SimJulia Documentation

Release

Ben Lauwens

August 09, 2016

Contents

CHAPTER 1

Welcome to SimJulia!

SimJulia is a combined continuous time / discrete event process oriented simulation framework written in Julia inspired by the Simula library DISCO and the Python library SimPy.

Its event dispatcher is based on a `Task`. This is a control flow feature in Julia that allows computations to be suspended and resumed in a flexible manner. *Processes* in SimJulia are defined by functions yielding *Events*. SimJulia also provides three types of shared resources to model limited capacity congestion points: *Resources*, *Containers* and *Stores*. The API is modeled after the SimPy API but using some specific Julia semantics.

The continuous time simulation is still under development and will be based on a quantized state system solver that naturally integrates in the discrete event framework.

SimJulia contains tutorials, in-depth documentation, and a large number of examples. The tutorials and the examples are borrowed from the SimPy distribution to allow a direct comparison and an easy migration path for users. The differences between SimJulia and SimPy are clearly documented.

SimJulia is released under the MIT License. The source code is freely available at the GitHub page of SimJulia. New ideas or interesting examples are always welcome and can be submitted as an issue or a pull request on GitHub.

CHAPTER 2

SimJulia in 10 Minutes

In this chapter, you'll learn the basics of SimJulia in just a few minutes. Afterwards, you will be able to implement a simple simulation using SimJulia and you'll be able to make an educated decision if SimJulia is what you need. Some hints are also given on how to proceed to implement more complex simulations.

2.1 Installation

SimJulia is implemented in pure Julia and has no dependencies. SimJulia runs on Julia v0.3 and Julia v0.4.

Note: Julia can be run from the browser without setup: JuliaBox

The built-in package manager of Julia is used to install SimJulia:

```
julia> Pkg.add("SimJulia")
```

You can now optionally run SimJulia's tests to see if everything is working fine:

```
julia> Pkg.test("SimJulia")
...
INFO: SimJulia tests passed
...
```

2.2 Discrete Events

SimJulia is a discrete-event simulation library. The behavior of active components (like vehicles, customers or messages) is modeled with *processes*. All processes live in an *environment*. They interact with the environment and with each other via *events*.

Processes are described by simple Julia functions. During their lifetime, they create events and *yield* them in order to wait for them to be triggered.

When a process yields an event, the process gets suspended. SimJulia resumes the process, when the event occurs (we say that the event is triggered). Multiple processes can wait for the same event. SimJulia resumes them in the same order in which they yielded that event.

An important event is a timeout. Events of this type are triggered after a certain amount of (simulated) time has passed. They allow a process to sleep (or hold its state) for the given time. A timeout and all other events can be created by calling an appropriate constructor having a reference to the environment that the process lives in.

2.2.1 The First Process

The first example will be a *car* process. The car will alternately drive and park for a while. When it starts driving (or parking), it will print the current simulation time.

So let's start:

```
julia> using SimJulia

julia> function car(env::Environment)
         while true
           println("Start parking at $(now(env))")
           parking_duration = 5.0
           yield(Timeout(env, parking_duration))
           println("Start driving at $(now(env))")
           trip_duration = 2.0
           yield(Timeout(env, trip_duration))
         end
       end
car (generic function with 1 method)
```

The car process function requires a reference to an *Environment* (env) in order to create new events. The car's behavior is described in an infinite loop. Though it will never terminate, it will pass the control flow back to the simulation once a *yield(ev)* statement is reached. If the yielded event is triggered ("it occurs"), the simulation will resume the function at this statement.

The car switches between the states parking and driving. It announces its new state by printing a message and the current simulation time (as returned by the function *now(env)*). It then calls the contructor *Timeout(env, parking_duration)* to create a timeout event. This event describes the point in time the car is done parking (or driving, respectively). By yielding the event, it signals the simulation that it wants to wait for the event to occur.

Now that the behavior of the car has been modeled, create an instance of it and see how it behaves:

```
julia> env = Environment()
Environment(0.0,PriorityQueue{BaseEvent,EventKey}(),0,0,Nullable{Process}())

julia> Process(env, car)
SimJulia.Process 1: car

julia> run(env, 15.0)
Start parking at 0.0
Start driving at 5.0
Start parking at 7.0
Start driving at 12.0
Start parking at 14.0
```

The first thing to do is to create an instance of *Environment*. This instance is passed into the car process function. Calling *Process(env, car)* creates a *Process* that needs to be started and added to the environment. Note, that at this time, none of the code of our process function is being executed. Its execution is merely scheduled at the current simulation time. The *Process* returned by *Process(env, car)* can be used for process interactions (this will be covered in the next section). Finally, the simulation starts by calling *run(env, 15.0)* where the second argument is the end time.

2.3 Process Interaction

The *Process* instance that is returned by the constructor *Process(env, func)* can be utilized for process interactions. The two most common examples for this are to wait for another process to finish and to Interrupt another

process while it is waiting for an event.

2.3.1 Waiting for a Process

As it happens, a SimJulia *Process* can be used like an event (technically, a *Process* is a subtype of AbstractEvent). If you yield it, you are resumed once the process has finished. Imagine a car-wash simulation where cars enter the car-wash and wait for the washing process to finish. Or an airport simulation where passengers have to wait until a security check finishes.

Assume that the car from the last example magically became an electric vehicle. Electric vehicles usually take a lot of time charging their batteries after a trip. They have to wait until their battery is charged before they can start driving again.

This can be modeled with an additional charge process. Therefore, two process functions are created: car(env) and charge(env, duration).

A new charge process is started every time the vehicle starts parking. By yielding the *Process* instance that *Process(env, func, args...)* returns, the run process starts waiting for it to finish:

```
julia> using SimJulia

julia> function car(env::Environment)
         while true
           println("Start parking and charging at $(now(env))")
           charge_duration = 5.0
           charge_proc = Process(env, charge, charge_duration)
           yield(charge_proc)
           println("Start driving at $(now(env))")
           trip_duration = 2.0
           yield(Timeout(env, trip_duration))
         end
       end
car (generic function with 1 method)

julia> function charge(env::Environment, duration::Float64)
         yield(Timeout(env, duration))
       end
charge (generic function with 1 method)
```

Starting the simulation is straightforward again: create an environment, one (or more) cars and finally call *run(env, 15.0)*:

```
julia> env = Environment()
Environment(0.0,PriorityQueue{BaseEvent,EventKey}(),0,0,Nullable{Process}())

julia> Process(env, car)
SimJulia.Process 1: car

julia> run(env, 15.0)
Start parking and charging at 0.0
Start driving at 5.0
Start parking and charging at 7.0
Start driving at 12.0
Start parking and charging at 14.0
```

2.3.2 Interrupting Another Process

Imagine, you don't want to wait until your electric vehicle is fully charged but want to interrupt the charging process and just start driving instead.

SimJulia allows you to interrupt a running process by calling the constructor *Interrupt(proc)* that returns an interrupt event.

An interrupt is thrown into process functions as an `InterruptException` that can (should) be handled by the interrupted process. The process can than decide what to do next (e.g., continuing to wait for the original event or yielding a new event):

```
julia> using SimJulia

julia> function driver(env::Environment, car_proc::Process)
         yield(Timeout(env, 3.0))
         yield(Interrupt(car_proc))
       end
driver (generic function with 1 method)

julia> function car(env::Environment)
         while true
           println("Start parking and charging at $(now(env))")
           charge_duration = 5.0
           charge_proc = Process(env, charge, charge_duration)
           try
             yield(charge_proc)
           catch exc
             println("Was interrupted. Hopefully, the battery is full enough ...")
           end
           println("Start driving at $(now(env))")
           trip_duration = 2.0
           yield(Timeout(env, trip_duration))
         end
       end
car (generic function with 1 method)

julia> function charge(env::Environment, duration::Float64)
         yield(Timeout(env, duration))
       end
charge (generic function with 1 method)
```

When you compare the output of this simulation with the previous example, you'll notice that the car now starts driving at time `3` instead of `5`:

```
julia> env = Environment()
Environment(0.0,PriorityQueue{BaseEvent,EventKey}(),0,0,Nullable{Process}())

julia> proc = Process(env, car)
SimJulia.Process 1: car

julia> Process(env, driver, proc)
SimJulia.Process 3: driver

julia> run(env, 15.0)
Start parking and charging at 0.0
Was interrupted. Hopefully, the battery is full enough ...
Start driving at 3.0
Start parking and charging at 5.0
```

```
Start driving at 10.0
Start parking and charging at 12.0
```

2.4 Shared Resources

SimJulia offers three types of resources that help you modeling problems, where multiple processes want to use a resource of limited capacity (e.g., cars at a fuel station with a limited number of fuel pumps) or classical producer-consumer problems.

In this section, SimJulia's *Resource* type is briefly introduced.

2.4.1 Basic Resource Usage

The electric vehicle process car introduced in the last sections is slightly modified.

