Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

SIMULUS: EASY BREEZY SIMULATION IN PYTHON

Jason Liu

School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA

ABSTRACT

This paper introduces Simulus, a full-fledged open-source discrete-event simulator, supporting both event-driven and process-oriented simulation world-views. Simulus is implemented in Python and aspires to be a part of the Python's ecosystem supporting scientific computing. Simulus also provides several advanced modeling constructs to ease common simulation tasks (e.g., complex queuing models, inter-process synchronizations, and message-passing communications). Simulus also provides organic support for simultaneously running a time-synchronized group of simulators, either sequentially or in parallel, thereby allowing composable simulation of individual simulators handling different aspects of a target system, and enabling large-scale simulation running on parallel computers. This paper describes the salient features of Simulus and examines its major design decisions.

1 INTRODUCTION

There exist many discrete-event simulators, including, for example, AnyLogic (Borshchev 2014), Arena (Rossetti 2015), ExtendSim (Krahl 2008), SystemC (Panda 2001), and OMNet++ (Varga and Hornig 2008), to name just a few. Some of the simulators are available commercially and others in the public domain. They come with various design features and software support, such as the graphical user interface with drag-and-drop capabilities. Some of the simulators are designed as general-purpose simulation software and others primarily for modeling specific domains, including manufacturing, logistics, military, health care, computer systems, VLSI, communication networks, and so on. All in all, discrete-event simulation is a simple concept: it models the world as a sequence of events that take place at discrete points in time. A discrete-event simulator processes the events in timestamp order. To do that, the simulator keeps an event list, which is a data structure to store all future events (normally a priority queue to facilitate fast insertion and deletion operations), and a simulation clock, which gets advanced (by leaps and bounds) in accordance with the time of the events being processed.

Historically, discrete-event simulators were designed with special programming languages. Well-known examples are GPSS (Ståhl et al. 2011) and Simula (Nygaard and Dahl 1978). Modern simulators are commonly implemented in general-purpose programming languages, such as csim (Schwetman 1986) in C, OMNet++ (Varga and Hornig 2008) in C++, J-Sim (J-Sim Developers 2005) and SimJava (Page et al. 1997) in Java, Sim.JS (Maneesh Varshney 2011) in JavaScript, and SimX (Thulasidasan et al. 2014) and SimPy (Team SimPy 2007) in Python. Some simulators were implemented in multiple programming languages. For example, the Scalable Simulation Framework (SSF Research Network 2002) provides a reference API in both Java and C++. Simian (Santhi et al. 2015) has been realized in several scripting languages with just-in-time compilation, including Python, LUA, and Javascript.

Python is an interpreted, general-purpose programming language, created by Guido van Rossum in 1991. The language is dynamic typed and supports many advanced and extensible programming features, including multiple programming paradigms (such as object-oriented programming, functional programming,

and meta-programming), garbage collection, powerful standard library, and extensive modules and tools. Over the years, Python developers have evolved into a highly vibrant community. Python itself has become an ecosystem that facilitates fast software development with a large user base. According to Module Counts (Erik DeBill 2020), the Python Package Index (PyPI), which is an official repository for third-party Python software, now contains over 220,000 packages (by February 2020). Python supports many areas, especially scientific computing, data analytics, web, machine learning, and data visualization. Many packages and modules, such as numpy (Oliphant 2006), pandas (McKinney et al. 2011), matplotlib (Hunter 2007), scipy (Jones et al. 2001), scikit-learn (Pedregosa et al. 2011), pytorch (Paszke et al. 2019), tensorflow (Abadi et al. 2016), are easily accessible and widely popular among users.

Despite Python's increasing popularity, there are only a few full-fledged discrete-event simulators written for Python. SimPy (Team SimPy 2007) is a well-known example. SimPy was originally created by Klaus Müller and Tony Vignaux, and was later extended and maintained by Ontje Lünsdorf and Stefan Scherfke. SimPy supports process-oriented simulation, and as such, can model the world as consisted of one or more simulation processes, each being an independent thread of execution. SimPy implements the simulation processes as generator functions. A generator function in Python is a function that contains one or more `yield` statements, which can suspend (and later resume) the execution of the function. Python provides native support for generator functions; thus, they are convenient and quite efficient. However, to use them for simulation processes, the SimPy users must explicitly use the `yield` keyword in places for potentially suspending the processes and for invoking other generator functions as subroutines. Doing so can significantly obfuscate the control flow of the simulation model and consequently may result in complications for maintaining and debugging the code. SimX is a Python library designed specifically for parallel discrete-event simulation (Thulasidasan et al. 2014). The core of SimX, including event management and parallel synchronization, is written in C++ to provide the high performance necessary for running large-scale models. Simian (Santhi et al. 2015) is another open-source process-oriented discrete-event simulator written in Python. It is also a parallel simulator capable of running on parallel platforms. Simian has also been implemented with LUA and JavaScript in addition to Python. This can be largely attributed to Simian's minimalistic design. Simian has a very simple and intuitive application programming interface (API). The entire Python implementation contains only around 500 lines of code. However simple and intuitive, the simplicity also entails that Simian lacks high-level modeling support. It would require significant effort and expertise for building sophisticated models.

