Implementing Event Graphs in Python using DESpy

# Event Graphs

Event Graphs are a particularly effective and expressive way to represent complex discrete event simulation (DES) models. An Event Graph model consists of defining (1) Parameters are values which do not change during any given replication; (2) State variables are values which may change during any given replication; and (3) Events, which correspond to instantaneous state transitions; and (4) Scheduling relationships between events. That is, when a given event occurs, specifying which events may be scheduled in the future.

Event Graphs identify the Events (state transitions) with nodes and each scheduling relationship between events as a directed edge.

The fundamental Event Graph construct is shown in Figure 1.



Figure 1. Fundamental Event Graph Construct

The meaning of Figure 1 is as follows. When event A occurs, then after its state transition is executed, if boolean condition (i) is true, then event B is scheduled to take place t time units in the future.

By convention, a special event called “Run” is always executed at the start of every replication. Its state transition consists of initializing every state variable and scheduling the first event(s). It is otherwise an ordinary event.

The simplest non-trivial Event Graph model is the Arrival Process, which consists of a single parameter {tA}, the sequence of interarrival times, a single state variable N, the number of times the Arrival event has occurred (initially 0), and the Arrival event itself, whose state transition simple increments N by 1. The Arrival Process Event Graph is shown in Figure 2.



Figure 2. Arrival Process Event Graph

# DESpy

DESpy is a package for implementing Event Graph models, and is freely available under the Apache 2.0 Open Source license. It provides an Event List for scheduling and executing events, a base class for defining Event Graph models, many of the standard probability distributions, and some classes for computing simple sample statistics from simulation experiments.

DESpy can be installed from the PyPi web site using pip; from the command prompt, simply enter:

pip install --upgrade DESpy

# Event Graph – DESpy Relationships

Every element of an Event Graph model corresponds directly to an element in DESpy. Thus, creating a Python implementation is less about *programming* as it is about *translating*.

|  |  |
| --- | --- |
| **Event Graph** | **DESpy** |
| Model | Subclass of SimEntityBase |
| Parameter | Instance variable with value passed to constructor |
| State Variable | Instance variable with argument *not* passed to constructor |
| Event | Instance method; names changed to lower-case in keeping with Python conventions |
| Scheduling Edge | Call to self.schedule() with first argument the name of the scheduled event |
| Boolean condition on scheduling edge | Use if test before calling self.schedule() |
| Delay on scheduling edge | Second argument of call to self.schedule() |
| Run event | reset() method initializes state variables; run() method schedules first event(s) |

Figure 3. Event Graph/DESpy Translation

# Example: The Arrival Process

The DESpy implementation of the Arrival Process in Figure 2 is shown in Figure 4 (with comments omitted for brevity).

**from** simkit.simkit **import** SimEntityBase *# (1)*  
**from** math **import** nan  
  
**class** ArrivalProcess(SimEntityBase): *# (2)* **def** \_\_init\_\_(self, generator): *# (3)* SimEntityBase.\_\_init\_\_(self)  
 self.generator = generator  
 self.number\_arrivals = nan  
  
 **def** reset(self): *# (4)*

SimEntityBase.reset(self)self.number\_arrivals = 0  
  
 **def** run(self): *# (5)* self.notify\_state\_change(**"number\_arrivals"**, self.number\_arrivals)  
  
 self.schedule(**'arrival'**, self.generator.generate())  
  
 **def** arrival(self): *# (6)* self.number\_arrivals += 1  
 self.notify\_state\_change(**"number\_arrivals"**, self.number\_arrivals)  
  
 self.schedule(**'arrival'**, self.generator.generate())

Figure 4. Arrival Process in DESpy

The elements of the ArrivalProcess class in Figure 4 are as follows:

1. Import SimEntityBase class so it can be used
2. Define ArrivalProcess as a subclass of SimEntityBase
3. In the constructor, first call the super class constructor. Then define generator as a parameter, set to the passed-in argument; then define number\_arrivals as a state variable. Scalar state variables should ne initialized here to nan by convention – state variables actual initial values should be set in the reset() method.
4. The reset() method always invokes the super class reset() method, then initializes all state variables (in this case just number\_arrivals) to their respective initial values as defined in the corresponding Event Graph model.
5. The run() method first notifies for the initial value of number\_arrivals, then schedules the first arrival event with a delay generated by the generator parameter. The notify\_state\_change() call should be made following every state transition so any “listeners” can be notified. A “state change listener” is typically one that computes appropriate statistics for that state. Another commonly used state change listener simply prints the state and its new value to the console. This is useful for debugging purposes.
6. The arrival() method performs the state transition, in this case incrementing number\_arrivals by 1, notifying the state change, and scheduling the next arrival event.

# Testing the Implementation

It is recommended that separate Python files be used to implement a model. A class like the ArrivalProcess above can be used and re-used in many different configurations. It is important to verify that an implementation accurately reflects its corresponding Event Graph model. A useful way to do that is to execute short runs in “verbose” mode with the state transitions being output.