The car drives now to a battery charging station (BCS) and requests one of its two charging spots. If both of these spots are currently in use, it waits until one of them becomes available again. It then starts charging its battery and leaves the station afterwards:

```
julia> using SimJulia

julia> function car(env::Environment, name::Int, bcs::Resource, driving_time::Float64, charge_duratio
         yield(Timeout(env, driving_time))
         println("$name arriving at $(now(env))")
         yield(Request(bcs))
         println("$name starting to charge at $(now(env))")
         yield(Timeout(env, charge_duration))
         println("$name leaving the bcs at $(now(env))")
         yield(Release(bcs))
       end
car (generic function with 1 method)
```

The constructor *Request(bcs)* generates an event that makes the process wait until the resource becomes available again. If the process is resumed, the process "owns" the resource until it releases it. Once the process is done using the resource, the event generated by the constructor *Release(bcs)* must be yielded. When a resource is released, the next waiting process is resumed and now "owns" one of the resource's slots. The basic *Resource* sorts waiting processes in a *FIFO* (first in—first out) way.

A resource needs a reference to an *Environment* and a *capacity* when it is created:

```
julia> env = Environment()
Environment(0.0,PriorityQueue{BaseEvent,EventKey}(),0,0,Nullable{Process}())

julia> bcs = Resource(env, 2)
Resource(Environment(0.0,PriorityQueue{BaseEvent,EventKey}(),0,0,Nullable{Process}()),2,0,PriorityQue
```

The car processes can now be created and a reference to our resource as well as some additional parameters are passed to them:

```
julia> for i=0:3
         Process(env, car, i, bcs, 2.0*i, 5.0)
       end
```

Finally, the simulation can be started. Since the car processes all terminate on their own in this simulation, an until time is not specified—the simulation will automatically stop when there are no more events left:

```
julia> run(env)
0 arriving at 0.0
0 starting to charge at 0.0
1 arriving at 2.0
1 starting to charge at 2.0
2 arriving at 4.0
0 leaving the bcs at 5.0
2 starting to charge at 5.0
3 arriving at 6.0
1 leaving the bcs at 7.0
3 starting to charge at 7.0
2 leaving the bcs at 10.0
3 leaving the bcs at 12.0
```

Note that the first two cars can start charging immediately after they arrive at the BCS, while cars `2` an `3` have to wait.

2.5 How to Proceed

If you are not certain yet if SimJulia fulfills your requirements or if you want to see more features in action, you should take a look at the various examples.

If you are looking for a more detailed description of a certain aspect or feature of SimJulia, the Topical Guides section might help you.

Finally, there is an API Reference that describes all functions and classes in full detail.

CHAPTER 3

Topical Guides

This sections covers various aspects of SimJulia more in-depth. It assumes that you have a basic understanding of SimJulia's capabilities.

3.1 SimJulia basics

This guide describes the basic concepts of SimJulia: How does it work? What are processes, events and the environment? What can I do with them?

3.1.1 How SimJulia works

If you break SimJulia down, it is just an asynchronous event dispatcher, very similar to SimPy. You generate events and schedule them at a given simulation time. Events are sorted by priority, simulation time, and an increasing event id. An event also has a list of callbacks, which are executed when the event is triggered and processed by the event loop. Events may also have a return value.

The components involved in this are the *Environment*, `AbstractEvent` and the process functions that you write.

Process functions implement your simulation model, that is, they define the behavior of your simulation. They are plain Julia functions that have at least an environment as argument and are wrapped into a `Task`. A task is a control flow feature that allows computations to be suspended and resumed. A process function can so be interrupted by switching to another task. The original task can later be resumed, at which point it will pick up right where it left off. Julia provides the functions `produce()` and `consume()` to implement this functionality. (These functions are however not used directly in SimJulia)

A call of the SimJulia function *yield(ev::AbstractEvent)* suspends the process function until the event is triggered and processed. The environment stores the event in its event list and a resume function is added to the event's callbacks. The environment selects the next event from its event list and keeps track of the current simulation time. The callbacks from this event are called in the same order as they were added. In case the callback is a resume function the associated task is resumed and its process function continues at the point where it left off.

Here is a very simple example that illustrates this all; the code is more verbose than it needs to be to make things extra clear. You find a compact version of it at the end of this section:

```
using SimJulia

function example(env::Environment)
  ev = Timeout(env, 1.0, 42)
  value = yield(ev)
  time = now(env)
```

```
  println("now=$time, value=$value")
end

env = Environment()
p = Process(env, example)
run(env )
```

The `example(env::Environment)` process function above first creates a timeout event with the constructor *`Timeout(env::AbstractEnvironment, delay::Float64, value::Any)`*. It passes the environment, a delay, and a value. This event is automatically schedulled at now + delay (that's why the environment is required); other events usually schedule themselves at the current simulation time.

The process function then yields the timeout event and thus gets suspended. It is resumed, when SimJulia processes the event. The process function also receives the event's value (*42*) – this is however optional.

Finally, the process function prints the current simulation time that is accessible via the function *`now(env::Environment)`* and the value of the event.

Once all process functions are defined, you can instantiate the objects for your simulation. In most cases, you start by creating an instance of *`Environment`*, because you'll need to pass it around a lot when creating everything else.

You have to call the constructor *`Process(env::AbstractEnvironment, func::Function, args...)`* to wrap the process function into the `Task`. This will not execute any code of the process function yet but will schedule an event at the current simulation time which starts the execution of the process function. An instance of the *`Process`* can also be yielded and is triggered when its process function returns.

Finally, you start SimJulia's event loop by calling *`run(env)`*. By default, it will run as long as there are events in the event list, but you can also let it stop earlier by providing an until argument.

"Best practice" version of the example above:

```
using SimJulia

function example(env::Environment)
  value = yield(Timeout(env, 1.0, 42))
  println("now=$(now(env)), value=$value")
end

env = Environment()
p = Process(env, example)
run(env )
```

3.2 Environments

A simulation environment manages the simulation time as well as the scheduling and processing of events. It also provides means to step through or execute the simulation.

The abstract type for all environments is `AbstractEnvironment`. "Normal" simulations usually use its subtype *`Environment`*.

3.2.1 Simulation control

SimJulia is very flexible in terms of simulation execution. You can run your simulation until there are no more events, until a certain simulation time is reached, or until a certain event is triggered. You can also step through the simulation event by event. Furthermore, you can mix these things as you like. For example, you could run your simulation until

an interesting event occurs. You could then step through the simulation event by event for a while; and finally run the simulation until there are no more events left and your processes have all terminated.

The most important method in this section is *run()*:

- If you call it with only one argument *run(env::Environment)*, it steps through the simulation until there are no more events left.

Warning: If your process function runs forever, e.g.

`while true`
`  yield(Timeout(env, 1.0))`
`end`

run(env) will never terminate unless you kill your script by pressing Ctrl-C.

- In most cases it is advisable to stop your simulation when it reaches a certain simulation time. Therefore, you can pass the desired time via a second argument: *run(env, 10.0)*. The simulation will then stop when the internal clock reaches `10.0` but will not process any events scheduled for time `10.0`. This is similar to a new environment where the clock is `0.0` but (obviously) no events have yet been processed.
- Instead of passing a floating point value as second argument, you can also pass any instance of a `AbstractEvent` to it. The function returns when this event has been processed. Assuming that the current time is `0.0`, *run(env, Timeout(env, 5.0))* is equivalent to *run(env, 5.0)*. You can also pass other types of events (remember, that *Process* is a subtype of `AbstractEvent`).

```
using SimJulia

function my_proc(env::Environment)
  yield(Timeout(env, 1.0))
  return "Monty Python's Flying Circus"
end

env = Environment()
proc = Process(env, my_proc)
println(run(env, proc))
```

To step through the simulation event by event, the environment offers *peek(env::Environment)* and *step(env::Environment)*:

- *peek(env::Environment)* returns the time of the next scheduled event or `Inf` when no more events are scheduled.
- *step(env::Environment)* processes the next scheduled event. It raises an `EmptySchedule` exception if no event is available.

In a typical use case, you use these methods in a loop like:

```
until = 10.0
while peek(env) < until
  step(env)
end
```

3.2.2 State access

The environment allows you to get the current simulation time via the function *now(env::Environment)*. The simulation time is a floating point value without unit and is increased via timeout events.

By default, the constructor *Environment()* starts the simulation time at `0.0`, but you can pass an initial value, *Environment(initial_value::Float64)* to use something else.

The function *active_process(env::Environment)* is comparable to `Base.getpid()` and returns the currently active *Process*. A process is *active* when its process function is being executed. It becomes *inactive* (or suspended) when it yields an event.

Thus, it only makes sense to call this function from within a process function or a function that is called by your process function, otherwise, a `NullException` is thrown:

```
using SimJulia

function subfunc(env::Environment)
  println("Active process: $(active_process(env))")
end

function my_proc(env::Environment)
  println("Active process: $(active_process(env))")
  yield(Timeout(env, 1.0))
  subfunc(env)
end

env = Environment()
Process(env, my_proc)
println("Time: $(peek(env))")
try
  println(active_process(env))
catch exc
  println("No active process")
end
step(env)
println("Time: $(peek(env))")
try
  println(active_process(env))
catch exc
  println("No active process")
end
step(env)
println("Time: $(peek(env))")
step(env)
println("Time: $(peek(env))")
```

A nice example of this function can be found in the resource system. When a process function calls the constructor *Request(res::Resource)* to generate a request event for a resource, the resource determines the requesting process via *active_process(env)*.