This paper introduces Simulus, a new full-fledged open-source discrete-event simulator, designed and implemented specifically for Python. Simulus aspires to be a part of the Python's ecosystem supporting scientific computing. We design the simulator in an attempt to maximize its usability and model expressiveness, and do so without significantly sacrificing performance. Achieving maximum usability and expressiveness is the first priority when designing the simulator, as it is considered a major means to reducing the development time and model complexity for most simulation tasks. More specifically, we adhere to the following design principles: *1)* Make use and take advantage of expressive constructs and facilities specific to the Python programming language; *2)* Support both event-driven and process-oriented simulation world-views; for the latter, support efficient light-weight simulation processes without exposing complexity; *3)* Provide advanced modeling constructs for inter-process synchronization and communication to ease most common simulation tasks (such as queuing models, producer-consumer synchronization, and conditional blocking); *4)* Provide basic support for interactive simulation and for model/data visualization; *5)* Support simultaneous execution of multiple simulation models, and thereby allow composable simulation of individual models handling different aspects of a target system; and *6)* Support parallel discrete-event simulation of large-scale models on shared-memory and distributed-memory platforms.

2 RELATED WORK

Process-oriented simulation has a long history that can be traced all the way back to the early ages of computer simulation where special simulation languages were used to describe discrete-event systems,

including notably Simula (Nygaard and Dahl 1978), GPSS (Ståhl et al. 2011), ASPOL (MacDougall and McAlpine 1973), and several others (see Nance 1996 for a good overview).

CSIM (Schwetman 1986) is a process-oriented simulator embedded in C and later in C++. CSIM models a system as a collection of high-level structures and interacting processes. The model maintains simulation time for the processes as they visit the high-level structures competing for the resources and when they execute hold statements. CSIM includes several high-level structures, such as "facilities" to represent servers reserved or used by processes, "storages" to represent resources that can be partially allocated to processes, "events" for synchronizing process activities, and "mailboxes" for inter-process communications. In the same fashion, we implemented the simulation processes and some of high-level modeling constructs for Python. The simulation processes and modeling constructs are natural (and therefore much simplified) in Python using coroutines and object-oriented design of the language itself.

SimPy (Team SimPy 2007) is a full-fledged process-based discrete-event simulator designed specifically for Python. SimPy uses Python's generator functions to implement simulation processes. Using generator functions has several drawbacks, as we discuss earlier. The bottom line is that users must understand that the processes are generators (rather than normal functions); they need to explicitly use the `yield` keyword wherever the processes can be interrupted and suspended from execution (such as waiting for certain events to happen, or invoking a subroutine that could be suspended). This creates extra burden for the users to be mindful of the simulator design. In contrast, Simulus uses coroutines from the greenlet package (Armin Rigo and Christian Tismer 2011) to implement the simulation processes. In this case, the user can simply treat the simulation processes as normal functions. SimPy also invented the concept of a simulation "environment", which manages the simulation time, processes events, and coordinates simulation processes. In Simulus, these simulation environments are simply called "simulators". Supporting multiple simulators simultaneously in execution allows Simulus to be used interactively and also enables Simulus to run in parallel, as we discuss in later sections. Similar to CSIM (and early simulation languages), SimPy includes several high-level modeling constructs: "resources" as servers used by processes, "stores" and "containers" for production and consumption of objects and quantities. Simulus provides similar high-level modeling constructs for use by simulation processes.

Simian (Santhi et al. 2015) is an open-source, process-oriented parallel discrete-event simulator. Simian has independent implementations in Python, Lua, and JavaScript. Simian has a minimalistic design with a simple API that contains only three main modules: simulation engine, entity, and process. A simulation engine is a logical process (with an event list and a simulation clock), which is mainly responsible for synchronizing with other logical processes for parallel simulation. An entity is a modeling unit that contains the simulation state (such as a compute node or an airport) and can schedule events with other entities. A process is an independent thread of execution on the entities. Simian takes advantage of just-in-time (JIT) compilation for interpreted languages. For certain models, it was demonstrated that Simian can outperform the C/C++ based simulation engine. The simulation engine in Simian is similar to our "simulator" design. However, Simulus provides a much richer API for synchronizing the simulators for parallel simulation (see Section 5). Simulus does not have "entities" and "processes" as we intentionally hide such complexities in Simulus. Event handlers and processes in Simulus are both treated as normal Python functions used almost indiscriminately as we try to maximize the usability. Simian also uses greenlet (Armin Rigo and Christian Tismer 2011) to implement the simulation processes. Simulus borrows the same idea.

Python is an interpreted language. PyPy (Rigo and Pedroni 2006) is a replacement of CPython (the default Python interpreter) with a just-in-time compiler. It is reported that, on average, PyPy is 4.4 times faster than CPython (The PyPy Team 2020). For this paper, we performed very limited performance studies on Simulus. With simple models, we have observed significant speedup using PyPy. We plan to report Simulus performance in future work.

