Обзор фреймворка SimPy (simpy.readthedocs.io)

SimPy - Python фреймворк моделирования дискретных событий.

Поведение активных элементов (транспортных средств, клиентов, сообщений и т.д.) моделируется процессами, происходящими в особом программном окружении (Environment). Процессы взаимодействуют с этой средой и друг с другом через события.

Базовые понятия Simpy	**Реализация**
Environment	Цикл событий (Event loop)
Process	Со-программа (Task / Coroutine)
Event	Фьючерс (Future / Promise / Deffered)
Resource	Семафор (Semaphore)
RealtimeEnvironment	Цикл в режиме реального времени

Процессы описываются функциями-генераторами Python. Они могут быть реализованы в виде простой функции или метода класса. Процессы создают события, возвращают (**yield**) их и дожидаются выполнения.

Когда процесс генерирует событие, он приостанавливается. **SimPy** возобновляет процесс, когда событие должно быть выполнено. Несколько процессов могут ждать одно и то же событие.

Важным типом событий является Timeout. Это событие позволяет процессу находиться в спящем режиме (спать), то есть удерживать свое состояние в течение заданного модельного времени. Timeout, как и другие события, могут быть созданы путем вызова соответствующего метода модельной среды, в которой находится этот процесс (например, Environment.timeout ()).

Последовательность действий процесса задается в методе подкласса, называемом методом выполнения процесса (или сокращенно PEM). PEM взаимодействует с моделирующим движком, выдавая одно из нескольких ключевых слов (yield), определенных в процессе модели. Само моделирование выполняется с помощью функций базового модуля. Состояние процесса модели сохраняется в глобальной области. Это упрощает его реализацию и выполнение в модели (с наследованием от процесса и созданием экземпляров процессов перед запуском их PEM). Надо отметить, что наличие глобального состояния всей модели затрудняет распараллеливание нескольких симуляций.

Целью версии SimPy_3 было упрощение внутреннего API, необходимость очистить и структурировать его внутренние компоненты. Большинство изменений, с точки зрения конечного пользователя в основном синтаксические. Наиболее заметными изменениями в SimPy 3 являются:

- Больше не требуется подклассов процесса. PEM могут быть простыми функциями уровня модуля;

- Состояние модели теперь хранится в среде, которая может быть использована PEM для взаимодействия со средой моделирования;

- PEM теперь возвращают объекты событий: это интересная возможность, позволяющая легко расширять их с помощью новых типов событий.

Пример модели процесса

Пусть автомобиль будет поочередно ездить и парковаться на некоторое время. Когда машина начинает движение или паркуется выводится текущее время. Реализуем функцию, описывающую процесс движения и изменения состояния автомобиля.

```
def car(env):
    while True:
        print('Начало парковки в %d' % env.now)
        parking_duration = 5
        yield env.timeout(parking_duration)

        print('Начало движения в %d' % env.now)
        trip_duration = 2
        yield env.timeout(trip_duration)
```

Чтобы процесс создавал новые события, необходимо передать в функцию ссылку на среду моделирования (env). Поведение автомобиля прописано в бесконечном цикле. Так как функция является генератором, она вернет управление в среду моделирования, как только будет достигнута инструкция **yield**. После выполнения события выполнение функции возобновится со строчки, следующей за инструкцией **yield**.

Автомобиль поочередно находится в состояние движения или парковки. Он объявляет о своем новом состоянии, выводя в консоль сообщение и текущее модельное время Environment.now. Затем процесс вызывает метод `Environment.timeout ()` для создания события Timeout. Это событие описывает время, когда автомобиль находится на парковке или в движении. Когда процесс возвращает (через yield) событие, он сигнализирует среде моделирования, что хочет дождаться выполнения этого события.

```
import simpy
env = simpy.Environment()
env.process(car(env))
>>>
<Process(car) object at 0x...>
>>>
env.run(until=15)
>>>
Начало парковки в 0
Начало движения в 5
Начало парковки в 7
Начало движения в 12
Начало парковки в 14
```

Создадим экземпляр модельной среды, который передается в функцию. Далее вызовем метод среды моделирования ***Environment*.process**(), в который передадим экземпляр процесса. Он добавит в модельную среду новый процесс. Процесс, возвращаемый методом **process**(), может использоваться для взаимодействия процессов между собой.

