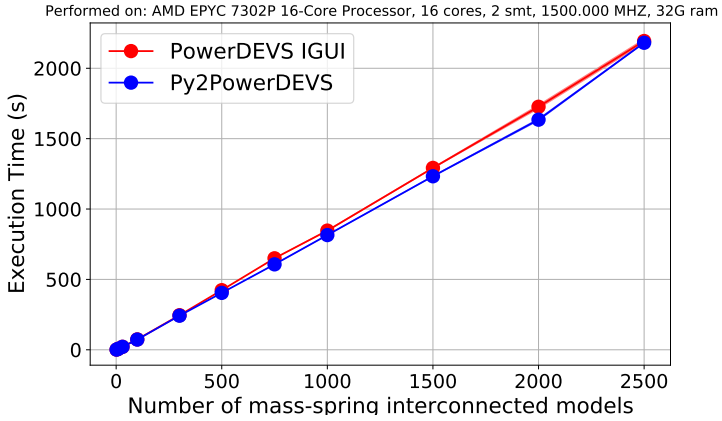


Figure 7: Performance comparison between py2PowerDEVS and compiled C++ for executions of the multiple interconnected mass-spring models. Each point represents the mean of 30 simulation repetitions (standard deviation is negligible).

approximately 1% faster in average). It is expected that both models yield similar execution times since in both cases most of the execution occurs in precompiled C++ libraries (the PowerDEVS core simulation engine and the user-defined atomic models) as shown in Figure 1. The Pyhton code is in charge only of creating the model structure and setting parameters, thus demanding a small fraction of execution time. This result, together with the performance studies in Van Tendeloo and Vangheluwe (2017) suggest that the py2PowerDEVS approach is among the most computationally efficient tools for DEVS M&S currently available. More experiments must be conducted to assess the extent to which these performance results can be generalized.

6 CONCLUSIONS AND FUTURE WORK

In this paper we presented py2PowerDEVS, a PowerDEVS binding to Python aimed at building models for systems with complex, large-scale structures. This way a DEVS model in PowerDEVS can be described using a powerful scripting language. Yet, py2PowerDEVS complements but does not replace the PowerDEVS GUI. A GUI provides advantages when a quick visual understanding of a model is required, or in cases when the modeler is not familiar with scripting or programming.

Resorting to a variety of examples from dissimilar domains we can conclude that py2PowerDEVS is effective at greatly simplifying the modeling workflow for systems with large and/or complex structures.

In the first example, we consider a model composed of multiple instances of a simpler system, described by a single ordinary differential equation, interconnected with a regular structure. This simple example showed how easy it can be to get a programmatic equivalent of the original model developed with the GUI, and how the number of replicas is easily changed with a single parameter. Next, a second example showed a model with a graph of interconnected nodes, where the behavior of the nodes was described by a set of nonlinear differential equations. The structure in this case was irregular concerning both the number and intensity of links, and based upon real data of relations between population clusters. In this case, we took advantage of the parameterization and post-processing capabilities offered by py2PowerDEVS to read realistic parameters from .csv files and produce useful georeferenced plots. This model is being used as part of a study of the spread of SARS-CoV-2 in Argentina.

Finally, the third example presented a very large network infrastructure involving many computing elements (servers, routers, switches) grouped in subnets with complex interconnections and parameters which would be almost impossible (and error-prone) to be built manually handling blocks with the GUI. py2PowerDEVS is currently being used for the simulation of a high-throughput data acquisition network ($\sim$3000 servers, 5.2 TB/s) in the ATLAS experiment at CERN to study the impact of network upgrades in a distributed architecture (Bonaventura et al. 2016; Abud et al. 2021).

Preliminary experiments indicate that the new features do not impose any extra overhead to the simulation performance. An interesting avenue for future development is to provide GUI support for atomic model definition directly from py2PowerDEVS. This could offer the best of both worlds: building new DEVS atomic models through a graphical block-oriented paradigm with a GUI, combined with advanced scripting capabilities for model structure definition, all within a same py2PowerDEVS framework.

REFERENCES

Abud, A. A., M. Bonaventura, E. Farina, and F. Le Goff. 2021. "Design of a Resilient, High-Throughput, Persistent Storage System for the ATLAS Phase-II DAQ System". In *EPJ Web of Conferences*, Volume 251, 04014. EDP Sciences.

Bergero, F., and E. Kofman. 2011. "PowerDEVS: A Tool for Hybrid System Modeling and Real-time Simulation". *Simulation* 87:113–132.

Bergero, F., and E. Kofman. 2014. "A Vectorial DEVS Extension for Large Scale System Modeling and Parallel Simulation". *Simulation* 90(5):522–546.

Bonaventura, M., and R. Castro. 2018. "Fluid-flow and Packet-level Models of Data Networks Unified under a Modular/Hierarchical Framework: Speedups and Simplicity, Combined". In *Proc. of the 2018 Winter Simulation Conference*, edited by M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, 3825–3836. Piscataway, New Jersey: IEEE: Institute of Electrical and Electronics Engineers, Inc.