3.2.3 Event creation

To create events, you normally have to use a constructor *Event(env::AbstractEnvironment)* to instantiate the *Event* type and pass a reference to the environment to it.

More details on what events do can be found in the next sections.

3.2.4 Miscellaneous

A process function can have a return value:

```
using SimJulia

function my_proc(env::Environment)
```

```
  yield(Timeout(env, 1.0))
  return 42
end

function other_proc(env::Environment)
  ret_val = yield(Process(env, my_proc))
  @assert(ret_val == 42)
end

env = Environment()
Process(env, other_proc)
run(env)
```

The simulation can be stopped by throwing a `StopSimulation` exception in a process function. To keep your code more readable, the function *`stop_simulation(env::AbstractEnvironment)`* does exactly this.

3.3 Events

SimJulia includes an extensive set of event constructors for various purposes. This section details the following events:

- basic events:
  - *`Event`*
  - *`Timeout`*
- compound events:
  - *`EventOperator`*
  - *`AllOf`*
  - *`AnyOf`*

A *`Process`* is also an event and is also discussed.

The resource and container event constructors are discussed in a later section.

3.3.1 Event basics

SimJulia events are very similar – if not identical — to deferreds, futures or promises. Instances of the type `AbstractEvent` are used to describe any kind of events. Events can be in one of the following states:

- *not triggered*: an event may happen
- *triggered*: is going to happen
- *processing*: is happening
- *processed*: has happened

They traverse these states exactly once in that order. Events are also tightly bound to time and time causes events to advance their state. Initially, events are not triggered and just objects in memory.

If an event gets triggered, it is scheduled at a given time and inserted into SimJulia's event list. The function *`triggered(ev::Event)`* returns `true`. As long as the event is not processed, you can add callbacks to an event. Callbacks are functions that have as first argument an *`Event`* and are stored in the callbacks list of that event. An event becomes processed when SimJulia has popped it from the event list and has called all of its callbacks. It is no longer possible to add callbacks. The function *`processed(ev::Event)`* returns at that moment `true`.

Events also have a value. The value can be set before or when the event is triggered and can be retrieved via the function *value(ev::Event)* or, within a process, via the return value of the function *yield(ev::AbstractEvent)*.

3.3.2 Adding callbacks to an event

"What? Callbacks? I've never seen no callbacks!", you might think if you have worked your way through the tutorial.

That's on purpose. The most common way to add a callback to an event is yielding it from your process function (*yield(ev::AbstractEvent)*). This will add the function `proc.resume(ev::AbstractEvent)` as a callback. That's how your process gets resumed when it yielded an event.

However, you can add a function to the list of callbacks as long as it accepts an instance of type *Event* as its first argument using the function *append_callback(ev::AbstractEvent, callback::Function, args...)*:

```
using SimJulia

function my_callback(event::Event)
  println("Called back from $event")
end

env = Environment()
event = Event(env)
append_callback(event, my_callback)
succeed(event)
run(env)
```

If an event has been processed, all of its callbacks have been called. Adding more callbacks – these would of course never get called because the event has already happened - results in the throwing of a `EventProcessed` exception.

Processes are smart about this, though. If you yield a processed event, your process will immediately resume with the value of the event (because there is nothing to wait for).

3.3.3 Triggering events

When events are triggered, they can either succeed or fail. For example, if an event is to be triggered at the end of a computation and everything works out fine, the event will succeed. If an exceptions occurs during that computation, the event will fail.

To trigger an event and mark it as successful, you can use *succeed(ev::AbstractEvent, value=nothing)*. You can optionally pass a value to it (e.g., the results of a computation).

To trigger an event and mark it as failed, call *fail(ev::AbstractEvent, exc::Exception)* and pass an `Exception` instance to it (e.g., the exception you caught during your failed computation).

There is also a generic way to trigger an event: *trigger(ev::AbstractEvent, cause::BaseEvent)*. This will take the value and outcome (success or failure) of the event passed to it.

All three methods return the event instance they are bound to. This allows you to do things like:

```
yield succeed(Event(env))
```

Triggering an event that was already triggered before results in the throwing of a `EventTriggered` exception.

3.3.4 Example usages for `Event`

The simple mechanics outlined above provide a great flexibility in the way events can be used.

One example for this is that events can be shared. They can be created by a process or outside of the context of a process. They can be passed to other processes and chained:

```
using SimJulia

type School
  class_ends :: Event
  pupil_procs :: Vector{Process}
  bell_proc :: Process
  function School(env::Environment)
    school = new()
    school.class_ends = Event(env)
    school.pupil_procs = Process[Process(env, pupil, school) for i=1:3]
    school.bell_proc = Process(env, bell, school)
    return school
  end
end

function bell(env::Environment, school::School)
  for i=1:2
    yield(Timeout(env, 45.0))
    succeed(school.class_ends)
    school.class_ends = Event(env)
    println()
  end
end

function pupil(env::Environment, school::School)
  for i=1:2
    print(" \\o/")
    yield(school.class_ends)
  end
end

env = Environment()
school = School(env)
run(env)
```

3.3.5 Let time pass by

To actually let time pass in a simulation, there is the *Timeout*. A timeout constructor has three arguments: *Timeout(env::AbstractEnvironment, delay::Float64, value=nothing)*. It is triggered automatically and is scheduled at `now + delay`. Thus, the *succeed(ev::AbstractEvent, value=nothing)*, *fail(ev::AbstractEvent, exc::Exception)* and *trigger(ev::AbstractEvent, cause::AbstractEvent)* functions cannot be called again and you have to pass the event value to it when you create the timeout event.

3.3.6 Processes are events, too

SimJulia processes (as created by the constructor *Process(env::AbstractEnvironment, func::Function, args...)*) have the nice property of being a subtype of `AbstractEvent`, too.

That means, that a process can yield another process. It will then be resumed when the other process ends. The event's value will be the return value of that process:

```
using SimJulia

function sub(env::Environment)
  yield(Timeout(env, 1.0))
  return 23
end

function parent(env::Environment)
  return ret = yield(Process(env, sub))
end

env = Environment()
ret = run(env, Process(env, parent))
println(ret)
```

When a process is created, it schedules an event which will start the execution of the process when triggered. You usually won't have to deal with this type of event.

If you don't want a process to start immediately but after a certain delay, you can use *DelayedProcess(env::AbstractEnvironment, delay::Float64, func::Function, args...)*. This method returns a helper process that uses a timeout before actually starting the process.

The example from above, but with a delayed start of `sub(env::Environment)`:

```
using SimJulia

function sub(env::Environment)
  yield(Timeout(env, 1.0))
  return 23
end

function parent(env::Environment)
  start = now(env)
  sub_proc = yield(DelayedProcess(env, 3.0, sub))
  @assert(now(env) - start == 3.0)
  ret = yield(sub_proc)
end

env = Environment()
ret = run(env, Process(env, parent))
println(ret)
```

The state of the *Process* can be queried with the function `is_process_done(proc::Process)` that returns `true` when the process function has returned.

3.3.7 Waiting for multiple events at once

Sometimes, you want to wait for more than one event at the same time. For example, you may want to wait for a resource, but not for an unlimited amount of time. Or you may want to wait until all a set of events has happened.

SimJulia therefore offers the event constructors *AnyOf(events...)* and *AllOf(events...})*. Both take a list of events as an argument and are triggered if at least one or all of them of them are triggered. There is a specific constructors for the more general *EventOperator(eval::Function, events...)*. The function `eval(events...)` takes a tuple of `AbstractEvent` as argument and returns true when the condition is fulfilled.

As a shorthand for *AllOf(events...)* and *AnyOf(events...)*, you can also use the logical operators `&` (and) and `|` (or):

```
using SimJulia
using Compat

function test_condition(env::Environment)
  t1, t2 = Timeout(env, 1.0, "spam"), Timeout(env, 2.0, "eggs")
  ret = yield(t1 | t2)
  @assert(ret == @compat Dict(t1=>"spam"))
  t1, t2 = Timeout(env, 1.0, "spam"), Timeout(env, 2.0, "eggs")
  ret = yield(t1 & t2)
  @assert(ret == @compat Dict(t1=>"spam", t2=>"eggs"))
  e1, e2, e3 = Timeout(env, 1.0, "spam"), Timeout(env, 2.0, "eggs"), Timeout(env, 3.0, "eggs")
  yield((e1 | e2) & e3)
  @assert(all(map((ev)->processed(ev), [e1, e2, e3])))
end

env = Environment()
Process(env, test_condition)
run(env)
```

The result of the `yield` of a multiple events is of type `Dict` with as keys the processed (processing) events and as values their values. This allows the following idiom for conveniently fetching the values of multiple events specified in an and condition (including *`AllOf(events...)`*):

```
using SimJulia
using Compat

function fetch_values_of_multiple_events(env::Environment)
  t1, t2 = Timeout(env, 1.0, "spam"), Timeout(env, 2.0, "eggs")
  ret = yield(t1 & t2)
  @assert(ret == @compat Dict(t1=>"spam", t2=>"eggs"))
end

env = Environment()
Process(env, fetch_values_of_multiple_events)
run(env)
```

3.4 Process Interaction

Discrete event simulation is only made interesting by interactions between processes.