Чтобы процесс начал свое выполнение, необходимо запустить моделирование, вызывая метод ***Environment*.run**(). Этот метод принимает в качестве параметра время окончания моделирования.

Взаимодействие процессов

Процесс, возвращаемый методом ***Environment*.process**(), может использоваться для взаимодействия процессов между собой. Это необходимо, когда следует дождаться выполнения другого процесса или прервать другой процесс.

Ожидание процесса

В **SimPy** процесс может использоваться как событие (технически, процесс тоже является событием). Если функция возвращает процесс, то ее выполнение продолжится после завершения процесса.

Реализуем модель парковки и движения электромобиля, который будет ждать, когда его батарея зарядится, прежде чем он снова сможет начать движение.

Процесс зарядки реализуем с помощью дополнительной функции **charge**(). Поэтому необходимо реорганизовать автомобиль как класс **Car** с двумя методами: **run**() и **charge**().

Процесс run автоматически запускается при создании экземпляра класса **Car**. Новый процесс **charge** выполняется каждый раз, когда автомобиль начинает парковку. Процесс run ожидает завершения процесса, который вернул метод ***Environment*.process**(), то есть процесса **charge**.

```
class Car(object):
    def __init__(self, env):
        self.env = env
        # Процесс запускается каждый раз, когда создается объект
        self.action = env.process(self.run())

    def run(self):
        while True:
           print('Начало парковки и зарядки в %d' % self.env.now)
           charge_duration = 5
           # Ожидаем, когда завершится зарядка
           yield self.env.process(self.charge(charge_duration))
           # Зарядка завершена. Начинаем движение
           print('Начало движения в %d' % self.env.now)
           trip_duration = 2
           yield self.env.timeout(trip_duration)

    def charge(self, duration):
        yield self.env.timeout(duration)
```

Чтобы запустить моделирование, создаем среду, машину и вызываем метод ***Environment*.run**().

```
import simpy
env = simpy.Environment()
car = Car(env)
env.run(until=15)
>>>
Начало парковки и зарядки в 0
Начало движения в 5
Начало парковки и зарядки в 7
Начало движения в 12
Начало парковки и зарядки в 14
```

Прерывание другого процесса

Предположим, что нет необходимости ждать, когда электромобиль зарядится полностью, то есть надо прервать процесс зарядки и начать движение.

SimPy позволяет прервать текущий процесс, вызвав метод ***interrupt***().

```
def driver(env, car):
   yield env.timeout(3)
   car.action.interrupt()
```

Процесс **driver** принимает в качестве параметра ссылку на экземпляр класса **Car**. Далее ожидает 3 единицы модельного времени и вызывает метод **interrupt**(), тем самым прерывая процесс. Прерывания передаются в функцию процесса как исключения, которые должны быть обработаны в прерванном процессе.