There are many parallel simulators created over the years. Some of the prominent ones include GTW (Das et al. 1994), DaSSF (Liu et al. 1999), ROSS (Carothers et al. 2000), μsic (Perumalla 2005), MiniSSF (Rong et al. 2014), and Charades (Mikida et al. 2016). To achieve best performance, most parallel discrete-event

simulators are written in compiled languages, predominantly in C or C++. Scalable Simulation Framework (SSF) is a public-domain standard for parallel discrete-event simulation (SSF Research Network 2002). SSF specifies both C++ and Java bindings and has been used extensively to simulate large-scale networks. Both DaSSF and MiniSSF are C++ implementations of the SSF specification. A simulation process in DaSSF and MiniSSF is implemented a coroutine, the execution of which can be interrupted using sleep and wait statements. Both DaSSF and MiniSSF require user annotation to identify potential suspension points in functions. Simulus, with the help of greenlet, removes all the complexities involved with annotating the coroutines. As such, a simulation process is syntactically indistinguishable from a normal function. For parallel simulation synchronization, Simulus uses the YAWNS protocol, a window-based conservative algorithm (Nicol 1993). As with Simian (and as opposed to SimX), the synchronization protocol is implemented directly in Python using Python's MPI support (Lisandro Dalcin 2019).

3 BASIC APPLICATION PROGRAMMING INTERFACE

3.1 Simulators

To use Simulus, one needs to import the `simulus` package and create a simulator using its `simulator()` function. A simulator maintains an event list on which all events are stored in a priority queue according to their timestamps. A simulation clock keeps the current simulation time, which gets advanced while executing the events on the event list in timestamp order. Simulus can simultaneously run multiple simulators, each maintaining its own event list and simulation clock. If multiple simulators are created, they run independently, by default. That is, the current time at one simulator bears no relation to the current time of another simulator. In doing so, a user can create simulators on the fly in an interactive environment, such as the Jupyter Notebook (Kluyver et al. 2016). Simulators can also be time synchronized as a group (which we discuss in Section 5).

Each simulator processes events when we call the `run()` function. We can specify either the `offset` or the `until` argument. The `offset` argument specifies a relative time from the simulator's current simulation time, while the `until` argument specifies an absolute simulation time. When the `run()` function returns, the simulator is expected to have processed events on its event list with timestamps smaller than the designated time and its simulation clock be advanced to the designated time. The user can also step through simulation by processing events one at a time using the `step()` function.

The simulator's event list uses a priority queue dictionary, which is a data structure implemented to combine a binary heap and a dictionary (Nezar Abdennur 2013). The binary heap supports $O(1)$ access to the top-priority event (with the earliest timestamp), and $O(\log n)$ for inserting and removing an event. A dictionary is used to map the dictionary key to the position index of the event in the binary heap. This index is maintained as the heap is updated (from the insertion and deletion operations). With the dictionary, which allows $O(1)$ lookup of an event's position index, one can either cancel an event (by effectively removing the event from the priority queue) and reschedule an event (by changing the timestamp as the priority key of the event) in $O(\log n)$ time.

3.2 Direct-Event Scheduling

Simulus supports modeling using an event-oriented approach via scheduling and processing events. This is called *direct-event scheduling* in Simulus. It also supports modeling using a process-oriented approach via creating simulation processes and having them execute independently and synchronize with one another. We call this approach *process scheduling*. One can combine both direct-event scheduling and process scheduling in the same simulation models. We describe the direct-event scheduling approach in this section and the process scheduling approach in the next section.

In Simulus, a simulation event is simply a function to be invoked at a designated (simulation) time. The function is known as the event handler. We use the `sched()` function of the simulator to schedule a function invocation, by passing the name of the function and either the `until` or the `offset` argument.

Both `until` and `offset` are keyword arguments. In Simulus, an event handler can be an arbitrary Python function or class method, and we can pass arbitrary arguments to the event handler by placing the arguments right after the name of function when we call `sched()`. Simulus takes advantage of Python's ability to include both positional arguments and keyword arguments. In Python, the single-asterisk form of `*args` can be used to take a non-keyword variable-length list of arguments passed to a function. The double asterisk form of `**kwargs` can be used to take a keyword, variable-length dictionary of arguments passed to a function. The `sched()` function first filters out the recognized keyword arguments for itself (such as `until` and `offset`) before scheduling an event at the designated time on the simulator's event list. An event is a data structure that contains a reference to the intended event handler along with the function arguments, including all positional arguments and all unfiltered keyword arguments. Once the event is processed, the simulator invokes the the event handler and passes to it all the arguments stored in the event data structure.

The event data structure in Simulus is an opaque object to the user. The `sched()` function returns a reference to the event object which has been scheduled onto the simulator's event list. The user can use the reference to either cancel or reschedule the event (using `cancel()` or `resched()` functions respectively), but is not expected to directly access the variables and methods within the event object.

3.3 Process Scheduling

Another way to model the world using Simulus is to use simulation processes. Conceptually, a simulation process is just a thread of control, similar to running a sequential program that constitutes a sequence of instructions, including if-else branches, while-loops, and function calls. One creates a simulation process using the `process()` function of the simulator and pass to it the name of the starting function of the simulation process and either the `until` or `offset` argument to indicate the simulation time at which the process should start running. As with the event handler for the `sched()` function, the starting function can be any arbitrary Python function or class method, and we can pass arguments to the starting function—either as positional arguments, or keyword arguments, or both—by placing the arguments right after the name of function when we call `process()`.

