---

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

tags:

- dynamical modeling

- simulation

- discrete-event simulation

- object-oriented simulation

- parallel discrete-event simulation

- biochemical modeling

- whole-cell modeling

- Python

authors:

- name: Arthur P. Goldberg

orcid: 0000-0003-2772-1484

affiliation: "1"

- name: Jonathan R. Karr

orcid: 0000-0002-2605-5080

affiliation: "1"

affiliations:

- name: 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

index: 1

date: 20 July 2020

bibliography: paper.bib

---

# Summary

Many fields of science and engineering seek to understand how the dynamics of complex systems arise from interactions among their components. Models based on the physical interactions between components in these systems often represent the interactions as instantaneous events. Such models include studies of biochemical dynamics, computer network performance, wargame strategy, infectious disease epidemics, and other scientific, engineering and social science systems [@banks2005discrete]. Dynamical models that assume events are instantaneous are called \*discrete-event models\* and are simulated by the \*discrete-event simulation\* (DES) method [@fishman2013discrete].

To make it easier to construct and simulate complex discrete-event models, we developed DE-Sim ([github.com/KarrLab/de\_sim](https://github.com/KarrLab/de\_sim)), an open-source, Python-based DES tool. DE-Sim supports two main functions. First, DE-Sim users can define discrete-event models as object-oriented Python programs. And second, DE-Sim conveniently and efficiently simulates these models.

# The need for DE-Sim

DE-Sim simplifies the construction and simulation of discrete-event models used by studies that seek to understand the dynamical properties of complex systems which emerge from discrete, instantaneous interactions. An example problem asks “How might an epidemic spread?” To answer this question epidemic models which predict the variability of epidemics in small populations represent the transmission of disease as many random, instantaneous interactions in which an infected individual transmits the disease to a susceptible individual [@allen2017primer].

The complexity of large discrete-event models makes them challenging to construct. They can contain multiple types of components, and multiple types of interactions between components. 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), recommends that models represent components in the system being modeled as simulation objects, and that models represent interactions between components as event messages that schedule timed events executed by simulation objects [@zeigler1987hierarchical]. Simulation object types in DE-Sim models are built by simply sub-classing DE-Sim’s base simulation object class. Complex systems that contain multiple component types can be easily modeled in DE-Sim by creating multiple simulation object classes. And arbitrarily many instances of a component type are naturally represented by many instantiations of its simulation object type. Simulation object types defined in DE-Sim can exploit all the features of Python objects. For example, hierarchical refinement relationships among the components in a system being modeled can be mirrored by subclass relationships among the simulation object types that represent the components in a DE-Sim model. We find that DE-Sim’s use of OO programming to construct discrete-event models greatly simplifies and accelerates the process.

Recent advances in data collection have enabled many scientific fields to collect detailed data about the components of complex systems and their interactions. For example, the revolution in sequencing macromolecules has dramatically increased the acquisition of biochemical data. These data can parameterize discrete-event models of biochemical systems. As a Python tool, DE-Sim can leverage Python's extensive suite of high-quality data science tools to easily manage and integrate large, heterogeneous, multidimensional data into dynamical models. For example, Python tools such as NumPy [@oliphant2006guide], pandas [@mckinney2010data], SciPy [@virtanen2020scipy], and SQLAlchemy [@bayer2020sqlalchemy] can be used by DE-Sim models to store and integrate model inputs, simplify analyses during simulation, and organize and save predictions for downstream analysis.

DE-Sim is designed for scientists and engineers who want to build and use quantitative, dynamical models to understand 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 software; it is easy to learn because it provides extensive tutorials, examples, and documentation; and it is thoroughly tested and reliable. We are already using DE-Sim to develop a multi-algorithmic simulator that simulates comprehensive models of biological cells which predict phenotype from genotype by capturing all of the biochemical activity in a cell.

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

Multiple DES tools already exist.

\autoref{fig:comparison} lists the most important simulation tools, selected by determining the tools most frequently cited at the simulation community's largest annual conference, the Winter Simulation Conference.

These tools all provide a programming environment for developing DES models, a simulator that simulates models, and methods for reviewing simulation predictions.

All of the simulation tools in \autoref{fig:comparison} accept models written in code.

In addition, two tools provide a graphical interface for describing models, SIMUL8 [@concannon2003dynamic] and SimEvents [@clune2006discrete].

The commercial simulation tools all use proprietary modeling languages.

DE-Sim and three other tools, SystemC [@mueller2001simulation], SIMSCRIPT III [@rice2005simscript] and SimEvents support object-oriented descriptions of models [@zeigler1987hierarchical].

SimEvents obtains its OO modeling functionality from the OO features of MATLAB.

![\*\*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.

DE-Sim's combination of features makes it uniquely suitable for creating discrete-event models to study complex systems because it combines the power and convenience of OO modeling with the ability to leverage Python's extensive library of data science tools to build complex models from large datasets.

