The Journal of Open Source Software

DE-Sim: an object-oriented, discrete-event simulation tool for complex, data-driven modeling in Python

Arthur P. Goldberg[1] **and Jonathan R. Karr**[1]

1 Icahn Institute for Data Science and Genomic Technology, and Department of Genetics and Genomic Sciences, Icahn School of Medicine at Mount Sinai, New York, NY 10029, USA

DOI: 10.21105/joss.0XXXX

Software

- Review
- Repository
- Archive

Editor: Editor Name

Submitted: 01 January XXXX
Published: 01 January XXXX

Summary

Recent advances in data collection and storage have created unprecedented opportunities to gain insights into complex systems such as the biochemical networks that generate cellular behavior. Understanding the behavior of such systems will likely require larger and more comprehensive dynamical models that are based on a combination of first principles and large datasets. Large models often contain multiple types of components, and multiple types of interactions between component types. The first principles typically use a mechanistic approximation of the interactions in a system, for example, that reactants in a chemical reaction may bond when they collide, to derive mathematics that must be parameterized by data.

One of the most promising methods for building and simulating large, data-driven dynamical models is to simulate them with discrete-event simulation (DES) (Fishman, 2013). For example, discrete-event simulations are frequently used to study the dynamics of biochemical networks, characterize the performance of computer networks, evaluate potential military strategies, and forecast epidemics (Banks, Carson II, Nelson, & Nicol, 2009). However, it is difficult to design, encode and simulate large, comprehensive models of complex systems as discrete-event simulations because the existing DES tools lack adequate expressive power.

To address this problem, we present DE-Sim, an open-source, object-oriented (OO), Python-based DES tool that makes it easier to create and use dynamical models of complex systems. Because DE-Sim encodes discrete-event simulations in OO Python programs, numerous types of components and complex interactions can be modeled by leveraging Python's powerful OO features. In addition, because DE-Sim is implemented in Python, DE-Sim models can easily use Python's powerful data science tools such as pandas and SciPy to help build complex models, store and analyze data during simulations, and analyze simulation results. We anticipate that DE-Sim will enable a new generation of models which capture systems with unprecedented breadth and depth. An initial example is a multi-algorithmic simulator that we are developing on DE-Sim to simulate whole-cell models that predict phenotype from genotype by capturing all of the biochemical activity in a biological cell.

The need for tools that help researchers build and simulate complex models

Many scientific fields can now collect detailed data about the components of complex systems and their interactions. For example, the revolution in deep sequencing has dramatically increased the availability of molecular data. When the measurements of a complex system generate a large, heterogeneous dataset, models of the system that use the measurements are

invariably complex and large because the dataset contains measurements of many types of components, and many instances of each component. For example, measurements of the biochemistry in a cell collect the properties of many types of molecules, many types of processes that transform or translocate molecules, and many instances of each type. These data mirror the intrinsic complexity of cellular biology and lead to complex and large models of cells.

DE-Sim simplifies the construction and simulation of *discrete-event models* which represent the dynamics of a complex system as a sequence of instantaneous events. Dynamical models constructed with DE-Sim can leverage Python's extensive suite of high-quality data science tools to easily manage and integrate large, heterogeneous, multidimensional data into models. For example, tools such as NumPy (Oliphant, 2015), pandas (McKinney & others, 2010), SciPy (Virtanen et al., 2020), and SQLAlchemy (Bayer, 2020) can be used by DE-Sim models to store and integrate model inputs, simplify analyses and data storage during simulation, and organize and save predictions for downstream analysis.

Discrete-event models of complex systems are challenging to construct. DE-Sim addresses this challenge by structuring discrete-event models as object-oriented programs. This approach, known as *object-oriented discrete-event simulation* (OO DES), implements the component types in a model as simulation object classes and implements their interaction events as *event messages* that schedule the simulation objects to execute events at specified times (Zeigler, 1987). With DE-Sim, users define classes of simulation object by subclassing DE-Sim's simulation object class and specifying their behavior. Complex systems that contain multiple types of components can be easily modeled in DE-Sim by creating multiple classes of simulation objects. Users can model arbitrarily many instances of a type of component by creating multiple instances of the corresponding simulation object class. DE-Sim simulation object classes can exploit all the features of Python objects. For example, hierarchical relationships among the components in a system being modeled can be mirrored by subclass relationships among the simulation object classes that represent the components in a DE-Sim model. Thus, a model of a biochemical system that represents macromolecule components such as DNA and RNA could define a macromolecule class, and then define DNA and RNA subclasses of the macromolecule class. In our experience, DE-Sim's use of OO programming to construct discrete-event models simplifies and accelerates the design and development of models.