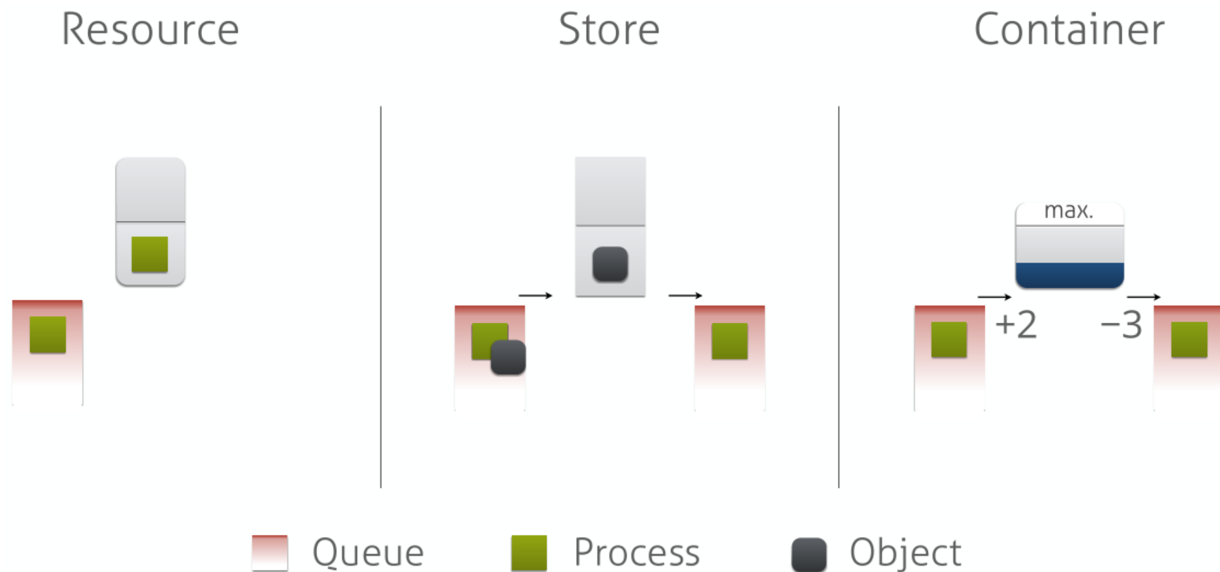
```
class Car(object):
    def __init__(self, env):
       self.env = env
       self.action = env.process(self.run())
    def run(self):
        while True:
           print('Начало парковки и зарядки в %d' % self.env.now)
           charge_duration = 5
           # Можно прервать зарядку
           try:
              yield self.env.process(self.charge(charge_duration))
           except simpy.Interrupt:
               # Прерываем зарядку и начинаем движение
               print('Зарядка прервана')
           print('Начало движения в %d' % self.env.now)
           trip_duration = 2
           yield self.env.timeout(trip_duration)

    def charge(self, duration):
        yield self.env.timeout(duration)
```

Теперь автомобиль начинает движение в момент времени 3, а не 5.

```
env = simpy.Environment()
car = Car(env)
env.process(driver(env, car))
>>>
<Process(driver) object at 0x...>
>>>
env.run(until=15)
>>>
Начало парковки и зарядки в 0
Зарядка прервана
Начало движения в 3
Начало парковки и зарядки в 5
Начало движения в 10
Начало парковки и зарядки в 12
```

Общие ресурсы

SimPy имеет несколько типов ресурсов (Resource), которые позволяют моделировать конфликты, когда несколько процессов хотят одновременно использовать ограниченный ресурс. Есть объекты – ресурсы трех видов: ресурс, хранилище(склад), контейнер(емкость).



Использование базового ресурса

Пусть автомобиль ездит на электрозарядку (ЭЗС), на которой есть две точки подключения зарядки. Если обе зарядки заняты, автомобиль ждет, когда одна из них освободиться. Затем он заряжается и покидает станцию.

```
def car(env, name, bcs, driving_time, charge_duration):
    # Моделирование ЭЗС
    yield env.timeout(driving_time)
    # Запрос одной из зарядок
    print('%s прибыл в %d' % (name, env.now))
    with bcs.request() as req:
        yield req
    # Зарядка
    print('%s начал зарядку в %s' % (name, env.now))
    yield env.timeout(charge_duration)
    print('%s покинул ЭЗС в %s' % (name, env.now))
```

Метод ресурса **request()** создает событие, которое занимает ресурс. Если процесс вернул это событие, то он владеет ресурсом, а все остальные ждут, когда ресурс освободится.

Если использовать менеджер контекста **with**, то ресурс освободится автоматически. Если метод **request()** вызван без менеджера контекста, то необходимо вызвать метод **release()**, чтобы закончить использование ресурса.