The simulation process starts to run at the designated time from the beginning of the starting function. When the control reaches the end of the function, the process terminates. During its execution, a simulation process can be suspended, either by sleeping for some simulation time to pass, or when requesting for resources that are currently unavailable. A suspended simulation process can resume execution after the specified duration has passed or when the resource blocking condition has been removed.

For sleeping, one can call the `sleep()` function of the simulator. The function takes one argument: either an `offset` argument, which specifies the amount of time the simulation process needs to be put on hold, or an `until` argument, which specifies the absolute simulation time at which the process will resume execution.

A simulation process can request for resources that may potentially block the process from execution due to unavailability. Processes often need to interact with one another in order to accomplish tasks. For example, the agents in a multi-agent system may be modeled as simulation processes; they communicate and synchronize with one another as they compete for resources. Simulus provides the necessary facilities for managing and synchronizing the processes, as we describe in the next section.

As with Simian (Santhi et al. 2015), Simulus uses Python's greenlet package (Armin Rigo and Christian Tismer 2011) for simulation processes as coroutines (Conway 1963). Greenlet is implemented as a C-extension module that can run with unmodified Python interpreter. According to Knuth (1997), Section 1.4.2: Coroutines, pp. 193—200, the concept of coroutines is a generalization of subroutines. A subroutine is executed from start to finish without holding the state between invocations. A coroutine, on the contrary, holds the state between invocations (including the function stack and variables). As such, a coroutine can "yield" to other coroutines in the middle of its execution; the control may later return to the same point in the original coroutine at which the other coroutines were invoked. The greenlet implementation of the

coroutines is efficient without any implicit scheduling of the coroutines, and thus can be incorporated with the event processing loop of the discrete-event simulator to handle a large number of simulation processes.

In Python, one can also use generators to implement coroutines. This is the approach taken by SimPy (Team SimPy 2007). A generator in Python can be defined as a normal function with one or more `yield` statements. While a `return` statement terminates a function entirely, a `yield` statement suspends the function by saving its state so that it can later continue its execution on successive invocations. There are two important differences between using greenlets and generators. One difference is that a greenlet can call nested functions and the nested functions can also yield values. The other difference is a greenlet does not require the `yield` keyword. Consequently, a simulation process in Simulus is simply a normal Python function. The user is not tangled with the subtle differences with respect to running and managing the simulation processes. Handling simulation processes in Simulus is thus intuitive and straightforward as with using normal Python functions.

4 SYNCHRONIZATION MECHANISMS

Simulus provides a rich set of mechanisms to support inter-process communication and synchronization. In this section, we first discuss the basic synchronization primitives (traps and semaphores), then go on to describe the generalized concept of "trappables" for implementing conditional waits, and finally introduce some advanced simulation constructs for managing complex and yet common simulation scenarios.

4.1 Traps and Semaphores

Traps are one-time signaling mechanisms for synchronizing and communicating between simulation processes. A trap has three states. It's "unset" when tee trap is first created. It's "set" when one or more simulation processes are waiting for the trap to be triggered. A trap turns to be "sprung" when the trap is triggered by a simulation process and all waiting processes will be unblocked. The life cycle of a trap is as follows. A trap starts with the unset state when it's created. When a simulation process waits for a trap, the trap is set, at which state more processes may come and wait on the same trap, and the trap would remain to be in the same set state. When a simulation process triggers the trap and if the trap is in the set state, all processes waiting on the trap will be unblocked and resume execution. The trap will then be transitioned into the sprung state. When a process triggers the trap which is in the unset state, the trap will just be transitioned to the sprung state (since there are no processes waiting on the trap). If a trap has sprung, further waiting on the trap will be considered as an no-op; that is, the processes trying to wait on a sprung trap will have not effect; the process will not be suspended.

Semaphores are multi-use signaling mechanisms for inter-process communication and synchronization. A semaphore implements what is commonly called a "counting semaphore". Initially, a semaphore can have a nonnegative integer count, which indicates the number of available resources. A simulation process can atomically increment the semaphore count to represent resources being added or returned to the pool (using the `signal()` method). Similarly, a simulation process can atomically decrement the semaphore count to represent resources being removed from the pool (using the `wait()` method). When the semaphore count is zero, it means that there are no available resources. In that case, a process attempting to decrement the semaphore count will be blocked until more resources are added back to the pool.

A semaphore is different than a trap: a trap is a one-time signaling mechanism while multiple processes can wait on a trap. Once a process triggers the trap, all waiting processes will be unblocked at once. Moreover, a trap cannot be reused. Once a trap is sprung, subsequent waits will not block the processes; a trap cannot be triggered multiple times. In comparison, a semaphore is a multi-use signaling mechanism. Each time a process waits on a semaphore, the semaphore value will be decremented. If the count becomes negative, the process will be blocked. Each time a process signals a semaphore, the semaphore count will be incremented. If there are blocked processes, one of these processes will be unblocked. Processes can use the same semaphore continuously throughout the simulation. If multiple processes are waiting on a

```
1  class Barrier(object):
2      def __init__(self, sim, total_procs):
3          self.sim = sim
4          self.total_procs = total_procs
5          self.trap = self.sim.trap()
6          self.num = 0
7
8      def barrier(self):
9          self.num += 1
10         if self.num < self.total_procs:
11             self.trap.wait()
12         else:
13             self.trap.trigger()
14             self.trap = self.sim.trap()
15             self.num = 0
```

Figure 1: Barrier implementation using traps.

semaphore, the order in which the processes are unblocked may be important. The queuing discipline can be specified when the semaphore is created.