So this section is about:

- Sleep until woken up
- Waiting for another process to terminate
- Interrupting another process

The first two items were already covered in the previous section, but they are included here for the sake of completeness.

Another possibility for processes to interact are resources. They are discussed in the next section.

3.4.1 Sleep until woken up

Imagine you want to model an electric vehicle with an intelligent battery-charging controller. While the vehicle is driving, the controller can be passive but needs to be reactivate once the vehicle is connected to the power grid in order

to charge the battery.

In SimJulia, you can accomplish that with a simple, shared Event:

```
using SimJulia

type EV
  bat_ctrl_reactivate :: Event
  function EV(env::Environment)
    ev = new()
    ev.bat_ctrl_reactivate = Event(env)
    Process(env, drive, ev)
    Process(env, bat_ctrl, ev)
    return ev
  end
end

function drive(env::Environment, ev::EV)
  while true
    yield(Timeout(env, 20.0*rand()+20.0))
    println("Start parking at $(now(env))")
    succeed(ev.bat_ctrl_reactivate)
    ev.bat_ctrl_reactivate = Event(env)
    yield(Timeout(env, 300.0*rand()+60.0))
    println("Stop parking at $(now(env))")
  end
end

function bat_ctrl(env::Environment, ev::EV)
  while true
    println("Bat. ctrl. passivating at $(now(env))")
    yield(ev.bat_ctrl_reactivate)
    println("Bat. ctrl. reactivated at $(now(env))")
    yield(Timeout(env, 60*rand()+30))
  end
end

env = Environment()
ev = EV(env)
run(env, 150.0)
```

The process function `bat_ctrl()` just waits for a normal event.

3.4.2 Waiting for another process to terminate

The example above has a problem: it may happen that the vehicles wants to park for a shorter duration than it takes to charge the battery (this is the case if both, charging and parking would take 60 to 90 minutes).

To fix this problem we have to slightly change our model. A new `bat_ctrl()` will be started every time the EV starts parking. The EV then waits until the parking duration is over and until the charging has stopped:

```
using SimJulia

function drive(env::Environment)
  while true
    yield(Timeout(env, 20.0*rand()+20.0))
    println("Start parking at $(now(env))")
    charging = Process(env, bat_ctrl)
```

```
    parking = Timeout(env, 300.0*rand()+60.0)
    yield(charging & parking)
    println("Stop parking at $(now(env))")
  end
end

function bat_ctrl(env::Environment)
  println("Bat. ctrl. started at $(now(env))")
  yield(Timeout(env, 60*rand()+30))
  println("Bat. ctrl. done at $(now(env))")
end

env = Environment()
Process(env, drive)
run(env, 310.0)
```

Again, nothing new and special is happening. SimJulia processes are subtypes of `BaseEvent`, so you can yield. You can also wait for two events at the same time by concatenating them with `&`.

3.4.3 Interrupting another process

As usual, another problem can be considered: Imagine, a trip is very urgent, but with the current implementation, we always need to wait until the battery is fully charged. If we could somehow interrupt that ...

Fortunate coincidence, there is indeed a way to do exactly this. You can call the event constructor *`Interrupt(proc::Process, cause::Any=nothing)`*. This will throw an `InterruptException` into that process, resuming it immediately:

```
using SimJulia

function drive(env::Environment)
  while true
    yield(Timeout(env, 20.0*rand()+20.0))
    println("Start parking at $(now(env))")
    charging = Process(env, bat_ctrl)
    parking = Timeout(env, 60.0)
    yield(charging | parking)
    if !is_process_done(charging)
      yield(Interrupt(charging, "Need to go!"))
    end
    println("Stop parking at $(now(env))")
  end
end

function bat_ctrl(env::Environment)
  println("Bat. ctrl. started at $(now(env))")
  try
    yield(Timeout(env, 60*rand()+30))
    println("Bat. ctrl. done at $(now(env))")
  catch(exc)
    println("Bat. ctrl. Interrupted at $(now(env)), msg: $(msg(exc))")
  end
end

env = Environment()
Process(env, drive)
run(env, 100.0)
```

What the event constructor *Interruption(proc::Process, cause::Any=nothing)* actually does is scheduling an interrupt event for immediate execution. If this event is executed it will remove the victim process' `proc.resume(ev::AbstractEvent)` from the callbacks of the event that it is currently waiting for. Following that it will throw an `InterruptException` into the process function.

An interrupt event constructor must be yielded immediately. The interrupt event has a higher priority than all other events and only after the scheduling of the interrupt event, the interrupting process can be resumed.

The cause of the interrupt can be found by calling the function *cause(inter::InterruptException)*.

Since nothing special has been done to the original target event of the process, the interrupted process can yield the same event again after catching the Interrupt – Imagine someone waiting for a shop to open. The person may get interrupted by a phone call. After finishing the call, he or she checks if the shop already opened and either enters or continues to wait.

3.5 Shared Resources

Shared resources are another way to model process interaction. They form a congestion point where processes queue up in order to use them.

SimJulia defines three categories of resources:

- *Resource*: Resources that can be used by a limited number of processes at a time (e.g., a gas station with a limited number of fuel pumps).
- `Container`: Resources that model the production and consumption of a homogeneous, undifferentiated bulk. It may either be continuous (like water) or discrete (like apples).
- `Store`: Resources that allow the production and consumption of Julia types.

Note: All resources are implemented using only the exported functions of SimJulia and are showcases of the functionalities of the previous chapters.

3.5.1 The basic concept of resources

All resources share the same basic concept: The resource itself is some kind of a container with a, usually limited, capacity. Processes can either try to put something into the resource or try to get something out. If the resource is full or empty, they have to queue up and wait.

Every resources has maximum capacity and two queues, one for processes that want to put something into it and one for processes that want to get something out. The *Put()* and *Get()* constructors both return an event that is triggered when the corresponding action was successful.

3.5.2 Resources and interrupts

While a process is waiting for a resource, it may be interrupted by another process. After catching the interrupt, the process has two possibilities:

- It may continue to wait for the request (by yielding the event again).
- It may stop waiting for the request (by calling the *cancel()*).

The resource system is modular and extensible. Resources can, for example, use specialized queues. This allows them to add priorities to events or to offer preemption.

3.5.3 Resources

Resources can be used by a limited number of processes at a time (e.g., a gas station with a limited number of fuel pumps). Processes request these resources to become a user (or to "own" them) and have to release them once they are done (e.g., vehicles arrive at the gas station, use a fuel-pump, if one is available, and leave when they are done).

Requesting a resources is modeled as "putting a process' token into the resources" and releasing a resources correspondingly as "getting a process' token out of the resource". Releasing a resource will always succeed immediately. Requesting and releasing a resource is done by yielding a request / release event. The request event has the following constructor `Request(res::Resource, priority::Int=0, preempt::Bool=false)` and the release event `Release(res::Resource)`.

The `Resource` is conceptually a semaphore. The only argument of its constructor – apart from the obligatory reference to an Environment – is its capacity. It must be a positive number and defaults to 1: `Resource(env::AbstractEnvironment, capacity::Int=1)`.

Instead of just counting its current users, it stores the requesting process as an "access token" for each user. This is, for example, useful for adding preemption (see further).

Here is as basic example for using a resource:

```
using SimJulia

function print_stats(res::Resource)
  println("$(count(res)) of $(capacity(res)) are allocated.")
end

function resource_user(env::Environment, res::Resource)
  print_stats(res)
  yield(Request(res))
  print_stats(res)
  yield(Release(res))
  print_stats(res)
end

env = Environment()
res = Resource(env, 1)
Process(env, resource_user, res)
Process(env, resource_user, res)
run(env)
```

The functions `count(res::Resource)` and `capacity(res::Resource)` return respectively the number of processes using the resource and the capacity of the resource.

3.5.4 Priority resource

As you may know from the real world, not every one is equally important. To map that to SimJulia, the constructor `Request(res::Resource, priority::Int=0, preempt::Bool=false)` lets requesting processes provide a priority for each request. More important requests will gain access to the resource earlier than less important ones. Priority is expressed by integer numbers; smaller numbers mean a higher priority:

```
using SimJulia

function resource_user(env::Environment, name::Int, res::Resource, wait::Float64, prio::Int)
  yield(Timeout(env, wait))
  println("$name Requesting at $(now(env)) with priority=$prio")
  yield(Request(res, prio))
  println("$name got resource at $(now(env))")
```

```
  yield(Timeout(env, 3.0))
  yield(Release(res))
end

env = Environment()
res = Resource(env, 1)
p1 = Process(env, resource_user, 1, res, 0.0, 0)
p2 = Process(env, resource_user, 2, res, 1.0, 0)
p3 = Process(env, resource_user, 3, res, 2.0, -1)
run(env)
```

Although `p3` requested the resource later than `p2`, it could use it earlier because its priority was higher.