\label{fig:comparison}](comparison.pdf)

In contrast with SimPy [@matloff2008introduction], DE-Sim models are defined using object-oriented programs, whereas SimPy models must be defined with functions at a lower-level.

In addition, DE-Sim supports a uniform approach for scheduling events, whereas SimPy models that contain multiple processes must use two approaches to schedule events.

DE-Sim objects always schedule an event by sending an event message to the object that will execute the event.

However, SimPy processes schedule events for themselves by using a timeout call and Python's `yield` function, but schedule events for other processes by raising an interrupt exception.

DE-Sim is more accessible to scientific researchers than SystemC, because DE-Sim builds upon Python whereas SystemC uses C++, a lower-level language.

In addition, because SystemC is designed for modeling digital electronics, its concepts do not map well onto the systems that scientific researchers study.

The last three DES tools in \autoref{fig:comparison} are commercial software.

Two of them, SIMUL8 and SimEvents, specialize in modeling domains that do not contain scientific problems.

An important benefit of OO DES models is that individual simulation runs can be sped up by parallel execution on multiple cores.

Consider, for example, an OO DES model composed of objects that only interact with each other via event messages and do not access shared memory.

A simulation of this model could be sped up by distributing its objects across multiple cores and executing them in parallel while they are synchronized by a parallel DES simulator, such as Time Warp [@Jefferson1985, @carothers2000ross].

Parallel DES simulations can achieve substantial speedup, as Barnes et al. demonstrated by running the PHOLD benchmark [@fujimoto1990performance] on nearly 2 million cores [@Barnes2013].

By contrast, while independent SimPy simulations can be run in parallel, a single SimPy simulation cannot be parallelized [@muller2011running].

![\*\*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 with three simulation objects. The timeline (grey line) for each object shows its events (grey dots). Each event in PHOLD schedules another event, as illustrated by event messages (arrows) sent from the earlier event to the event being scheduled. The curved blue arrows indicate an event scheduled by a simulation object for itself in the future, 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.

\label{fig:phold\_space\_time\_plot}](phold\_space\_time\_plot.pdf)

# Tutorial: Building and simulating models with DE-Sim

An OO DES model that uses DE-Sim can be defined in three steps:

1: Create event message types by subclassing `SimulationMessage`.

```python

class MessageSentToSelf(SimulationMessage):

"A message type with no attributes"

class MessageWithAttribute(SimulationMessage):

"An event message type with an attribute called 'value'"

attributes = ['value']

```

An event message class must be documented by a docstring, and may include attributes.

2: Define simulation application objects by subclassing `ApplicationSimulationObject`.

```python

class SimpleSimulationObject(ApplicationSimulationObject):

def \_\_init\_\_(self, name, delay):

self.delay = delay

super().\_\_init\_\_(name)

def init\_before\_run(self):

self.send\_event(self.delay, self, MessageSentToSelf())

def handle\_simulation\_event(self, event):

self.send\_event(self.delay, self, MessageSentToSelf())

event\_handlers = [(MessageSentToSelf, handle\_simulation\_event)]

# register the message types sent

messages\_sent = [MessageSentToSelf]

```

Each object in a simulation must have a unique `name`.

This example adds an instance attribute that provides the delay between events.

All `ApplicationSimulationObject`s also have a read-only attribute called `time` that always provides the current simulation time.

A simulation object may define an `init\_before\_run` method, which, if present, will be called by the simulator just before simulation begins, after all simulation objects have been added to the simulation engine.

A simulation object that needs to schedule initial events for itself or for other objects in the simulation should do so in `init\_before\_run`.

This method can also perform any other initialization.

A simulation must schedule at least one initial event to commence.

`ApplicationSimulationObject` provides the method

`send\_event(delay, receiving\_object, event\_message)` which schedules an event to occur `delay` time units in the future.

`event\_message` is an instance of a `SimulationMessage`, and may have attributes that contain data used by the event.

The event will be executed by an event handler in simulation object `receiving\_object`, which will receive a simulation event containing `event\_message` at its scheduled simulation time.

A simulation object must include at least one method that handles simulation events.

The simulator vectors incoming message types as directed by an `event\_handlers` attribute that associates each message type received by a simulation object with one of its methods.

In the example, when an event storing a `MessageSentToSelf` instance occurs at a `SimpleSimulationObject`, the method `handle\_simulation\_event` will be invoked with the event as an argument.

In this example, all simulation events are scheduled to be executed by the object that creates the event, but complex simulations usually contain multiple simulation objects which schedule events for each other.

3: Execute a simulation by creating a `SimulationEngine`, instantiating the application objects, sending their initial event messages, and running the simulation.

```python

# create a simulation engine

simulation\_engine = SimulationEngine()

# create a simulation object and add it to the simulation

simulation\_engine.add\_object(SimpleSimulationObject('object\_1', 6))

# initialize and run the simulation

simulation\_engine.initialize()

num\_events = simulation\_engine.run(100).num\_events

```

This runs a simulation for 100 time units, and obtains the number of events executed.