DE-Sim is designed for scientists and engineers who want to build and use quantitative, dynamical models to understand the properties of complex, discrete-time systems. DE-Sim's features address the needs of this audience: it uses Python, one of the most popular languages; it is open-source; it is easy to learn because it provides several tutorials, examples, and documentation; and it is thoroughly tested.

Summary of DE-Sim's key features

DE-Sim provides the following features that help users design, build and simulate complex, data-driven, discrete-event models:

- **Object-oriented modeling:** Modelers who use DE-Sim write their models as object-oriented Python programs. This simplifies the construction of complex models.
- **Access Python's comprehensive data-science tools:** Researchers who use DE-Sim can employ Python's powerful, comprehensive data science packages such as NumPy, pandas, SciPy, and SQLAlchemy to easily integrate large, heterogeneous datasets and complex analyses into their models.
- **Simple simulation logging:** DE-Sim supports easily-configured, high-performance Python logs which can log simulation data that help users debug their models.
- **Checkpointing of simulation state:** DE-Sim can checkpoint the state of a simulation to a file. A record of the predictions made by a simulation run are easily obtained by

subclassing a DE-Sim abstract class that creates periodic checkpoints. In addition, DE-Sim automatically records configuration information such as simulation run arguments and metadata such as the start time and duration of a simulation.

- **Powerful stop conditions:** DE-Sim simulations can easily implement complex stop conditions. A model simply specifies Python functions that return true when the simulation should be terminated.
- **Space-time visualizations for analysis and debugging:** DE-Sim can generate space-time visualizations of simulation trajectories (Figure 1). These diagrams help design, understand and debug models. DE-Sim automatically generates these diagrams from log files.
- **Reproducible simulations:** DE-Sim simulation runs are *reproducible*, which means that repeated executions of a simulation with the same input – including seeds for random number generators – will produce exactly the same simulation trajectories and output.
- **Controlled, reproducible execution of simultaneous events:** An OO discrete-event simulation may contain simultaneous events. A simulation object may receive multiple events simultaneously, and multiple simulation objects may receive events at the same simulation time. In both of these cases DE-Sim provides discrete-event models with full and convenient control over the execution order of simultaneous messages, as documented in a Jupyter notebook.