3.5.5 Preemptive resource

Sometimes, new requests are so important that queue-jumping is not enough and they need to kick existing users out of the resource (this is called preemption). As before the constructor *Request(res::Resource, priority::Int=0, preempt::Bool=false)* allows you to do exactly this:

```
using SimJulia

function resource_user(env::Environment, name::Int, res::Resource, wait::Float64, prio::Int)
  yield(Timeout(env, wait))
  println("$name Requesting at $(now(env)) with priority=$prio")
  yield(Request(res, prio, true))
  println("$name got resource at $(now(env))")
  try
    yield(Timeout(env, 3.0))
    yield(Release(res))
  catch exc
    pre = cause(exc)
    usage = now(env) - usage_since(pre)
    println("$name got preempted by $(by(pre)) at $(now(env)) after $usage")
  end
end

env = Environment()
res = Resource(env, 1)
p1 = Process(env, resource_user, 1, res, 0.0, 0)
p2 = Process(env, resource_user, 2, res, 1.0, 0)
p3 = Process(env, resource_user, 3, res, 2.0, -1)
run(env)
```

An `InterruptException` is generated. Its cause is of type *Preempted*, so that the functions *by(pre::Preempted)* and *usage_since(pre::Preempted)* return respectively the preempting process and the duration that the preempted process has hold the resource.

The implementation values priorities higher than preemption. That means preempt request are not allowed to cheat and jump over a higher prioritized request. The following example shows that preemptive low priority requests cannot queue-jump over high priority requests:

```
using SimJulia

function user(env::Environment, name::ASCIIString, res::Resource, wait::Float64, prio::Int, preempt:
  println("$name Requesting at $(now(env))")
  yield(Request(res, prio, preempt))
  println("$name got resource at $(now(env))")
```

```
  try
    yield(Timeout(env, 3.0))
    yield(Release(res))
  catch exc
    println("$name got preempted at $(now(env))")
  end
end

env = Environment()
res = Resource(env, 1)
A = Process(env, user, "A", res, 0.0, 0, true)
run(env, 1.0)
B = Process(env, user, "B", res, 1.0, -2, false)
C = Process(env, user, "C", res, 2.0, -1, true)
run(env)
```

- Process `A` requests the resource with priority `0`. It immediately becomes a user.
- Process `B` requests the resource with priority `-2` but sets preempt to `false`. It will queue up and wait.
- Process `C` requests the resource with priority `-1` but sets preempt to `true`. Normally, it would preempt `A` but in this case, `B` is queued up before `C` and prevents `C` from preempting `A`. `C` can also not preempt `B` since its priority is not high enough.

Thus, the behavior in the example is the same as if no preemption was used at all. Be careful when using mixed preemption! Due to the higher priority of process `B`, no preemption occurs in this example. Note that an additional request with a priority of `-3` would be able to preempt `A`.

3.5.6 Containers

Containers help you modelling the production and consumption of a homogeneous, undifferentiated bulk. It may either be continuous (like water) or discrete (like apples).

You can use this, for example, to model the gas / petrol tank of a gas station. Tankers increase the amount of gasoline in the tank while cars decrease it.

The following example is a very simple model of a gas station with a limited number of fuel dispensers (modeled as :class:`Resource`) and a tank modeled as :class:`Container`:

```
using SimJulia

type GasStation
  fuel_dispensers :: Resource
  gas_tank :: Container{Float64}
  function GasStation(env::Environment)
    gs = new()
    gs.fuel_dispensers = Resource(env, 2)
    gs.gas_tank = Container{Float64}(env, 1000.0, 100.0)
    Process(env, monitor_tank, gs)
    return gs
  end
end

function monitor_tank(env::Environment, gs::GasStation)
  while true
    if level(gs.gas_tank) < 100.0
      println("Calling tanker at $(now(env))")
      Process(env, tanker, gs)
```

```
    end
    yield(Timeout(env, 15.0))
  end
end

function tanker(env::Environment, gs::GasStation)
  yield(Timeout(env, 10.0))
  println("Tanker arriving at $(now(env))")
  amount = capacity(gs.gas_tank) - level(gs.gas_tank)
  yield(Put(gs.gas_tank, amount))
end

function car(env::Environment, name::Int, gs::GasStation)
  println("Car $name arriving at $(now(env))")
  yield(Request(gs.fuel_dispensers))
  println("Car $name starts refueling at $(now(env))")
  yield(Get(gs.gas_tank, 40.0))
  yield(Timeout(env, 15.0))
  yield(Release(gs.fuel_dispensers))
  println("Car $name done refueling at $(now(env))")
end

function car_generator(env::Environment, gs::GasStation)
  for i = 0:3
    Process(env, car, i, gs)
    yield(Timeout(env, 5.0))
  end
end

env = Environment()
gs = GasStation(env)
Process(env, car_generator, gs)
run(env, 55.0)
```

The constructors *Put(cont::Container, amount::T, priority::Int=0)* and *Get(cont::Container, amount::T, priority::Int=0)* create respectively events to put and to get an amount of fuel. The function *level(cont::Container)* returns the amount of fuel still in the tank.

Priorities can be given to a put or a get event by setting the argument `priority`.

3.5.7 Stores

Using a `Store` you can model the production and consumption of concrete objects (in contrast to the rather abstract "amount" stored in a `Container`). A single `Store` can even contain multiple types of objects.

A custom function can also be used to filter the objects you get out of the store.

Here is a simple example modelling a generic producer/consumer scenario:

```
using SimJulia

function producer(env::Environment, sto::Store)
  for i = 1:100
    yield(Timeout(env, 2.0))
    yield(Put(sto, "spam $i"))
    println("Produced spam at $(now(env))")
  end
end
```

```
function consumer(env::Environment, name::Int, sto::Store)
  while true
    yield(Timeout(env, 1.0))
    println("$name requesting spam at $(now(env))")
    item = yield(Get(sto))
    println("$name got $item at $(now(env))")
  end
end

env = Environment()
sto = Store{ASCIIString}(env, 2)

prod = Process(env, producer, sto)
consumers = [Process(env, consumer, i, sto) for i=1:2]

run(env, 5.0)
```

As with the other resource types, you can get a store's capacity via the function *capacity(sto::Store)*. The function items(sto::Store) returns a Set of items currently available in the store.

A store with a filter on the *Get* event can, for example, be used to model machine shops where machines have varying attributes. This can be useful if the homogeneous slots of a *Resource* are not what you need:

```
using SimJulia

type Machine
  size :: Int
  duration :: Float64
end

function user(env::Environment, name::Int, sto::Store, size::Int)
  machine = yield(Get(sto, (mach::Machine)->mach.size == size))
  println("$name got $machine at $(now(env))")
  yield(Timeout(env, machine.duration))
  yield(Put(sto, machine))
  println("$name released $machine at $(now(env))")
end

function machineshop(env::Environment, sto::Store)
  m1 = Machine(1, 2.0)
  m2 = Machine(2, 1.0)
  yield(Put(sto, m1))
  yield(Put(sto, m2))
end

env = Environment()
sto = Store{Machine}(env, 2)
ms = Process(env, machineshop, sto)
users = [Process(env, user, i, sto, (i % 2) +1) for i=0:2]
run(env)
```

CHAPTER 4

Examples

In this chapter, various practical examples are presented that demonstrate how to uses SimJulia's features.

4.1 Bank Renege

Covers:

- Resources
- Event operators

A counter with a random service time and customers who renege.

This example models a bank counter and customers arriving at random times. Each customer has a certain patience. It waits to get to the counter until she's at the end of her tether. If she gets to the counter, she uses it for a while before releasing it.

New customers are created by the source process every few time steps.

```
using SimJulia
using Distributions

const RANDOM_SEED = 150
const NEW_CUSTOMERS = 5  # Total number of customers
const INTERVAL_CUSTOMERS = 10.0  # Generate new customers roughly every x seconds
const MIN_PATIENCE = 1.0  # Min. customer patience
const MAX_PATIENCE = 3.0  # Max. customer patience

function source(env::Environment, number::Int, interval::Float64, counter::Resource)
  d = Exponential(interval)
  for i in 1:number
    Process(env, customer, "Customer$i", counter, 12.0)
    yield(Timeout(env, rand(d)))
  end
end

function customer(env::Environment, name::ASCIIString, counter::Resource, time_in_bank::Float64)
  arrive = now(env)
  println("$arrive $name: Here I am")
  req = Request(counter)
  patience = rand(Uniform(MIN_PATIENCE, MAX_PATIENCE))
  result = yield(req | Timeout(env, patience))
  wait = now(env) - arrive
```

```
  if in(req, keys(result))
    println("$(now(env)) $name: Waited $wait")
    yield(Timeout(env, rand(Exponential(time_in_bank))))
    println("$(now(env)) $name: Finished")
    yield(Release(counter))
  else
    println("$(now(env)) $name: RENEGED after $wait")
    cancel(counter, req)
  end
end

# Setup and start the simulation
println("Bank renege")
srand(RANDOM_SEED)
env = Environment()

# Start processes and run
counter = Resource(env, 1)
Process(env, source, NEW_CUSTOMERS, INTERVAL_CUSTOMERS, counter)
run(env)
```

The simulation's output:

```
Bank renege
0.0 Customer1: Here I am
0.0 Customer1: Waited 0.0
4.435484832567573 Customer1: Finished
21.013085103081753 Customer2: Here I am
21.013085103081753 Customer2: Waited 0.0
23.097746900916643 Customer3: Here I am
23.91170855317896 Customer2: Finished
23.91170855317896 Customer3: Waited 0.8139616522623179
30.113622311091923 Customer4: Here I am
30.621135918022613 Customer5: Here I am
32.43509581615485 Customer5: RENEGED after 1.8139598981322358
32.63868913452709 Customer3: Finished
32.63868913452709 Customer4: Waited 2.525066823435168
35.25594434892944 Customer4: Finished
```

4.2 Movie Renege

Covers:

- Resources
- Event operators
- Shared events

This examples models a movie theater with one ticket counter selling tickets for three movies (next show only). People arrive at random times and try to buy a random number (1–6) tickets for a random movie. When a movie is sold out, all people waiting to buy a ticket for that movie renege (leave the queue).