Both traps and semaphores can be used to implement other (more complicated) synchronization mechanisms. Fig. 1 shows an example of using traps to implement a barrier for the processes. A trap is created when the barrier is initialized (line 5). Each process calls the `barrier()` method upon entering the barrier. All processes except for the last one reaching the barrier will simply invoke the trap's `wait()` method (line 11). The last process entering the barrier will trigger the trap and therefore release all the waiting processes previously arrived at the barrier (line 13). After that, the last process also creates a new trap and resets the counter so that the barrier can be reused for the next time (lines 14 and 15).

4.2 Trappables and Conditional Waits

Both traps and semaphores are called *trappables*. A scheduled event is a trappable. A process is also a trappable. Simulus provides a powerful function, called `wait()`, to block the calling process until any or all of the given trappables are triggered, or until a pre-specified amount of time has elapsed. We say a trappable is "triggered" when the wait condition of the trappable has been satisfied. If it's a trap, it means the trap has been triggered by another process. If it's a semaphore, it means a wait on the semaphore has finished (when another process signals the semaphore). If it's an event, it means the event has happened (and the event handler has been invoked). If it's a process, it means the process has terminated. Simulus also provides high-level modeling constructs which contain trappables for simulation processes to wait on. We describe the details in the next section.

The simulator's `wait()` function expects the first argument to be either a trappable object or a list (or tuple) of trappables. If it takes more than one trappables, the calling process will be blocked until either one of the trappables, or all of the trappables are triggered. The behavior depends on the keyword argument `method`. If it's `any`, the wait condition is satisfied as soon as one of the trappables is triggered; if it's `all` (the default), the process needs to wait until all trappables are triggered. Other user-defined predicates can also be used here, for example, when two or more of the trappables are triggered. One can optionally provide an `offset` or `until` argument to the `wait()` function. They put a time limit on the wait. If ignored, there will be no time limit on the conditional wait. The `wait()` function returns a tuple with two elements: the first is itself a tuple indicates whether individual trappables have been triggered or not; and the second element is boolean to indicate whether timeout happens.

The `wait()` function described above is the most powerful statement that one can use to synchronize simulation processes. The function can be used anywhere inside a simulation process (in the starting function or in functions invoked directly or indirectly by the starting function of the simulation process).

4.3 High-Level Modeling Constructs

Both traps and semaphores are low-level signaling mechanisms for processes. Simulus provides several high-level modeling constructs designed specifically to ease the efforts in undertaking common modeling tasks. Although these high-level modeling constructs can be built using traps and semaphores, their implementation can be cumbersome and error-prone. Simulus provides these constructs because of their well-defined interface and expected popularity in many usage cases.

Resource. A `resource` is basically a (single-server or multi-server) queue, which allows only a limited number of simulation processes to be serviced at any given time. Upon arrival, a process acquires a server at the resource. If there is an available server, the process will immediately gain access to the resource and will occupy the resource for as long as the service is required. If there is no available server, the process will be put on hold and placed in a waiting queue. When another process releases a server, one of the waiting processes will be unblocked and thereby gain access to it. To use the resource, a simulation process is expected to first call the method `acquire()` to gain access to a server. This is potentially a blocking call: the process will be blocked if the server is unavailable at the time of the call. The method only returns if a server has been "acquired" by the process. The process can occupy the resource for as long as needed (for example, the process can sleep for some duration). After finished using the resource, the same process is expected to call the method `release()` to free the server, so that other waiting processes can have a chance to gain access to the resource.

Store and Bucket. Both `store` and `bucket` can be used for synchronizing producer and consumer processes. A store is a facility for storing countable objects, such as jobs in a queue, packets in a network router, and I/O requests on a storage device. A bucket is used for storing uncountable quantities or volumes, such as gas in a tank, water in a reservoir, and battery power in a mobile device. A producer process calls the `put()` method to deposit objects into the store or quantities into the bucket. The current storage level will increase accordingly as a result. However, if a producer process tries to put more objects or quantities into the store or bucket than the maximum capacity, the producer process will be blocked and it will remain blocked until the storage level decreases when some other processes get objects or quantities from the store or bucket. Similarly, a consumer process calls the `get()` method to retrieve objects from the store or quantities from the bucket. The current storage level will decrease accordingly as a result. If a consumer process attempts to retrieve more objects or quantities than the current storage level, the consumer process will be blocked, and it will remain blocked until the current storage level goes above the requested amount (when some other processes put objects or quantities into the store or bucket). A store can actually be used for storing real Python objects. A producer process can put one or more Python objects into the store, which can then be retrieved in a first-in-first-out fashion by another consumer process.