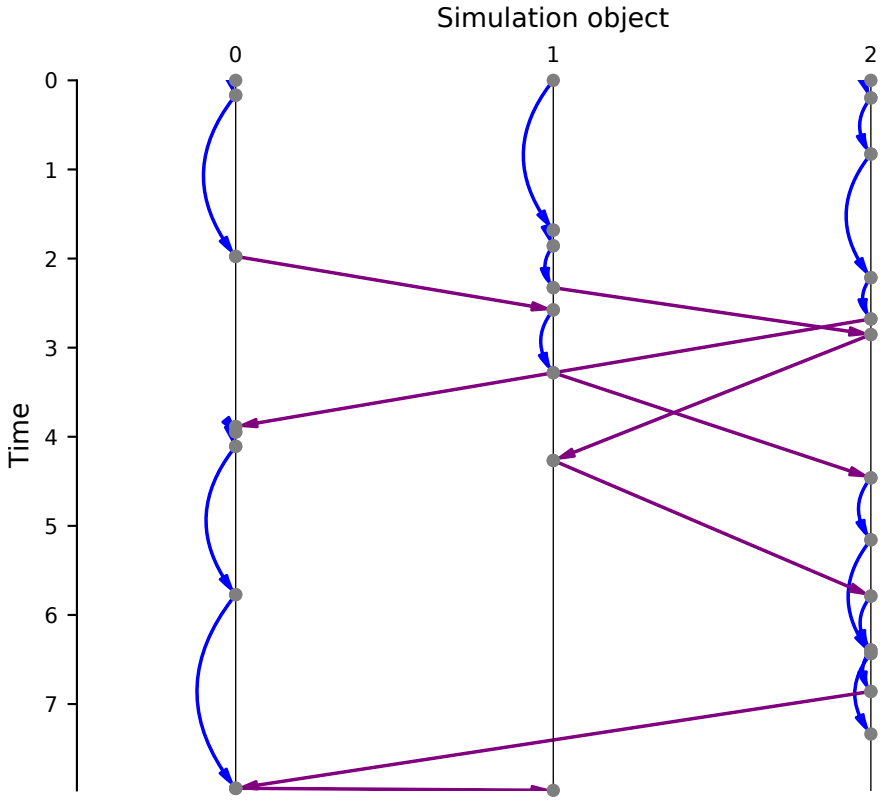

Figure 1: DE-Sim can generate space-time visualizations of simulation trajectories. This figure illustrates a space-time visualization of all of the events and messages in a simulation of the parallel hold (PHOLD) DES benchmark model (Fujimoto, 1990) with three simulation objects. The timeline (grey line) for each object shows its events (grey dots). Each event in PHOLD schedules one other event. Each event e is illustrated by an event message (arrow) that the simulation sends from the event that schedules e to e itself. The curved blue arrows indicate events scheduled by a simulation object for itself, while the straight purple arrows indicate event messages sent to other simulation objects. The programs for the PHOLD model and for visualizing the trajectory of any simulation are available in the DE-Sim Git repository.

The following sections describe how DE-Sim provides these features and how they help researchers build models of complex systems.

Comparison of DE-Sim with existing discrete-event simulation tools

Multiple DES tools already exist. Figure 2 lists some of the more popular tools for modeling nature and engineered systems. All of these tools provide a programming environment for developing DES models, a simulator that simulates models, and methods for recording simulation predictions.

Simulation tool	Application domain	Modeling language	Object-oriented models	GUI model builder	Open-source	Latest update
DE-Sim	Complex, data-intensive models	Python	✓		✓	2020
SimEvents	Communications networks and process flows	MATLAB	✓	✓		2020
SimPy	General purpose process-based framework	Python			✓	2018
SIMSCRIPT III	Object-oriented simulation of engineered systems	SIMSCRIPT III	✓			2020
SIMUL8	Business processes	Visual Logic		✓		2020
SystemC	Digital hardware	C++	✓		✓	2018

Figure 2: Comparison of DE-Sim with important existing DES tools. DE-Sim is the only open-source, object-oriented, discrete-event simulation tool based on Python. This combination of features makes it uniquely suitable for creating discrete-event models that can be used to study complex systems.

SimEvents (Clune, Mosterman, & Cassandras, 2006) is a MATLAB-based tool that adds DES capabilities to the Simulink environment for modeling dynamical systems. Its graphical interface enables visual definitions of queueing models of networks and processes. MATLAB code can also specify a SimEvents model's behavior. SimEvents obtains its OO modeling functionality from the OO features of MATLAB. However, MATLAB lacks Python's extensive library of scientific data analysis packages and broad adoption by researchers, which limits SimEvents' suitability for analyzing complex systems.

SimPy (Matloff, 2008) is an open-source, general-purpose DES tool which can leverage Python's data-science packages and popularity because it's written in Python. However, SimPy models are defined with functions that do not have the flexibility and power of the objects that DE-Sim employs to define models of complex systems. In addition, DE-Sim supports a uniform approach for scheduling events, whereas SimPy models that contain multiple processes that schedule events for themselves and each other must use two approaches. A DE-Sim object always schedules an event by sending an event message to the object that will execute the event. However, a SimPy process schedule events for itself by using a timeout call and Python's `yield` function, but schedules events for other processes by raising an interrupt exception.

SIMSCRIPT III (Rice, Marjanski, Markowitz, & Bailey, 2005) is a commercial DES tool that describes models in SIMSCRIPT III, a proprietary, object-oriented language. It's well suited for

modeling decision support systems in domains such as war-gaming, communications network analysis, and transportation and manufacturing. But DE-Sim is more appropriate for modeling complex systems because it can access Python packages.

SIMUL8 (Concannon, Hunter, & Tremble, 2003) specializes in modeling business processes. Like other DES tools that use proprietary languages it is not well designed for building scientific models of complex systems.

Lastly, SystemC (Mueller et al., 2001) is an OO DES tool based on C++ that is commonly used to model the design and performance of digital systems. DE-Sim is more accessible to scientific researchers than SystemC, because DE-Sim builds upon Python whereas SystemC uses C++, a lower-level language.