The movie theater is just a type to assemble all the related data (movies, the counter, tickets left, collected data, ...). The counter is a `Resource` with a capacity of one.

The moviegoer process function starts waiting until either it's his turn (it acquires the counter resource) or until the sold out signal is triggered. If the latter is the case it reneges (leaves the queue). If it gets to the counter, it tries to buy

some tickets. This might not be successful, e.g. if the process tries to buy 5 tickets but only 3 are left. If less then two tickets are left after the ticket purchase, the sold out signal is triggered.

Moviegoers are generated by the customer arrivals process. It also chooses a movie and the number of tickets for the moviegoer.

```
using SimJulia
using Distributions
using Compat

const RANDOM_SEED = 158
const TICKETS = 50  # Number of tickets per movie
const SIM_TIME = 120.0  # Simulate until

# Create movie theater
type Theater
  movies :: Vector{ASCIIString}
  counter :: Resource
  available :: Dict{ASCIIString, Int}
  sold_out :: Dict{ASCIIString, Event}
  when_sold_out :: Dict{ASCIIString, Float64}
  num_renegers :: Dict{ASCIIString, Int}
  function Theater(env)
    theater = new()
    theater.movies = ASCIIString["Julia Unchained", "Kill Process", "Pulp Implementation"]
    theater.counter = Resource(env, 1)
    theater.available = @compat Dict("Julia Unchained" => TICKETS, "Kill Process" => TICKETS, "Pulp I
    theater.sold_out = @compat Dict("Julia Unchained" => Event(env), "Kill Process" => Event(env), "P
    theater.when_sold_out = @compat Dict("Julia Unchained" => typemax(Float64), "Kill Process" => typ
    theater.num_renegers = @compat Dict("Julia Unchained" => 0, "Kill Process" => 0, "Pulp Implementa
    return theater
  end
end

function moviegoer(env::Environment, movie::ASCIIString, num_tickets::Int, theater::Theater)
  req = Request(theater.counter)
  result = yield(req | theater.sold_out[movie])
  if in(theater.sold_out[movie], keys(result))
    theater.num_renegers[movie] += 1
    cancel(theater.counter, req)
  elseif theater.available[movie] < num_tickets
    yield(Timeout(env, 0.5))
    yield(Release(theater.counter))
  else
    theater.available[movie] -= num_tickets
    if theater.available[movie] < 2
      succeed(theater.sold_out[movie])
      theater.when_sold_out[movie] = now(env)
      theater.available[movie] = 0
    end
    yield(Timeout(env, 1.0))
    yield(Release(theater.counter))
  end
end

function customer_arrivals(env::Environment, theater::Theater)
  t = Exponential(0.5)
  d = DiscreteUniform(1, 3)
  n = DiscreteUniform(1, 6)
```

```
  while true
    yield(Timeout(env, rand(t)))
    movie = theater.movies[rand(d)]
    num_tickets = rand(n)
    if theater.available[movie] > 0
      Process(env, moviegoer, movie, num_tickets, theater)
    end
  end
end

# Setup and start the simulation
println("Movie renege")
srand(RANDOM_SEED)
env = Environment()
theater = Theater(env)

# Start process and run
Process(env, customer_arrivals, theater)
run(env, SIM_TIME)

# Analysis/results
for movie in theater.movies
  if processed(theater.sold_out[movie])
    println("Movie $movie sold out $(theater.when_sold_out[movie]) minutes after ticket counter openi
    println("  Number of people leaving queue when film sold out: $(theater.num_reneger$[movie])")
  end
end
```

The simulation's output:

```
Movie renege
Movie Julia Unchained sold out 47.08786185479453 minutes after ticket counter opening.
  Number of people leaving queue when film sold out: 17
Movie Kill Process sold out 38.08786185479453 minutes after ticket counter opening.
  Number of people leaving queue when film sold out: 17
Movie Pulp Implementation sold out 48.08786185479453 minutes after ticket counter opening.
  Number of people leaving queue when film sold out: 10
```

4.3 Machine Shop

Covers:

- Interrupts
- Resources

This example comprises a workshop with *n* identical machines. A stream of jobs (enough to keep the machines busy) arrives. Each machine breaks down periodically. Repairs are carried out by one repairman. The repairman has other, less important tasks to perform, too. Broken machines preempt theses tasks. The repairman continues them when he is done with the machine repair. The workshop works continuously.

A machine has two processes: working implements the actual behaviour of the machine (producing parts). break_machine periodically interrupts the working process to simulate the machine failure.

The repairman's other job is also a process (implemented by `other_job()`). The repairman itself is a *Resource* with a capacity of `1`. The machine repairing has a priority of `1`, while the other job has a priority of `2` (the smaller the number, the higher the priority).

```
using SimJulia
using Distributions

const RANDOM_SEED = 23062015
const PT_MEAN = 10.0          # Avg. processing time in minutes
const PT_SIGMA = 2.0          # Sigma of processing time
const MTTF = 300.0            # Mean time to failure in minutes
const REPAIR_TIME = 30.0      # Time it takes to repair a machine in minutes
const JOB_DURATION = 30.0     # Duration of other jobs in minutes
const NUM_MACHINES = 10       # Number of machines in the machine shop
const WEEKS = 4               # Simulation time in weeks
const SIM_TIME = WEEKS * 7 * 24 * 60.0  # Simulation time in minutes

type Machine
  name :: ASCIIString
  parts_made :: Int
  broken :: Bool
  proc :: Process
  function Machine(env::Environment, name::ASCIIString, repairman::Resource)
    mach = new()
    mach.name = name
    mach.parts_made = 0
    mach.broken = false
    mach.proc = Process(env, name, working, mach, repairman)
    Process(env, break_machine, mach)
    return mach
  end
end

function working(env::Environment, mach::Machine, repairman::Resource)
  d = Normal(PT_MEAN, PT_SIGMA)
  while true
    done_in = abs(rand(d))
    while done_in > 0.0
      start = now(env)
      try
        yield(Timeout(env, done_in))
        done_in = 0.0
      catch(interrupted)
        mach.broken = true
        done_in -= now(env) - start
        yield(Request(repairman, 1, true))
        yield(Timeout(env, REPAIR_TIME))
        yield(Release(repairman))
        mach.broken = false
      end
    end
    mach.parts_made += 1
  end
end

function break_machine(env::Environment, mach::Machine)
  d = Exponential(MTTF)
  while true
    yield(Timeout(env, rand(d)))
    if !mach.broken
      yield(Interrupt(mach.proc))
    end
```

```
  end
end

function other_jobs(env::Environment, repairman::Resource)
  while true
    done_in = JOB_DURATION
    while done_in > 0.0
      yield(Request(repairman, 2, false))
      start = now(env)
      try
        yield(Timeout(env, done_in))
        done_in = 0.0
        yield(Release(repairman))
      catch(preempted)
        done_in -= now(env) - start
      end
    end
  end
end

# Setup and start the simulation
println("Machine shop")
srand(RANDOM_SEED)

# Create an environment and start the setup process
env = Environment()
repairman = Resource(env, 1)
machines = [Machine(env, "Machine $i", repairman) for i = 1:NUM_MACHINES]
Process(env, other_jobs, repairman)

# Execute!
run(env, SIM_TIME)

# Analyis/results
println("Machine shop results after $WEEKS weeks")
for machine in machines
  println("$(machine.name) made $(machine.parts_made) parts.")
end
```

The simulation's output:

```
Machine shop
Machine shop results after 4 weeks
Machine 1 made 3258 parts.
Machine 2 made 3266 parts.
Machine 3 made 3264 parts.
Machine 4 made 3196 parts.
Machine 5 made 3286 parts.
Machine 6 made 3323 parts.
Machine 7 made 3233 parts.
Machine 8 made 3292 parts.
Machine 9 made 3201 parts.
Machine 10 made 3342 parts.
```

4.4 Event Latency

Covers:

Resources: Store

Scenario:

This example shows how to separate the time delay of events between processes from the processes themselves.