Mailbox A mailbox is a facility designed specifically for message passing between simulation processes or functions. A mailbox consists of one or more compartments. A sender can send a message to a designated compartment of a mailbox with a delay for the message transportation. The message will be delivered to the mailbox at the expected time. Messages arriving at a mailbox will be stored in the designated compartments until they are retrieved and removed by some processes or functions. In Simulus, a message takes a broader meaning and can be any Python objects. A process can send multiple messages to a mailbox, each potentially with a different delay. One or more simulation processes can call the mailbox's `recv()` method trying to receive the messages at a specific compartment. If there are already messages stored at the mailbox compartment, the process will retrieve the messages (by removing them) and return them (as a list) without delay. If there are no messages in the mailbox compartment, the process will be suspended. Once a new message arrives, all waiting processes at the compartment will be unblocked to retrieve the message. The `recv()` method can either retrieve all messages from the mailbox compartment at once, or only the first message by option. When multiple processes are waiting to receive a message, Simulus does not specify which one shall wake up first to retrieve the message. There are no specific tie-breaking rules for simultaneous events in Simulus.

Conditional waits and data collection A resource is trappable, which means one can apply conditional wait on resources using the simulator's `wait()` function. A resource is triggered if the process acquires a server and gains access to the resource. However, both store and bucket are not trappables. There are two reasons. One is that a store or a bucket has two methods, `get()` and `put()`, both of which could block a process. One needs to explicitly specify which of the two the process should be waiting on. The other reason is that both methods can take arguments, for example, the number of objects for store or the amount of quantities for bucket. To allow conditional wait, the store and bucket provide two methods, `getter()` and `putter()`, which return the corresponding trappable object to be used by the `wait()` function. For the same reason, mailbox is also not a trappable. A mailbox can have multiple compartments; the `recv()` method can be blocked on receiving messages from any of the compartments. There are also arguments expected for the `recv()` method. To allow conditional wait, the mailbox provides the `receiver()` method, which returns the corresponding trappable object for receiving messages from a designated mailbox compartment. The high-level modeling constructs also provide flexible data collection capabilities for users to gather statistics on resources, stores, buckets, and mailboxes.

5 PARALLEL SIMULATION SUPPORT

5.1 Synchronized Group

A synchronized group is a group of simulators whose simulation clocks advance synchronously. That is, although each simulator still processes events on its own event list according to the timestamp ordering, the simulators in a synchronized group advance their simulation clock in a coordinated fashion such that no one simulator will get too far ahead in simulation time from the rest of the simulators in the group. In Simulus, a synchronized group is created with a "lookahead", which dictates how far a simulator can advance its simulation clock ahead of the rest of the simulators in the group. The value of the lookahead for a synchronized group is calculated automatically as we discuss momentarily.

To create a synchronized group, one calls the `sync()` function and passes as the parameter a list of the simulators (either by name of by reference). The `sync()` function will first bring all simulators in the group to synchrony, by advancing the simulation time of the individual simulators in the group to the maximum of the current simulation time of all simulators. From then on, the simulators are all bound to the synchronized group. That is, their simulation time will be advanced synchronously. In particular, Simulus uses the YAWNS protocol, which is a window-based conservative simulation synchronization algorithm, for synchronizing the simulators (Nicol 1993). Basically, at the start of a synchronization window, each simulator S_i finds the time of the earliest event on its event list, say t_i (infinite if the event list is empty). A global min-reduction of $(t_i + \delta)$ among all simulators determines the time of the next synchronization window, where δ is the lookahead.

5.2 Communication among Simulators

The simulators are placed in a synchronized group usually intended as a part of a large simulation model. The simulators need to communicate by sending and receiving timestamped messages among them. Simulus facilitates communication between simulators through named mailboxes. A message in simulus takes a broader meaning: a message can be any Python object as long as it's *pickle-able*. The simulators may run in different operating system processes (using separate Python interpreters) or on separate machines. Simulus uses Python's `pickle` module to serialize and deserialize the Python objects for the messages to travel across the process or machine boundaries. Each named mailbox must also specify a minimum delay for message transportation. It is required that the minimum delay of all named mailboxes in a synchronized group be strictly positive, as required by the conservative synchronization algorithm. The lookahead is calculated to be the minimum among all named mailboxes.

5.3 Parallel and Distributed Simulation

A synchronized group of simulators can run sequentially or in parallel. In the latter case, it can run on shared-memory multiprocessors, on distributed-memory machines in a cluster, or a combination of both.

To enable parallel simulation with shared-memory multiprocessing, one only needs to set the keyword argument `enable_smp` to be True when creating the synchronized group. Simulus will automatically fork separate operating system processes to run the simulators in the group. By default, Simulus will create as many operating system processes as the number of "processors" to run the simulation. If there are more simulators in the synchronized group than the processors, simulus tries to evenly divide the simulators among the processors. Simulus uses Python's `multiprocessing` package, which provides the interface for creating and managing operating system processes. Python does have a `threading` package to use threads. However, the Python interpreter (CPython) imposes a Global Interpreter Lock that allows only one thread to be executed at any given time. The `multiprocessing` uses operating system processes rather than threads and can thereby effectively sidestep the global lock issue. Simulus uses the `Queue` module for sending and receiving of the messages between the processes. If the message is a Python object, when the object is put on a queue, it is first "pickled" (for serialization) and a background thread later flushes the pickled message through an underlying pipe. As such, substantial overhead may incur, which unfortunately would negate the performance advantage of using shared-memory multiprocessing.