An important benefit of OO DES tools like DE-Sim and the other OO DES tools in Figure 2 is that individual simulation runs may be sped up by parallel execution on multiple cores. More precisely, the simulation of an OO DES model composed of simulation objects that only interact with each other via event messages and do not access shared memory might be sped up by distributing its objects across multiple cores and executing them in parallel. This simulation would need to be synchronized by a parallel DES simulator, such as Time Warp (Jefferson, 1985, p. @carothers2000ross). SystemC has been parallelized (Schumacher, Leupers, Petras, & Hoffmann, 2010), for example. Parallel DES simulations can achieve substantial speedup, as was demonstrated by running the PHOLD benchmark on nearly 2 million cores (Barnes, Carothers, Jefferson, & Lapre, 2013).

In summary, the primary advantages of DE-Sim are that it combines the power and convenience of OO modeling with the ability to leverage Python's library of data science tools to manage and analyze the large datasets needed by models of complex systems.

Tutorial: Building and simulating models with DE-Sim

A simple DE-Sim model can be defined in three steps: define an event message class; define a simulation object class; and build and run a simulation. We illustrate this process with a model of a random walk on the integer number line (this example is available in a Jupyter notebook).

1: Create an event message class by subclassing `EventMessage`.

Each DE-Sim event contains an event message that provides data to the simulation object which executes the event. The random walk model sends event messages that contain the value of a random step.

```
import de_sim

class RandomStepMessage(de_sim.EventMessage):
    "An event message class that stores the value of a random walk step"
    attributes = ['step']
```

The attribute `attributes` is a special attribute of a `EventMessage` that specifies the names of an event message class' attributes. These names must be valid Python identifiers. `attributes` is optional.

An event message class must be documented by a docstring, as illustrated.

2: Define a simulation object class by subclassing `SimulationObject`.

Simulation objects are like threads, in that a simulation's scheduler decides when to execute them, and their execution is suspended when they have no work to do. But DES simulation

objects and threads are scheduled by different algorithms. Whereas a thread can be scheduled whenever it has work to do, a DES scheduler schedules simulation objects to ensure that events occur in simulation time order, as summarized by the fundamental invariant of discrete-event simulation:

1. All events in a simulation are executed in non-decreasing time order.

By guaranteeing this behavior, the DE-Sim scheduler ensures that causality relationships between events are respected. (The invariant says *non-decreasing* instead of *increasing* time order because events can occur simultaneously, as discussed above.)

This invariant has two consequences:

1. All synchronization between simulation objects is controlled by the simulation times of events.
2. Each simulation object executes its events in non-decreasing time order.

The Python classes that generate and handle simulation events are simulation object classes, which are defined as subclasses of `SimulationObject`. DE-Sim provides a custom class creation method for `SimulationObject` that gives special meaning to certain methods and attributes.

Below, we define a simulation object class that models a random walk, and illustrates all key features of `SimulationObject`. To add variety to its temporal behavior we modify the traditional random walk by randomly selecting the time delay between steps.

```
import random

class RandomWalkSimulationObject(de_sim.SimulationObject):
    " A 1D random walk model, with random delays between steps "

    def __init__(self, name):
        super().__init__(name)

    def init_before_run(self):
        " Initialize before a simulation run; called by the simulator "
        self.position = 0
        self.history = {'times': [0],
                        'positions': [0]}
        self.schedule_next_step()

    def schedule_next_step(self):
        " Schedule the next event, which is a step "
        # A step moves -1 or +1 with equal probability
        step_value = random.choice([-1, +1])
        # The time between steps is 1 or 2, with equal probability
        delay = random.choice([1, 2])
        # Schedule an event `delay` in the future for this object
        # The event contains a `RandomStepMessage` with `step=step_value`
        self.send_event(delay, self, RandomStepMessage(step_value))

    def handle_step_event(self, event):
        " Handle a step event "
        # Update the position and history
        step = event.message.step
```

```
        self.position += step
        self.history['times'].append(self.time)
        self.history['positions'].append(self.position)
        self.schedule_next_step()

    # `event_handlers` contains pairs that map each event message class
    # received by this simulation object to the method that handles
    # the event message class
    event_handlers = [(RandomStepMessage, handle_step_event)]

    # messages_sent registers all message types sent by this object
    messages_sent = [RandomStepMessage]
```