When Useful:

When modeling physical things such as cables, RF propagation, etc. it better encapsulation to keep this propagation mechanism outside of the sending and receiving processes.

Can also be used to interconnect processes sending messages.

```
using SimJulia

const SIM_DURATION = 100.0

type Cable
  env :: Environment
  delay :: Float64
  store :: Store{ASCIIString}
  function Cable(env::Environment, delay::Float64)
    cable = new()
    cable.env = env
    cable.delay = delay
    cable.store = Store{ASCIIString}(env)
    return cable
  end
end

function latency(env::Environment, cable::Cable, value::ASCIIString)
  yield(Timeout(env, cable.delay))
  yield(Put(cable.store, value))
end

function put(cable::Cable, value::ASCIIString)
  Process(cable.env, latency, cable, value)
end

function get(cable::Cable)
  return yield(Get(cable.store))
end

function sender(env::Environment, cable::Cable)
  while true
    yield(Timeout(env, 5.0))
    put(cable, "Sender send this at $(now(env))")
  end
end

function receiver(env::Environment, cable::Cable)
  while true
    msg = get(cable)
    println("Received this at $(now(env)) while $msg")
  end
```

```
end

println("Event latency")
env = Environment()

cable = Cable(env, 10.0)
Process(env, sender, cable)
Process(env, receiver, cable)

run(env, SIM_DURATION)
```

The simulation's output:

```
Event latency
Received this at 15.0 while Sender send this at 5.0
Received this at 20.0 while Sender send this at 10.0
Received this at 25.0 while Sender send this at 15.0
Received this at 30.0 while Sender send this at 20.0
Received this at 35.0 while Sender send this at 25.0
Received this at 40.0 while Sender send this at 30.0
Received this at 45.0 while Sender send this at 35.0
Received this at 50.0 while Sender send this at 40.0
Received this at 55.0 while Sender send this at 45.0
Received this at 60.0 while Sender send this at 50.0
Received this at 65.0 while Sender send this at 55.0
Received this at 70.0 while Sender send this at 60.0
Received this at 75.0 while Sender send this at 65.0
Received this at 80.0 while Sender send this at 70.0
Received this at 85.0 while Sender send this at 75.0
Received this at 90.0 while Sender send this at 80.0
Received this at 95.0 while Sender send this at 85.0
```

4.5 Gas Station Refueling

Covers:

- Resources: *Resource*
- Resources: Container
- Waiting for other processes

This examples models a gas station and cars that arrive at the station for refueling.

The gas station has a limited number of fuel pumps and a fuel tank that is shared between the fuel pumps. The gas station is thus modeled as a *Resource*. The shared fuel tank is modeled with a Container.

Vehicles arriving at the gas station first request a fuel pump from the station. Once they acquire one, they try to take the desired amount of fuel from the fuel pump. They leave when they are done.

The gas stations fuel level is requlary monitored by gas station control. When the level drops below a certain threshold, a tank truck is called to refuel the gas station itself.

```
using SimJulia
using Distributions

const RANDOM_SEED = 14021986
const GAS_STATION_SIZE = 200                    # liters
const THRESHOLD = 10                            # Threshold for calling the tank truck (in %)
```

```
const FUEL_TANK_SIZE = 50                           # liters
const FUEL_TANK_LEVEL = DiscreteUniform(5, 25)   # Min/max levels of fuel tanks (in liters)
const REFUELING_SPEED = 2.0                         # liters / second
const TANK_TRUCK_TIME = 300.0                       # Seconds it takes the tank truck to arrive
const T_INTER = Uniform(30.0, 300.0)                # Create a car every (min, max) seconds
const SIM_TIME = 2000.0                             # Simulation time in seconds

function car(env::Environment, name::ASCIIString, gas_station::Resource, fuel_pump::Container{Int})
  fuel_tank_level = rand(FUEL_TANK_LEVEL)
  println("$name arriving at gas station at $(round(now(env), 2)) with $fuel_tank_level liters left
  start = now(env)
  yield(Request(gas_station))
  liters_required = FUEL_TANK_SIZE - fuel_tank_level
  yield(Get(fuel_pump, liters_required))
  yield(Timeout(env, liters_required / REFUELING_SPEED))
  println("$name finished refueling in $(round((now(env)-start), 2)) seconds.")
  yield(Release(gas_station))
end

function gas_station_control(env::Environment, fuel_pump::Container{Int})
  while true
    if level(fuel_pump) / capacity(fuel_pump) * 100 < THRESHOLD
      println("Calling tank truck at $(round(now(env), 2)).")
      yield(Process(env, tank_truck, fuel_pump))
    end
    yield(Timeout(env, 10.0))  # Check every 10 seconds
  end
end

function tank_truck(env::Environment, fuel_pump::Container)
  yield(Timeout(env, TANK_TRUCK_TIME))
  println("Tank truck arriving at time $(round(now(env), 2)).")
  amount = capacity(fuel_pump) - level(fuel_pump)
  println("Tank truck refuelling $amount liters.")
  yield(Put(fuel_pump, amount))
end


function car_generator(env::Environment, gas_station::Resource, fuel_pump::Container{Int})
  i = 0
  while true
    yield(Timeout(env, rand(T_INTER)))
    Process(env, car, "Car $(i+=1)", gas_station, fuel_pump)
  end
end

# Setup and start the simulation
println("Gas Station refuelling")
srand(RANDOM_SEED)

# Create environment and start processes
env = Environment()
gas_station = Resource(env, 2)
fuel_pump = Container{Int}(env, GAS_STATION_SIZE, GAS_STATION_SIZE)
Process(env, gas_station_control, fuel_pump)
Process(env, car_generator, gas_station, fuel_pump)

# Execute!
```

```
run(env, SIM_TIME)
```

The simulation's output:

```
Gas Station refuelling
Car 1 arriving at gas station at 212.43 with 8 liters left in tank.
Car 1 finished refueling in 21.0 seconds.
Car 2 arriving at gas station at 482.13 with 22 liters left in tank.
Car 2 finished refueling in 14.0 seconds.
Car 3 arriving at gas station at 779.36 with 25 liters left in tank.
Car 3 finished refueling in 12.5 seconds.
Car 4 arriving at gas station at 964.75 with 17 liters left in tank.
Car 4 finished refueling in 16.5 seconds.
Car 5 arriving at gas station at 1011.92 with 9 liters left in tank.
Car 5 finished refueling in 20.5 seconds.
Car 6 arriving at gas station at 1121.88 with 20 liters left in tank.
Calling tank truck at 1130.0.
Car 6 finished refueling in 15.0 seconds.
Car 7 arriving at gas station at 1361.4 with 25 liters left in tank.
Tank truck arriving at time 1430.0.
Tank truck refuelling 199 liters.
Car 7 finished refueling in 81.1 seconds.
Car 8 arriving at gas station at 1605.04 with 19 liters left in tank.
Car 8 finished refueling in 15.5 seconds.
Car 9 arriving at gas station at 1890.62 with 14 liters left in tank.
Car 9 finished refueling in 18.0 seconds.
```

4.6 Carwash

Covers:

- Waiting for other processes
- Resources: *Resource*

The Carwash example is a simulation of a carwash with a limited number of machines and a number of cars that arrive at the carwash to get cleaned.

The carwash uses a *Resource* to model the limited number of washing machines. It also defines a process for washing a car.

When a car arrives at the carwash, it requests a machine. Once it got one, it starts the carwash's *wash* processes and waits for it to finish. It finally releases the machine and leaves.

The cars are generated by a *setup* process. After creating an intial amount of cars it creates new *car* processes after a random time interval as long as the simulation continues.

```
using SimJulia
using Distributions

const RANDOM_SEED = 09011977
const NUM_MACHINES = 2  # Number of machines in the carwash
const WASHTIME = 5.0    # Minutes it takes to clean a car
const T_INTER = 5.0     # Create a car every ~5 minutes
const SIM_TIME = 30.0   # Simulation time in minutes

function wash(env::Environment)
  yield(Timeout(env, WASHTIME))
```

```
end

function car(env::Environment, cw::Resource)
  println("$(active_process(env)) arrives at the carwash at time $(round(now(env), 2)).")
  yield(Request(cw))
  println("$(active_process(env)) enters the carwash at time $(round(now(env), 2)).")
  yield(Process(env, wash))
  println("$(active_process(env)) leaves the carwash at time $(round(now(env), 2)).")
  yield(Release(cw))
end

function setup(env::Environment)
  cw = Resource(env, NUM_MACHINES)
  for i = 1:4
    Process(env, "Car $i", car, cw)
  end
  d = Uniform(T_INTER-2.0, T_INTER+2.0)
  i = 4
  while true
    yield(Timeout(env, rand(d)))
    Process(env, "Car $(i+=1)", car, cw)
  end
end

# Setup and start the simulation
println("Carwash")
srand(RANDOM_SEED)

# Create an environment and start the setup process
env = Environment()
Process(env, setup)

# Execute!
run(env, SIM_TIME)
```