Когда процесс освобождает ресурс, следующий процесс из очереди занимает освободившееся место. Базовый ресурс **(Resource)** располагает процессы в очереди с дисциплиной FIFO (первый пришел – первый вышел). При создании ресурса необходимо предать ссылку на среду моделирования **(Environment)** и указать емкость ресурса.

```
import simpy
env = simpy.Environment()
bcs = simpy.Resource(env, capacity=2)
```

Создадим несколько процессов моделирующих автомобили и передадим ссылку на ресурс и дополнительные параметры.

```
for i in range(4):
    env.process(car(env, 'Автомобиль %d' % i, bcs, i*2, 5))
>>>
<Process(car) object at 0x...>
<Process(car) object at 0x...>
<Process(car) object at 0x...>
<Process(car) object at 0x...>
```

Теперь запустим моделирование. Оно закончится автоматически, как только все автомобили заправятся.

```
env.run()
>>>
Автомобиль 0 прибыл в 0
Автомобиль 0 начал заправку в 0
Автомобиль 1 прибыл в 2
Автомобиль 1 начал заправку в 2
Автомобиль 2 прибыл в 4
Автомобиль 0 покинул АЗС в 5
Автомобиль 2 начал заправку в 5
Автомобиль 3 прибыл в 6
Автомобиль 1 покинул АЗС в 7
Автомобиль 3 начал заправку в 7
Автомобиль 2 покинул АЗС в 10
Автомобиль 3 покинул АЗС в 12
```

Среда SimPy гибкая с точки зрения управления выполнением модели.

Можно выполнять модели до тех пор, пока больше не останется событий; или пока не пройдет определенное время моделирования; или до определенного события. Можно провести моделирование процесса по шагам по событиям. Кроме того, можно смешивать эти варианты.

Например, можно запустить моделирование до тех пор, пока не произойдет интересующее событие. Затем можно провести моделирование по событиям в течение некоторого времени; и затем запустить моделирование, пока не останется больше событий и все процессы закончатся.

Главный метод здесь - **Environment.run**():

Если вызвать его без каких-либо аргументов (env.run ()), он будет выполнять моделирование до тех пор, пока не останется больше событий. Если ваши процессы работают вечно (`while True: yield env.timeout (1)`), то метод никогда не завершится (пока вы не прервете свою модель, например, нажав Ctrl-C).

В большинстве случаев более целесообразно остановить моделирование, когда оно достигнет определенного времени моделирования. Для этого можно передать желаемое время через параметр until, например: `env.run(until=10)`.

Моделирование остановится, когда внутренние часы достигнут 10, и не будут обработаны события, запланированные на время 10.

Если надо использовать моделирование в GUI, и например, требуется нарисовать process bar, то можно многократно вызывать эту функцию с увеличением до конечного значения и обновлять индикатор выполнения после каждого вызова:

```
for i in range(100):
    env.run(until=i)
    progressbar.update(i)
```

Вместо того, чтобы передавать число run (), также можно передать ему любое событие. Тогда run () завершится, когда событие будет обработано. Например, если текущее время равно 0, `env.run (until=env.timeout(5))` будет эквивалентно `env.run(until=5)`.

Также можно передавать другие типы событий (здесь и процесс является событием):

```
import simpy
def my_proc(env):
... yield env.timeout(1)
... return 'Monty Python’s Flying Circus'
env = simpy.Environment()
proc = env.process(my_proc(env))
env.run(until=proc)

        'Monty Python’s Flying Circus'
```

Чтобы моделировать пошагово по событиям, в Simpy предлагаются методы peek() и step().

- peek () возвращает время следующего запланированного события.
- step () обрабатывает следующее запланированное событие. При этом, если событие недоступно, вызывается исключение.

В типичном случае использования эти методы используются в цикле, например:

```
untillimit = 10
while env.peek() < untillimit:
    env.step()
```

Среда моделирования позволяет получить текущее модельное время через свойство **Environment.now.** Время моделирования – это счетчик, который увеличивается через события тайм-аута.