DE-Sim simulation objects (subclasses of `SimulationObject`) use these special methods and attributes:

- Special `SimulationObject` methods:
    1. **`init_before_run`** (optional): immediately before a simulation run, after all simulation objects have been added to a simulator, the simulator calls each simulation object's `init_before_run` method. In this method simulation objects can send initial events and perform other initializations. For example, in `RandomWalkSimulationObject`, `init_before_run` schedules the object's first event and initializes the object's position and history attributes.
    2. **`send_event`**: `send_event(delay, receiving_object, event_message)` schedules an event to occur `delay` time units in the future at simulation object `receiving_object`. `event_message` must be an `EventMessage` instance. An event can be scheduled for any simulation object in a simulation, including the object scheduling the event, as `RandomWalkSimulationObject` does. The event will be executed at its scheduled simulation time by an event handler in the simulation object `receiving_object`. The handler defines an `event` parameter. Its value will be the scheduled event, which contains `event_message` in its `message` attribute. Object-oriented DES terminology often describes the event message as being sent by the sending object at the message's send time (the simulation time when the sending object schedules the event) and being received by the receiving object at the event's receive time (the simulation time when the event is executed). An event message can thus be viewed as a directed edge in simulation space-time from the pair (sending object, send time) to (receiving object, receive time), as illustrated by Figure 1.
    3. **event handlers**: an event handler is a method that handles and executes a simulation event. Event handlers have the signature `event_handler(self, event)`, where `self` is the simulation object that handles (receives) the event, and `event` is a simulation event. A subclass of `SimulationObject` must define at least one event handler, as illustrated by `handle_step_event` in the example above.
- Special `SimulationObject` attributes:
    1. **`event_handlers`**: a simulation object can receive arbitrarily many types of event messages, and implement arbitrarily many event handlers. The attribute `event_handlers` must contain an iterator over pairs that map each event message class received by a `SimulationObject` subclass to the event handler which handles the event message class. In the example above, `event_handlers` associates `RandomStepMessage` event messages with the `handle_step_event` event handler.
    2. **`messages_sent`**: the types of messages sent by a subclass of `SimulationObject` must be listed in `messages_sent`. It ensures that a simulation object doesn't send messages of the wrong `EventMessage` class.

3. `time`: `time` is a read-only attribute that always equals the current simulation time in every simulation object. For example, a `RandomWalkSimulationObject` saves the value of `time` when recording its history.