# Performance of DE-Sim

DE-Sim achieves good performance by using Python's `heapq` priority queue package to schedule events.

\autoref{fig:performance} shows the performance of DE-Sim simulating a model of a cyclic messaging network over range of network sizes.

![\*\*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 executions 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 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 source code for this model is available in the DE-Sim Git repository.

\label{fig:performance}](performance.pdf)

# Case study: a multi-algorithmic simulation tool for whole-cell modeling

We have used DE-Sim to develop WC-Sim [@goldberg2020wc\_sim], a multi-algorithmic simulation tool that simulates comprehensive models of the biochemical dynamics inside biological cells [@karr2015principles; @goldberg2018emerging; @karr2012whole].

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 build whole-cell models is to represent each cellular process at a suitable granularity and simulate each with the most appropriate algorithm. For example, slow biochemical processes, such as transcription, can be simulated with a discrete-event model of biochemical dynamics, the Stochastic Simulation Algorithm (SSA, @gillespie1977exact). Faster processes, such as signal transduction, can be simulated with ordinary differential equations (ODEs) while metabolism, another fast process, can be simulated with flux-balance analysis (FBA, @orth2010flux). Resolving the simulation of these different processes at the whole-cell level requires co-simulating SSA, ODE and FBA. However, tools for co-simulating these algorithms do not exist.

To enable whole-cell modeling, we have created WC-Sim, a whole-cell simulator that simulates whole-cell models described in the WC-Lang language [@karr2020wc\_lang].

We implemented WC-Sim by using DE-Sim to construct a simulation object class for each of the SSA, ODE, and FBA submodels.

DE-Sim event messages schedule the activities of each simulation object, while the advance of simulation time is used to coordinate the objects’ shared access to the counts of molecules that represent the shared state of the cell.

DE-Sim’s object-oriented modeling functionality made it easy to separately develop SSA, ODE, and FBA simulation objects and compose them into a multi-algorithmic simulator.

DE-Sim’s discrete-event framework provided the complete control needed to precisely synchronize the interactions between these objects.

And DE-Sim’s Python foundation enabled us to dramatically reduce the effort required to build WC-Sim by leveraging data-science tools including pandas, networkx, matplotlib, NumPy, and SciPy.

We anticipate that WC-Sim will enable researchers to conduct unprecedented simulation experiments in cellular biochemistry.

# Summary of DE-Sim’s key features

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

\* \*\*Object-oriented modeling:\*\* DE-Sim uses object-oriented Python programming to build models. This simplifies the construction of dynamic, discrete-event models of large, complex systems.

\* \*\*Access to Python’s data-science tools:\*\* Because DE-Sim uses Python to build models, researchers can easily use its high-level data science packages such as NumPy, pandas, and SciPy to integrate large, heterogeneous datasets in their models.

\* \*\*Simple simulation logging:\*\* DE-Sim supports easily configured, high performance Python logs which can log simulation data that can 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 an abstract class that creates periodic checkpoints.

In addition, DE-Sim automatically records configuration information such as a simulation’s run arguments and metadata such as the start time and duration of the simulation.

\* \*\*Powerful stop conditions:\*\* DE-Sim makes it easy to implement complex stop conditions. These are implemented as arbitrary Python functions that return true when the simulation state reaches the desired stop condition.

\* \*\*Space-time visualizations for analysis and debugging:\*\* DE-Sim can generate space-time visualizations of simulation trajectories (\autoref{fig:phold\_space\_time\_plot}). These diagrams help understand and debug models.

\* \*\*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.

\* \*\*Controlled, reproducible execution of simultaneous events:\*\* An OO discrete-event simulation may contain simultaneous events. A simulation object may receive multiple events at the same simulation time, 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.

# Availability of DE-Sim

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

\* Python package: [PyPI: de-sim](https://pypi.org/project/de-sim/)

\* Docker image: [DockerHub: karrlab/de\_sim](https://hub.docker.com/r/karrlab/de\_sim)

\* Examples, tutorials, and documentation: [docs.karrlab.org](https://docs.karrlab.org/de\_sim/)

\* Issue tracker: [GitHub: KarrLab/de\_sim](https://github.com/KarrLab/de\_sim/issues/)

\* Source code: [GitHub: KarrLab/de\_sim](https://github.com/KarrLab/de\_sim/)

\* Guide to contributing and code of conduct: [GitHub: KarrLab/de\_sim](https://github.com/KarrLab/de\_sim/)

\* Continuous integration: [CircleCI: gh/KarrLab/de\_sim](http://circleci.com/gh/KarrLab/de\_sim/)

DE-Sim requires [Python](https://www.python.org/) 3.7 or higher and [pip](https://pip.pypa.io/).

This article discusses version 0.0.6 of DE-Sim.

# Acknowledgements

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.

We thank Yin Hoon Chew for her helpful feedback on this paper.

# References