По умолчанию время начинается с 0, но можно передать initial_time в Environment для использования другого начала отсчета. Хотя время моделирования технически абстрактно, вы можете предположить что это, например, в секундах и использовать его как метку времени, возвращаемую time.time() для расчета даты или дня недели.

Свойство **Environment.active_process** можно сравнить с os.getpid(), который указывает на текущий активный процесс. Процесс активен, когда выполняется его функция **process**. Он становится неактивным, когда он отдает событие (yield). Таким образом, имеет смысл получить доступ к этому свойству только из функции процесса или функции, вызываемой функцией процесса:

```
def subfunc(env):
  print(env.active_process) # will print "p1"

def my_proc(env):
  while True:
    print(env.active_process) # will print "p1"
    subfunc(env)
    yield env.timeout(1)

env = simpy.Environment()
p1 = env.process(my_proc(env))
env.active_process # None
env.step()
>>>
<Process(my_proc) object at 0x...>
<Process(my_proc) object at 0x...>
env.active_process # None
```

Примером использования является система ресурсов: если функция процесса вызывает request() чтобы запросить ресурс, ресурс определяет запрос процесс через env.active_process.

Чтобы создать события, обычно надо импортировать simpy.events, создать экземпляр класса Event и передать ему ссылку на среду. Чтобы уменьшить количество кода, среда предоставляет некоторые сокращения для обращения. Например, **Environment.event()** эквивалентен **simpy.events.Event(env)**. Есть и другие сокращения:

```
Environment.process()
Environment.timeout()
Environment.all_of()
Environment.any_of()
```

В Simpy функция-генератор может использоваться для предоставления возвращаемых значений для процессов, которые могут быть другими процессами:

```
def other_proc(env):
ret_val = yield env.process(my_proc(env))
assert ret_val == 42
```

или обычно так:

```
def my_proc(env):
  yield env.timeout(1)
  return 42
```

Для удобства читаемости, в среде есть метод exit(), чтобы сделать то же самое:

```
def my_proc(env):
   yield env.timeout(1)
   env.exit(42) # SimPy equivalent to "return 42"
```

Диспетчер событий Simpy основан на генераторах Python, и может использоваться и для асинхронной сети, и для реализации мультиагентных систем с моделируемыми взаимодействиями между ними.

Simpy включает набор типов событий для различных целей. Они наследуют **simpy.events.Event.** В приведенном ниже списке показана иерархия событий в Simpy:

```
events.Event
|
+— events.Timeout
|
+— events.Initialize
|
+— events.Process
|
+— events.Condition
|  |
|  +— events.AllOf
|  |
|  +— events.AnyOf
⋮
+- [resource events]
```

Это набор основных событий. События являются расширяемыми, и ресурсы, например, определяют дополнительные события.

Рассмотрим события в модуле Simpy.events. События SimPy очень похожи на отсрочки, фьючерсы или обещания, как и события класса, используемые для описания любого вида события. События могут находиться в одном из следующих состояний.

Событие может произойти (не срабатывает),

произойдет (срабатывает),

произошло (обработано).

События проходят эти состояния ровно по разу в этом порядке. События также плотно привязаны ко времени, и время заставляет события исполняться. Изначально события не запускаются, а создаются как объекты в памяти. Если событие запускается, оно назначается на заданное время и вставляется в Очередь событий SimPy. Состояние Event.triggered становится истинным (true) . Пока событие не обработано, можно добавлять обратные вызовы к событию.

Обратные вызовы - это вызываемые объекты, которые принимают событие в качестве параметра и хранятся в Event.callbacks.

Событие обрабатывается, когда SimPy выводит его из очереди событий и вызовет все его обратные вызовы. В этой фазе больше нельзя добавлять обратные вызовы. И для события свойство Event.processed становится истинным (true).