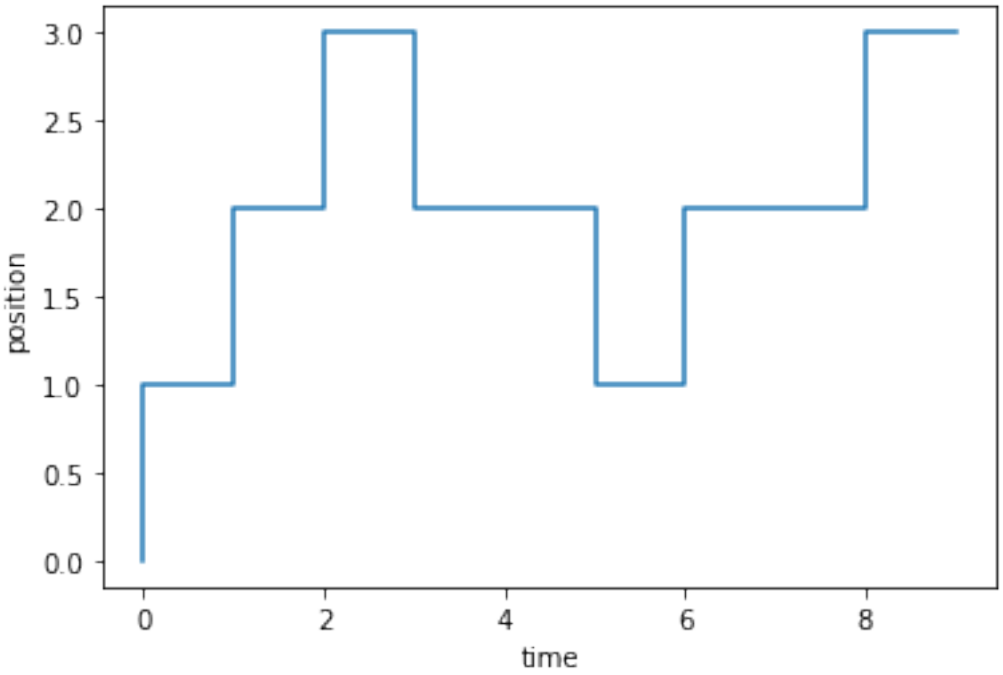

Figure 3: Trajectory of a simulation of a model of a random walk on the integer number line. The random walk model starts at position 0 and moves +1 or -1 with equal probability at each step. Steps take place every 1 or 2 time units, also with equal probability. This trajectory illustrates two key characteristics of discrete-event models. First, the state, in this case the position, changes at discrete times. Second, since the state does not change between instantaneous events, the trajectory of any state variable is a step function. The source code for this model is available in the DE-Sim Git repository.

3: Execute a simulation by creating and initializing a `Simulator`, and running the simulation.

The `Simulator` class simulates models. Its `add_object` method adds a simulation object to the simulator. Each object in a simulation must have a unique `name`. The `initialize` method, which calls each simulation object's `init_before_run` method, must be called before a simulation starts. At least one simulation object in a simulation must schedule an initial event–otherwise the simulation cannot start. More generally, a simulation with no events to execute will terminate. Finally, `run` simulates a model. It takes the maximum time of a simulation run. `run` also takes several optional configuration arguments, as described in the DE-Sim API documentation.

```
# Create a simulator
simulator = de_sim.Simulator()

# Create a random walk simulation object and add it to the simulation
random_walk_sim_obj = RandomWalkSimulationObject('rand_walk')
simulator.add_object(random_walk_sim_obj)

# Initialize the simulation
# This executes `init_before_run` in `random_walk_sim_obj`
```

```
simulator.initialize()

# Run the simulation until time 10
max_time = 10
simulator.run(max_time)

# Plot the random walk
import matplotlib.pyplot as plt
plt.step(random_walk_sim_obj.history['times'],
         random_walk_sim_obj.history['positions'])
plt.xlabel('time')
plt.ylabel('position')
plt.show()
```

This example runs a simulation for `max_time` time units, and plots the random walk's trajectory (Figure 3). This tutorial and additional examples are available in a Jupyter notebook.

Performance of DE-Sim

Figure 4 shows the performance of DE-Sim simulating a model of a cyclic messaging network over range of network sizes. The code for this performance test is available in the DE-Sim Git repository and a Jupyter notebook that runs the test.

		Run time (s)		Event rate (events s^{-1})	
Nodes	**Events**	**Mean**	**Std dev**	**Mean**	**Std dev**
4	400	0.05	0.01	8,930	1,930
16	1,600	0.10	0.02	16,531	3,052
64	6,400	0.31	0.04	20,986	2,734
256	25,600	1.22	0.16	21,139	2,597
1,024	102,400	5.34	0.96	19,573	3,465
4,096	409,600	20.06	1.53	20,491	1,506

Figure 4: Performance of DE-Sim simulating a model of a cyclic messaging network over a range of network sizes. Each statistic represents the average of three simulation runs in a Docker container on a 2.9 GHz Intel Core i5 processor. The cyclic messaging network model consists of a ring of simulation objects. Each simulation object executes an event at every time unit and schedules an event for the next object in the ring 1 time unit in the future. The number of simulation objects in the ring is given by **Nodes**. Each simulation run executes for 100 time units.

Case study: a multi-algorithmic simulation tool for whole-cell modeling implemented using DE-Sim

We have used DE-Sim to develop WC-Sim (Goldberg & Karr, 2020), a multi-algorithmic simulator for comprehensive whole-cell models of the biochemical dynamics inside biological cells (Goldberg et al., 2018; Karr et al., 2012; Karr, Takahashi, & Funahashi, 2015). Whole-cell models which predict phenotype from genotype by representing all of the biochemical activity in a cell have great potential to help scientists elucidate the basis of cellular behavior, help bioengineers rationally design biosensors and biomachines, and help physicians personalize medicine.