Bonaventura, M., D. Foguelman, and R. Castro. 2016. "Discrete Event Modeling and Simulation-driven Engineering for the ATLAS Data Acquisition Network". *Computing in Science & Engineering* 18(3):70–83.

BoostLib 2022. "Boost.Python library". Available at https://www.boost.org/, accessed 18th April.

Capocchi, L., J. Santucci, B. Poggi, and C. Nicolai. 2011. "DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems". In *2011 IEEE 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 170–175. June 27th-29th, Paris, France.

Castro, R., E. Pecker-Marcosig, and J. I. Giribet. 2019. *Simulation Model Continuity for Efficient Development of Embedded Controllers in Cyber-Physical Systems*, Chapter 8, 191–222. John Wiley & Sons, Ltd.

Cellier, F. E. 1991. *Continuous System Modeling*. 1 ed. Springer New York.

Foguelman, D., M. Bonaventura, and R. Castro. 2016. "MASADA: A Modeling and Simulation Automated Data Analysis Framework for Continuous Data-intensive Validation of Simulation Models". In *30th European Simulation and Modelling Conference*, Volume 30, 34–42. October 26th-28th, Las Palmas, Spain.

Folk, M., G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. 2011. "An Overview of the HDF5 Technology Suite and its Applications". In *Proc. of the EDBT/ICDT 2011 Workshop on Array Databases*, 36–47. New York, NY, USA: March 25th, Uppsala, Sweden.: Association for Computing Machinery.

Hollot, C. V., V. Misra, D. Towsley, and W.-B. Gong. 2001. "A Control Theoretic Analysis of RED". In *Proc. of IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, Volume 3, 1510–1519. IEEE.

Kermack, W. O., and A. G. McKendrick. 1927. "A Contribution to the Mathematical Theory of Epidemics". *Proc. of the royal society of london. Series A* 115(772):700–721.

Laurito, A., M. Bonaventura, M. E. P. Astigarraga, and R. Castro. 2017. "TopoGen: A Network Topology Generation Architecture with Application to Automating Simulations of Software Defined Networks". In *Proc. of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, G. W. N. Mustafee, and E. Page, 1049–1060. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers Inc.

Liu, Y., F. Lo Presti, V. Misra, D. Towsley, and Y. Gu. 2003, 06. "Fluid Models and Solutions for Large-Scale IP Networks". *SIGMETRICS Performance Evaluation Review* 31(1):91–101.

Quesnel, Gauthier and Ramat, Eric 2022. "PyVLE Github Repository". Available at https://github.com/vle-forge/pyvle, accesed 18th April.

Risco-Martín, J.L. 2014. "xDEVS: M&S Framework". Available at https://github.com/iscar-ucm/xdevs, accesed 28th June.

SEDLab 2022. "py2PowerDEVS gitlab repository". Available at https://git-modsimu.exp.dc.uba.ar/sedlab/powerdevs3.0/, accesed 9th July.

Van Tendeloo, Y., and H. Vangheluwe. 2015. "PythonPDEVS: A Distributed Parallel DEVS Simulator". In *Proc. of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS '15, 91–98. San Diego, CA, USA: Society for Computer Simulation International.

Van Tendeloo, Y., and H. Vangheluwe. 2017. "An Evaluation of DEVS Simulation Tools". *Simulation* 93:103–121.

Vangheluwe, H. 2000. "DEVS As a Common Denominator for Multi-Formalism Hybrid Systems Modelling". In *Proc. of the IEEE International Symposium on Computer-Aided Control System Design*, CACSD'00, 129–134: IEEE.

Zeigler, B. P., A. Muzy, and E. Kofman. 2018. *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. 3 ed. San Diego, CA, USA: Academic Press.

AUTHOR BIOGRAPHIES

EZEQUIEL PECKER-MARCOSIG is a posdoctoral fellow at the Research Institute of Computer Science, University of Buenos Aires - CONICET. His research are M&S-driven controller design for cyber-physical systems. Email: emarcosig@dc.uba.ar.

MATIAS BONAVENTURA is a postdoctor in the CONICET Research Institute of Computer Science (ICC) and fellow the ATLAS TDAQ group at CERN. His research interests are networked and distributed systems. Email: mbonaventura@dc.uba.ar.

ESTEBAN LANZAROTTI is a postdoctoral researcher in the CONICET Research Institute of Computer Science (ICC). His research interests are hybrid simulations in biological and social systems. Email : elanzarotti@dc.uba.ar.

LUCIO SANTI is a postdoctoral researcher in the Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, where he earned his PhD in Computer Science in 2020. Email: lsanti@dc.uba.ar.

RODRIGO CASTRO is a Professor with the University of Buenos Aires (UBA) and Head of the Laboratory on Discrete Events Simulation at the CONICET Research Institute of Computer Science (ICC). His research includes simulation and control of hybrid systems. Email: rcastro@dc.uba.ar.