События также могут иметь значение. Значение может быть установлено до или во время срабатывания и может быть извлечено через Event.value или, в рамках процесса, получения события (value = yield event).

Наиболее распространенным способом добавления обратного вызова к событию является получение его из вашей функции процесса (событие yield). Этот способ добавит метод процесса

_resume () в качестве обратного вызова. Это происходит, если ваш процесс возобновляется, когда он выдал событие.

Однако можно добавить любой вызываемый объект (функцию) в список обратных вызовов, если он принимает экземпляр события в качестве единственного параметра:

```
 def my_callback(event):
  print('Called back from', event)

env = simpy.Environment()
event = env.event()
event.callbacks.append(my_callback)
event.callbacks
[<function my_callback at 0x...>]
```

Если событие было обработано, все его события были выполнены, то обратные вызовы и атрибут имеет значение None. Это помешает добавлять обратные вызовы - они, конечно, никогда не будут вызваны, потому что событие уже случилось.

Однако процессы «умны» в этом отношении. Если вы выдаете (yield) обработанное событие, _resume () немедленно возобновит ваш процесс со значением события (потому что ждать нечего).

Когда события срабатывают (triggered), они могут быть успешными или неудачными. Например, если событие должно быть инициировано в конце вычисления, и все работает отлично, то событие будет успешным. Если во время срабатывания происходит исключение, то событие будет неудачным.

Чтобы выполнить событие и отметить его как успешное, можно использовать `Event.succeed (value=None)`. Можно дополнительно передать ему значение(value) (например, результаты вычислений).

Чтобы исполнить событие и отметить его как неудачное (failed), вызовите Event.fail (exception) и передайте ему экземпляр исключения (например, исключение, которое случилось во время неудачного вычисления).

Существует также общий способ исполнения события: `Event.trigger(event)`. В этом случае можно будет принять значение и результат (успех или неудача) события, переданного ему.

Все три метода возвращают экземпляр события, к которому они привязаны. Что позволяет делать такие вещи, как `yield Event(env).succeed()`.

Рассмотрим пример работы с событиями: они могут создаваться процессом или за пределами контекста процесса; они могут передаваться другим процессам и вызывать другие события.

```
import simpy
class School:
 def __init__(self, env):
   self.env = env
   self.class_ends = env.event()
   self.pupil_procs = [env.process(self.pupil()) for i in range(3)]
   self.bell_proc = env.process(self.bell())

 def bell(self):
```

```
    for i in range(2):
      yield self.env.timeout(45)
      self.class_ends.succeed()
      self.class_ends = self.env.event()
      print('_bell_', i+1)

  def pupil(self):
    for i in range(2):
      print('\o/', end=' ')
      yield self.class_ends

env = simpy.Environment()
school = School(env)
env.run()
>>>
\o/ \o/ \o/ _bell_ 1
\o/ \o/ \o/ _bell_ 2
```

Такое поведение событий можно интерпретировать как действия passivate / reactivate. Ученики пассивизируются, когда занятие начинается и активизируются, когда звенит звонок с урока.

В Simpy процессы (как созданые Process или env.process ()) имеют полезное свойство быть событиями. Это означает, что один процесс может выдать (yield) другой процесс. Затем он будет возобновлен, когда закончится выданный процесс. Значением события будет возвращаемое значение этого процесса:

```
 def sub(env):
   yield env.timeout(1)
   return 23

 def parent(env):
   ret = yield env.process(sub(env))
   return ret

 env.run(env.process(parent(env)))
>>>
23
```

Когда процесс создается, он планирует инициализировать событие (Initialize), которое запустит выполнение процесса при срабатывании (triggered). Обычно вам не придется иметь дело с такого рода событиями.

Если вы не хотите, чтобы процесс начался немедленно, но после определенной задержки, можно использовать simpy.util.start_delayed (). Этот метод возвращает вспомогательный процесс, использующий тайм-аут перед началом процесса.