Simulus supports distributed-memory simulation by allowing the synchronized group of simulators to be instantiated and run on a parallel cluster. Each node in the parallel cluster runs a Simulus instance. They communicate using the Message-Passing Interface (MPI), which defines a common set of message-passing functions and data types, allowing users to write portable scientific code in C/C++ or Fortran. Simulus uses the `MPI4Py` package (MPI for Python), which provides a flexible Python interface for MPI using native Python data types (Lisandro Dalcin 2019). MPI4Py translates the syntax and semantics of standard MPI version 2 bindings for C++ to Python. The user creates by creating a *separate* synchronized group that includes only the simulators created by the Simulus instance, and sets the keyword argument `enable_spmd` to be True. Each Simulus instance runs as a separate MPI process identified by a rank, which is an integer that ranges from 0 to the total number of Simulus instances minus one. The instances are mapped by MPI, for example, one for each physical machine or one for each CPU core.

6 CONCLUSION

Simulus provides full-featured support for event-driven and process-oriented simulation by making use of language-specific features and available modules and packages in Python. Simulus is open source, which can be retrieved from github at https://github.com/liuxfiu/simulus/. Simulus can also be installed easily using `pip` (Package Installer for Python). This paper only focuses on important design decisions and major implementation issues. For future work, we plan to report the simulator's performance and scalability, using micro-benchmarks and also in the context of realistic simulation applications.

Currently, Simulus is being used to simulate HPC systems, computer networks, storage caching, cloud and edge computing systems, as well as transportation systems. Detailed applications will be reported in separate papers. We hereby examine some of the major use cases.

Interactive Simulation. A major advantage of using Simulus is that one can develop and run models interactively in Python's interactive development environments, such as the Jupyter Notebook (Kluyver et al. 2016). The Jupyter Notebook, in particular, is an web application that allows the users to create live documents with code, equations, visualizations, and narrative text. For example, one can do in-browser editing for code and directly execute the code from the browser with results attached to the code that generates them, including plots rendered using the `matplotlib` library. Simulation code can be put in one cell and straddled across multiple cells. Since Simulus functions are all tied to `simulator` objects, and the simulators can be created on the fly, one can treat a Jupyter Notebook document as an interactive console, for code development, documentation, testing, visualization, and debugging.

Simulation Ensembles. Simulation output analysis is the practice of analyzing the data generated by a simulation run to predict the performance of the simulated system. For stochastic simulation, one often needs to run the simulation for multiple times using different random streams. Statistical tools, such as the confidence interval, can be applied to help evaluate the performance measures of the target system. Similarly, simulation models often include many input factors, and the practice of determining the important factors hopefully with the least number of simulation runs is known as design of experiments. Both output analysis and design of experiments require multiple simulation runs with either different random sequences or input parameters. With Simulus, one can conduct experiments using multiple simulators instantiated either concurrently (in the case of the output analysis) or consecutively with guided parameter search (in the case of design of experiments).

Composable Simulation. Simulus provides the support for synchronously running multiple simulators. This allows for composable simulation with individual simulators separately designed for different components of the target system. While this does not solve the problem for reconciling the potentially different modeling details of the various simulation components, the synchronized group in Simulus provides an easy way, from a mechanical aspect, to assemble large complex system simulation models using small model components. Python provides relatively easier support for cross-language development (as a glue language) for interoperating with other simulators. One can create Python wrappers for existing simulation models and have them interoperate using Simulus.

REFERENCES

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al. 2016. "Tensorflow: A system for large-scale machine learning". In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 265–283.

Nezar Abdennur 2013. "Priority Queue Dictionary (Python Recipe)". http://code.activestate.com/recipes/578643-priority-queue-dictionary/. Last accessed: February 2020.

Borshchev, A. 2014. "Multi-method modelling: AnyLogic". *Discrete-event simulation and system dynamics for management decision making*:248–279.

Carothers, C. D., D. Bauer, and S. Pearce. 2000, May. "ROSS: A High-Performance, Low Memory, Modular Time Warp System". In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS'00)*, 53–60.

Conway, M. E. 1963, July. "Design of a Separable Transition-Diagram Compiler". *Commun. ACM* 6(7):396–408.

Lisandro Dalcin 2019. "MPI for Python". https://mpi4py.readthedocs.io/. Last accessed: February 2020.

Das, S., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994. "GTW: A Time Warp System for Shared Memory Multiprocessors". In *Proceedings of the 26th Conference on Winter Simulation (WSC'94)*, 1332–1339.

Erik DeBill 2020. "Module Counts". http://www.modulecounts.com. Last accessed: February 2020.

Hunter, J. D. 2007. "Matplotlib: A 2D graphics environment". *Computing in science & engineering* 9(3):90–95.