Due to the diverse timescales of the reactions inside cells, one promising way to simulate whole-cell models is to simulate each reaction with an appropriate algorithm for its timescale. For example, slow biochemical reactions, such as transcription, can be simulated with the Stochastic Simulation Algorithm (SSA, Gillespie (1977)). Faster processes, such as signal transduction, can be simulated with ordinary differential equations (ODEs). Metabolism, another fast process, can be simulated with flux-balance analysis (FBA, Orth, Thiele, & Palsson (2010)). Simulating entire cells requires co-simulating SSA, ODE and FBA. However, tools for co-simulating these algorithms did not exist before we created WC-Sim.

To accelerate whole-cell modeling, we have created WC-Sim, a tool for simulating multi-algorithmic whole-cell models described in the WC-Lang language (Karr, Goldberg, & Chew, 2020). We implemented WC-Sim by using DE-Sim to construct separate simulation object classes for SSA, ODE, and FBA. A cell's state is given by the populations of its molecular species, which are stored in an object that is shared by all simulation objects. DE-Sim event messages schedule the activities of each simulation object, while the exact simulation time of events is used to coordinate the objects' shared access to the cell's state. DE-Sim's object-oriented modeling functionality made it easy to separately develop SSA, ODE, and FBA simulation object classes and compose them into a multi-algorithmic simulator. DE-Sim's discrete-event framework provided the control needed to synchronize the interactions between these classes. In addition, accessing Python's data-science tools reduced the effort required to build WC-Sim. It uses NumPy's random number generator for stochastic simulation, and its arrays to store and compare molecule counts; pandas DataFrames store simulation predictions and transfer them to and from files; directed graphs and the DFS algorithm in networkx analyze reaction network dependencies (Hagberg, Swart, & S Chult, 2008); the ODE solver in scikits.ODES determines the rate of change of species populations modeled by ODEs (Malengier et al., 2018); and matplotlib visualizes simulation predictions (Hunter, 2007). We anticipate that WC-Sim will enable researchers to conduct unprecedented simulation studies of cellular biochemistry.

Conclusions

DE-Sim is an open-source, object-oriented, discrete-event simulation tool implemented in Python. We encourage researchers who use discrete-event models to understand the dynamics of complex systems to try DE-Sim because it combines two features that are not available together in other tools. First, discrete-event models defined in DE-Sim are constructed from multiple, interacting simulation objects. This gives modelers the ability to intuitively model a complex system that contains multiple types of interacting components by encoding each component type as a different DE-Sim simulation object class. Second, to conveniently store, manage and analyze the large, heterogeneous data needed by models of complex systems, modelers using DE-Sim can access Python's extensive library of data science packages.

We anticipate that DE-Sim will enable the construction and use of ambitiously detailed models of complex systems, and that simulation studies conducted with these models contribute to important engineering innovations and scientific discoveries.

Availability of DE-Sim

DE-Sim is freely and openly available under the MIT license at the locations below.

- Python package: PyPI: de-sim
- Docker image: DockerHub: karrlab/de_sim
- Tutorials: Jupyter notebooks at https://sandbox.karrlab.org

- Installation instructions and documentation of DE-Sim's API: docs.karrlab.org
- Issue tracker: GitHub: KarrLab/de_sim
- Source code: GitHub: KarrLab/de_sim
- Guide to contributing to DE-Sim and code of conduct for developers: GitHub: KarrLab/de_sim
- Continuous integration: CircleCI: gh/KarrLab/de_sim

DE-Sim requires Python 3.7 or higher and pip.

This article discusses version 0.0.8 of DE-Sim.

Acknowledgements

We thank Yin Hoon Chew for her helpful feedback on this paper. This worked was supported by the National Science Foundation [award 1649014 to J.R.K.], the National Institutes of Health [award R35GM119771 to J.R.K], and the Icahn Institute for Data Science and Genomic Technology.

References

Banks, J., Carson II, J., Nelson, B., & Nicol, D. (2009). *Discrete-event system simulation*. Pearson. ISBN: 978-0136062127

Barnes, P. D., Jr, Carothers, C. D., Jefferson, D. R., & Lapre, J. M. (2013). Warp speed: Executing Time Warp on 1,966,080 cores. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (pp. 327–336). Montrèal. doi:10.1145/2486092.2486134

Bayer, M. (2020). SQLAchemy-the database toolkit for Python. Retrieved from https://www.sqlalchemy.org/

Carothers, C. D., Bauer, D., & Pearce, S. (2000). ROSS: A high-performance, low memory, modular time warp system. *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, *62*, 53–60. doi:10.1109/PADS.2000.847144

Clune, M. I., Mosterman, P. J., & Cassandras, C. G. (2006). Discrete event and hybrid system simulation with simevents. In *Proceedings of the 8th international workshop on discrete event systems* (pp. 386–387).