The following are useful steps in testing:

1. Instantiate any RandomVariate parameters
2. Instantiate the component with all parameters; if not a stand-alone component, instantiate any other components to run the model.
3. Instantiate a SimpleStateChangeDumper and add it as a state change listener to the component
4. Set EventList.verbose to True
5. Set EventList.stop\_at\_time() or EventList.stop\_on\_event() for relatively short runs.
6. Invoke EventList.reset() and EventList.start\_simulation() to run the model

An example of a small test run in verbose mode is shown in Figure 5.

**from** simkit.rand **import** RandomVariate *# (1)***from** simkit.base **import** EventList *# (2)***from** simkit.examples.arrivalprocess **import** ArrivalProcess *# (3)***from** simkit.simutil **import** SimpleStateChangeDumper *# (4)*interarrival\_time\_generator = RandomVariate.instance(**'Exponential'**, mean=1.7) *#(5)*arrival\_process = ArrivalProcess(interarrival\_time\_generator) *#(6)*print(arrival\_process.describe()) *# (7)*simple\_state\_change\_dumper = SimpleStateChangeDumper() *# (8)*arrival\_process.add\_state\_change\_listener(simple\_state\_change\_dumper) *# (9)*EventList.verbose = **True** *# (10)*EventList.stop\_at\_time(10.0) *# (11)*EventList.reset() *# (12)*EventList.start\_simulation() *# (13)*

print(**"Simulation ended at time {time:.2f}"**.format(time=EventList.simtime)) *# (14)*print(**"There have been {number:,d} arrivals"**.format(number=arrival\_process.number\_arrivals)) *# (15)*

Figure . Small Test Run in Verbose Mode

1. Import to be able to declare RandomVariate instances
2. Import to have access to EventList for execution
3. The ArrivalProcess example in DESpy
4. The SimpleStateChangeDumper outputs every state change for models it “listens” to
5. Create the interarrival time RandomVariate object
6. Create the ArrivalProcess object
7. The describe() method displays the parameters and state variables.
8. Create the SimpleStateChangeDumper object
9. The SimpleStateChangeDumper object “listens” to state changes of the ArrivalProcess object
10. The EventList “verbose” mode will display the status of the event list after every event is executed
11. Set the ending time of the simulation (very short in this case)
12. The reset() method initializes the model by:
13. Setting simtime to 0.0
14. Clearing the event list (important for multiple replications)
15. Ensuring state variables are initialized by invoking the reset() method on all model objects
16. Scheduling the Run event on the event list (using the run() method) at time 0.0
17. Scheduling the Stop event at the stop\_time (if stop\_at\_time is the ending criterion)
18. The start\_simulation() method executes the model by executing the event list logic
19. This verifies that the run ended at the correct time (EventList.simtime always gives the current value of simulated time)
20. This outputs the number of arrivals that have occurred.

The output from this run is shown in Figure 6.

ArrivalProcess

id = 1

persistent = True

generator = Exponential (1.700)

number\_arrivals = nan

Starting Simulation...

\*\*\* Event List \*\*\*

0.0000 run <ArrivalProcess.1>

0.0000 run <Stopper.0>

ArrivalProcess.1> number\_arrivals: 0

CurrentEvent: 0.0000 run <ArrivalProcess.1> [1]

\*\*\* Event List \*\*\*

0.0000 run <Stopper.0>

1.4885 arrival <ArrivalProcess.1>

CurrentEvent: 0.0000 run <Stopper.0> [2]

\*\*\* Event List \*\*\*

1.4885 arrival <ArrivalProcess.1>

10.0000 stop <Stopper.0>

ArrivalProcess.1> number\_arrivals: 1

CurrentEvent: 1.4885 arrival <ArrivalProcess.1> [1]

\*\*\* Event List \*\*\*

9.2888 arrival <ArrivalProcess.1>

10.0000 stop <Stopper.0>

ArrivalProcess.1> number\_arrivals: 2

CurrentEvent: 9.2888 arrival <ArrivalProcess.1> [2]

\*\*\* Event List \*\*\*

9.6154 arrival <ArrivalProcess.1>

10.0000 stop <Stopper.0>

ArrivalProcess.1> number\_arrivals: 3

CurrentEvent: 9.6154 arrival <ArrivalProcess.1> [3]

\*\*\* Event List \*\*\*

10.0000 stop <Stopper.0>

11.6698 arrival <ArrivalProcess.1>

CurrentEvent: 10.0000 stop <Stopper.0> [1]

\*\*\* Event List \*\*\*

<Empty>

Simulation ended at time 10.00

There have been 3 arrivals

Figure . Output of Short Verbose Run

Note that there are two run events at time 0.0: one for the ArrivalProcess instance you created and a second for a “Stopper” object. Internally, the EventList creates this object and schedules its stop event, which has the effect of clearing the event list, thereby ending the simulation run.