J-Sim Developers 2005. "J-Sim Official". https://sites.google.com/site/jsimofficial/. Last accessed: February 2020.

Jones, E., T. Oliphant, and P. Peterson. 2001. "SciPy: Open source scientific tools for Python".

Kluyver, T., B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay et al. 2016. "Jupyter Notebooks-a publishing format for reproducible computational workflows". In *ELPUB*, 87–90.

Knuth, D. E. 1997. *Fundamental Algorithms. The Art of Computer Programming*, Volume 1. Addison-Wesley.

Krahl, D. 2008. "ExtendSim 7". In *2008 Winter Simulation Conference*, 215–221. IEEE.

Liu, J., D. Nicol, B. Premore, and A. Poplawski. 1999, May. "Performance Prediction of a Parallel Simulator". In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, 156–164.

MacDougall, M. H., and J. S. McAlpine. 1973. "Computer system simulation with ASPOL". In *Proceedings of the 1st symposium on Simulation of computer systems*, 92–103. IEEE Press.

McKinney, W. et al. 2011. "pandas: a foundational Python library for data analysis and statistics". *Python for High Performance and Scientific Computing* 14(9).

Mikida, E., N. Jain, L. Kale, E. Gonsiorowski, C. D. Carothers, P. D. Barnes Jr, and D. Jefferson. 2016. "Towards pdes in a message-driven paradigm: A preliminary case study using charm++". In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 99–110.

Nance, R. E. 1996. *A History of Discrete Event Simulation Programming Languages*, 369–427. New York, NY, USA: Association for Computing Machinery.

Nicol, D. M. 1993, April. "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations". *Journal of the ACM* 40(2):304–333.

Nygaard, K., and O.-J. Dahl. 1978. "The development of the SIMULA languages". In *History of programming languages*, 439–480.

Oliphant, T. E. 2006. *A guide to NumPy*, Volume 1.

Page, E. H., R. L. Moose Jr, and S. P. Griffin. 1997. "Web-based simulation in Simjava using remote method invocation". In *Proceedings of the 29th conference on Winter simulation*, 468–474.

Panda, P. R. 2001. "SystemC: A Modeling Platform Supporting Multiple Design Abstractions". In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, 75–80. New York, NY, USA: Association for Computing Machinery.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al. 2019. "PyTorch: An imperative style, high-performance deep learning library". In *Advances in Neural Information Processing Systems*, 8024–8035.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al. 2011. "Scikit-learn: Machine learning in Python". *Journal of machine learning research* 12(Oct):2825–2830.

Perumalla, K. S. 2005. "μsik – A Micro-Kernel for Parallel/Distributed Simulation Systems". In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*, 59–68.

Rigo, A., and S. Pedroni. 2006. "PyPy's approach to virtual machine construction". In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 944–953.

Armin Rigo and Christian Tismer 2011. "greenlet: Lightweight concurrent programming". https://greenlet.readthedocs.io/. Last accessed: February 2020.

Rong, R., J. Hao, and J. Liu. 2014. "Performance Study of a Minimalistic Simulator on XSEDE Massively Parallel Systems". In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE'14)*, 15:1–15:8.

Rossetti, M. D. 2015. *Simulation modeling and Arena*. John Wiley & Sons.

Santhi, N., S. Eidenbenz, and J. Liu. 2015, Dec. "The Simian concept: Parallel Discrete Event Simulation with interpreted languages and just-in-time compilation". In *2015 Winter Simulation Conference (WSC)*, 3013–3024.

Schwetman, H. 1986. "CSIM: a C-based process-oriented simulation language". In *Proceedings of the 18th conference on Winter simulation*, 387–396.

SSF Research Network 2002. "Modeling the Global Internet". http://ssfnet.org/. Last accessed: February 2020.

Ståhl, I., R. G. Born, J. O. Henriksen, and H. Herper. 2011, Dec. "GPSS 50 years old, but still young". In *Proceedings of the 2011 Winter Simulation Conference (WSC)*, 3947–3957.

The PyPy Team 2020. "PyPy". https://www.pypy.org. Last accessed: February 2020.

Team SimPy 2007. "SimPy: Discrete event simulation for Python". http://simpy.readthedocs.org/. Last accessed: February 2020.

Thulasidasan, S., L. Kroc, and S. Eidenbenz. 2014. "Developing Parallel, Discrete Event Simulations in Python - First Results and User Experiences with the SimX Library". In *4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications, (SIMULTECH'14)*, 188–194.

Varga, A., and R. Hornig. 2008. "An Overview of the OMNeT++ Simulation Environment". In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08. Brussels, BEL: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Maneesh Varshney 2011. "SIM.JS – discrete event simulation in JavaScript". http://simjs.com/. Last accessed: February 2020.

ACKNOWLEDGMENTS

The author would like thank the anonymous reviewers for their constructive comments. This project is partially supported by the National Science Foundation Awards CCF-2008000 and CNS-1956229.

AUTHOR BIOGRAPHY

JASON LIU is a Professor at the School of Computing and Information Sciences, Florida International University. His research focuses on parallel simulation and high-performance modeling of computer systems and communication networks. His email address is liux@cis.fiu.edu.