Concannon, K. H., Hunter, K. I., & Tremble, J. M. (2003). Dynamic scheduling II: SIMUL8-planner simulation-based planning and scheduling. In *Proceedings of the 35th Conference on Winter simulation* (pp. 1488–1493). doi:10.1145/1030818.1031019

Fishman, G. S. (2013). *Discrete-event simulation: Modeling, programming, and analysis*. Springer Science & Business Media.

Fujimoto, R. M. (1990). Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulations* (Vol. 22, pp. 23–28). Retrieved from https://gdo149.llnl.gov/attachments/20776356/24674621.pdf

Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, *81*(25), 2340–2361. doi:10.1021/j100540a008

Goldberg, A. P., & Karr, J. R. (2020). WC-Sim: A multi-algorithmic simulator for whole-cell models. Retrieved from https://github.com/KarrLab/wc_sim

Goldberg, A. P., Szigeti, B., Chew, Y. H., Sekar, J. A., Roth, Y. D., & Karr, J. R. (2018). Emerging whole-cell modeling principles and methods. *Current Opinion in Biotechnology*, *51*, 97–102. doi:10.1016/j.copbio.2017.12.013

Hagberg, A., Swart, P., & S Chult, D. (2008). *Exploring network structure, dynamics, and function using networkx*. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. doi:10.1109/MCSE.2007.55

Jefferson, D. R. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems*, *7*(3), 404–425. doi:10.1145/3916.3988

Karr, J. R., Goldberg, A. P., & Chew, Y. H. (2020). WC-Lang: A language for composite, multi-algorithmic whole-cell models. Retrieved from https://github.com/KarrLab/wc_lang

Karr, J. R., Sanghvi, J. C., Macklin, D. N., Gutschow, M. V., Jacobs, J. M., Bolival Jr, B., Assad-Garcia, N., et al. (2012). A whole-cell computational model predicts phenotype from genotype. *Cell*, *150*(2), 389–401. doi:10.1016/j.cell.2012.05.044

Karr, J. R., Takahashi, K., & Funahashi, A. (2015). The principles of whole-cell modeling. *Current Opinion in Microbiology*, *27*, 18–24. doi:10.1016/j.mib.2015.06.004

Malengier, B., Kišon, P., Tocknell, J., Abert, C., Bruckner, F., & Bisotti, M.-A. (2018). ODES: A high level interface to ode and dae solvers. *Journal of Open Source Software*, *3*(22), 165.

Matloff, N. (2008). Introduction to discrete-event simulation and the SimPy language. Retrieved from https://web.cs.ucdavis.edu/~matloff/matloff/public_html/156/PLN/DESimIntro.pdf

McKinney, W., & others. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51–56). doi:10.25080/majora-92bf1922-00a

Mueller, W., Ruf, J., Hoffmann, D., Gerlach, J., Kropf, T., & Rosenstiehl, W. (2001). The simulation semantics of SystemC. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition* (pp. 64–70). doi:10.1109/DATE.2001.915002

Oliphant, T. E. (2015). *A guide to NumPy*. CreateSpace Independent Publishing Platform. ISBN: 978-1517300074

Orth, J. D., Thiele, I., & Palsson, B. Ø. (2010). What is flux balance analysis? *Nature Biotechnology*, *28*(3), 245–248. doi:10.1038/nbt.1614

Rice, S. V., Marjanski, A., Markowitz, H. M., & Bailey, S. M. (2005). The simscript iii programming language for modular object-oriented simulation. In *Proceedings of the winter simulation conference, 2005.* (pp. 10–pp). IEEE.

Schumacher, C., Leupers, R., Petras, D., & Hoffmann, A. (2010). ParSC: Synchronous parallel systemc simulation on multi-core host architectures. In *2010 ieee/acm/ifip international conference on hardware/software codesign and system synthesis (codes+ isss)* (pp. 241–246). IEEE.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., et al. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*(3), 261–272. doi:10.1038/s41592-019-0686-2

Zeigler, B. P. (1987). Hierarchical, modular discrete-event modelling in an object-oriented environment. *Simulation*, *49*(5), 219–230. doi:10.1177/003754978704900506