Вот пример с отложенным запуском sub():

```
from simpy.util import start_delayed

 def sub(env):
   yield env.timeout(1)
   return 23
```

```
 def parent(env):
   start = env.now
   sub_proc = yield start_delayed(env, sub(env), delay=3)
   assert env.now - start == 3

   ret = yield sub_proc
   return ret

 env.run(env.process(parent(env)))
>>>
23
```

Иногда надо ждать более одного события одновременно. Например, может потребоваться дождаться ресурса, но не более ограниченного времени. Или надо подождать, пока несколько событий случится.

Simpy предлагает особые события AnyOf и AllOf, которые являются событиями Condition.

В обоих случаях эти события берут список событий в качестве аргумента и выполняют (triggered) события: или хотя бы одно или все из них соответственно:

```
from simpy.events import AnyOf, AllOf, Event
events = [Event(env) for i in range(3)]
a = AnyOf(env, events) #Triggers if at least one of "events" is trigger
b = AllOf(env, events) #Triggers if all each of "events" is triggered.
```

Значение события Condition - это словарь с записью для каждого выполненного события. В случае AllOf размер этого словаря будет такой же, как и длина списка событий. Значение dict в случае AnyOf будет иметь хотя бы одну запись. В обоих случаях экземпляры событий используются в качестве ключей, а значения событий будут значениями словаря.

Также с AnyOf и AllOf можно использовать логические операторы & (и), | (или):

```
import simpy
from simpy.events import AnyOf, AllOf, Event

def test_condition(env):
  t1, t2 = env.timeout(1, value='spam'), env.timeout(2, value='eggs')
  ret = yield t1 | t2
  assert ret == {t1: 'spam'}

  t1, t2 = env.timeout(1, value='spam'), env.timeout(2, value='eggs')
  ret = yield t1 & t2
  assert ret == {t1: 'spam', t2: 'eggs'}

 # You can also concatenate & and |
  e1, e2, e3 = [env.timeout(i) for i in range(3)]
  yield (e1 | e2) & e3
  assert all(e.triggered for e in [e1, e2, e3])
  print(t1, t2, e1, e2, e3)

env = simpy.Environment()
proc = env.process(test_condition(env))
env.run()
```

Пример модели взаимодействия «докладчик-модератор»

1)

```
def speaker(env, start):
    until_start = start - env.now
    yield env.timeout(until_start)

    yield env.timeout(30)
```

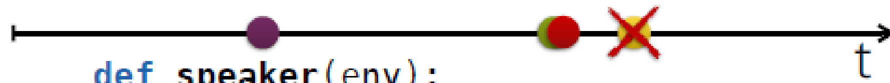
2)

```
def speaker(env):
    yield env.timeout(30)
    return 'handout'


def moderator(env):
    for i in range(3):
        val = yield env.process(speaker(env))
        print(val)
```

3)

```
def speaker(env):
    try:
        yield env.timeout(randint(25, 35))
    except simpy.Interrupt as interrupt:
        print(interrupt.cause)


def moderator(env):
    for i in range(3):
        speaker_proc = env.process(speaker(env))
        yield env.timeout(30)
        speaker_proc.interrupt('No time left')
```

4)




```python
def speaker(env):
    try:
        yield env.timeout(randint(25, 35))
    except simpy.Interrupt as interrupt:
        print(interrupt.cause)


def moderator(env):
    for i in range(3):
        speaker_proc = env.process(speaker(env))
        results = yield speaker_proc | env.timeout(30)

        if speaker_proc not in results:
            speaker_proc.interrupt('No time left')
```

Пример взаимодействия «клиент-сервер»

```python
def client(env, client_sock):
    message = Message(env, client_sock)
    reply = yield message.send('ohai')
    print(reply)


def server(env, server_sock):
    # Accept new connection
    sock = yield server_sock.accept()
    message = Message(env, PacketUTF8(sock))

    # Get message and send reply
    request = yield message.recv()
    print(request.content)
    yield request.succeed('cya')
```