The simulation's output:

```
Carwash
Car 1 arrives at the carwash at time 0.0.
Car 2 arrives at the carwash at time 0.0.
Car 3 arrives at the carwash at time 0.0.
Car 4 arrives at the carwash at time 0.0.
Car 1 enters the carwash at time 0.0.
Car 2 enters the carwash at time 0.0.
Car 5 arrives at the carwash at time 3.89.
Car 1 leaves the carwash at time 5.0.
Car 2 leaves the carwash at time 5.0.
Car 3 enters the carwash at time 5.0.
Car 4 enters the carwash at time 5.0.
Car 6 arrives at the carwash at time 9.02.
Car 3 leaves the carwash at time 10.0.
Car 4 leaves the carwash at time 10.0.
Car 5 enters the carwash at time 10.0.
Car 6 enters the carwash at time 10.0.
Car 7 arrives at the carwash at time 12.56.
Car 5 leaves the carwash at time 15.0.
Car 6 leaves the carwash at time 15.0.
Car 7 enters the carwash at time 15.0.
Car 8 arrives at the carwash at time 17.14.
```

```
Car 8 enters the carwash at time 17.14.
Car 7 leaves the carwash at time 20.0.
Car 9 arrives at the carwash at time 21.49.
Car 9 enters the carwash at time 21.49.
Car 8 leaves the carwash at time 22.14.
Car 9 leaves the carwash at time 26.49.
Car 10 arrives at the carwash at time 26.51.
Car 10 enters the carwash at time 26.51.
```

4.7 Process Communication

Covers:

Resources: Store

This example shows how to interconnect simulation model elements together using a `Store` for one-to-one, and many-to-one asynchronous processes. For one-to-many a simple type `BroadCastPipe` is constructed from `Store`.

When Useful:

- When a consumer process does not always wait on a generating process and these processes run asynchronously. This example shows how to create a buffer and also tell is the consumer process was late yielding to the event from a generating process.
- This is also useful when some information needs to be broadcast to many receiving processes
- Finally, using pipes can simplify how processes are interconnected to each other in a simulation model.

```
using SimJulia
using Distributions

const RANDOM_SEED = 17102015
const SIM_TIME = 100.0

type Message
  timestamp :: Float64
  txt :: ASCIIString
end

type BroadcastPipe{T}
  env :: Environment
  capacity :: Int
  pipes :: Vector{Store{T}}
  function BroadcastPipe(env::Environment, capacity::Int=typemax(Int))
    bc_pipe = new()
    bc_pipe.env = env
    bc_pipe.capacity = capacity
    bc_pipe.pipes = Store{T}[]
    return bc_pipe
  end
end

function put{T}(bc_pipe::BroadcastPipe{T}, value::T)
  return AllOf(ntuple((i)->Put(bc_pipe.pipes[i], value), length(bc_pipe.pipes))...)
end

function get_output_conn{T}(bc_pipe::BroadcastPipe{T})
  pipe = Store{T}(bc_pipe.env, bc_pipe.capacity)
```

```
  push!(bc_pipe.pipes, pipe)
  return pipe
end

function message_generator(env::Environment, out_pipe::BroadcastPipe{Message})
  d = Uniform(6.0, 10.0)
  while true
    yield(Timeout(env, rand(d)))
    msg = Message(now(env), "$(active_process(env)) says hello at time $(round(now(env), 2))")
    yield(put(out_pipe, msg))
  end
end

function message_consumer(env::Environment, in_pipe::Store{Message})
  d = Uniform(4.0, 8.0)
  while true
    msg = yield(Get(in_pipe))
    if msg.timestamp < now(env)
      println("LATE getting message at time $(round(now(env), 2)): $(active_process(env)) received m
    else
      println("At time $(round(now(env), 2)): $(active_process(env)) received message: $(msg.txt)")
    end
    yield(Timeout(env, rand(d)))
  end
end

# Setup and start the simulation
println("Process communication")
srand(RANDOM_SEED)

env = Environment()
bc_pipe = BroadcastPipe{Message}(env)
Process(env, "Generator A", message_generator, bc_pipe)
Process(env, "Consumer A", message_consumer, get_output_conn(bc_pipe))
Process(env, "Consumer B", message_consumer, get_output_conn(bc_pipe))

run(env, SIM_TIME)
```

The simulation's output:

```
Process communication
At time 8.42: Consumer A received message: Generator A says hello at time 8.42
At time 8.42: Consumer B received message: Generator A says hello at time 8.42
At time 17.02: Consumer A received message: Generator A says hello at time 17.02
At time 17.02: Consumer B received message: Generator A says hello at time 17.02
At time 23.17: Consumer B received message: Generator A says hello at time 23.17
LATE getting message at time 24.57: Consumer A received message: Generator A says hello at time 23.1
At time 29.62: Consumer B received message: Generator A says hello at time 29.62
LATE getting message at time 30.9: Consumer A received message: Generator A says hello at time 29.62
At time 37.04: Consumer B received message: Generator A says hello at time 37.04
LATE getting message at time 38.02: Consumer A received message: Generator A says hello at time 37.04
LATE getting message at time 44.55: Consumer B received message: Generator A says hello at time 43.24
LATE getting message at time 45.4: Consumer A received message: Generator A says hello at time 43.24
LATE getting message at time 50.78: Consumer B received message: Generator A says hello at time 50.63
LATE getting message at time 52.25: Consumer A received message: Generator A says hello at time 50.63
LATE getting message at time 57.61: Consumer B received message: Generator A says hello at time 57.47
LATE getting message at time 58.51: Consumer A received message: Generator A says hello at time 57.47
At time 64.82: Consumer B received message: Generator A says hello at time 64.82
```

```
LATE getting message at time 65.0: Consumer A received message: Generator A says hello at time 64.82
At time 71.76: Consumer A received message: Generator A says hello at time 71.76
LATE getting message at time 72.64: Consumer B received message: Generator A says hello at time 71.76
At time 81.03: Consumer A received message: Generator A says hello at time 81.03
At time 81.03: Consumer B received message: Generator A says hello at time 81.03
At time 89.32: Consumer A received message: Generator A says hello at time 89.32
At time 89.32: Consumer B received message: Generator A says hello at time 89.32
At time 96.33: Consumer A received message: Generator A says hello at time 96.33
LATE getting message at time 96.36: Consumer B received message: Generator A says hello at time 96.33
```

4.8 Repair Problem

Covers:

- Resources: *Resource*
- Resources: Store
- Interrupts

A system needs n working machines to be operational. To guard against machine breakdown, additional machines are kept available as spares. Whenever a machine breaks down it is immediately replaced by a spare and is itself sent to the repair facility, which consists of a single repairperson who repairs failed machines one at a time. Once a failed machine has been repaired it becomes available as a spare to be used when the need arises. All repair times are independent random variables having a common exponential distribution function. Each time a machine is put into use the amount of time it functions before breaking down is a random variable, independent of the past, having an exponential distribution function.

The system is said to "crash" when a machine fails and no spares are available. Assuming that there are initially n + s functional machines of which n are put in use and s are kept as spares, we are interested in simulating this system so as to approximate E[T], where T is the time at which the system crashes.

```
using SimJulia
using Distributions

const RUNS = 100
const N = 10
const S = 3
const SEED = 150
const LAMBDA = 100
const MU = 1

function work(env::Environment, repair_facility::Resource, spares::Store{Process})
  dist_work = Exponential(LAMBDA)
  dist_repair = Exponential(MU)
  while true
    try
      yield(Timeout(env, Inf))
    catch(exc)
    end
    println("At time $(now(env)): $(active_process(env)) starts working.")
    yield(Timeout(env, rand(dist_work)))
    println("At time $(now(env)): $(active_process(env)) stops working.")
    get_spare = Get(spares)
    res = yield(get_spare | Timeout(env, 0.0))
    if in(get_spare, keys(res))
      yield(Interrupt(res[get_spare]))
```

```
    else
      stop_simulation(env)
    end
    yield(Request(repair_facility))
    println("At time $(now(env)): $(active_process(env)) repair starts.")
    yield(Timeout(env, rand(dist_repair)))
    yield(Release(repair_facility))
    println("At time $(now(env)): $(active_process(env)) is repaired.")
    yield(Put(spares, active_process(env)))
  end
end

function start_sim(env::Environment, repair_facility::Resource, spares::Store{Process})
  procs = Process[]
  for i=1:N
    push!(procs, Process(env, "Machine $i", work, repair_facility, spares))
  end
  yield(Timeout(env, 0.0))
  for proc in procs
    yield(Interrupt(proc))
  end
  for i=1:S
    yield(Put(spares, Process(env, "Machine $(i+10)", work, repair_facility, spares)))
  end
end

function sim_repair()
  env = Environment()
  repair_facility = Resource(env)
  spares = Store{Process}(env)
  Process(env, start_sim, repair_facility, spares)
  run(env)
  now(env)
end

srand(SEED)
results = Float64[]
for i=1:RUNS
  push!(results, sim_repair())
end
println(sum(results)/RUNS)
```

The simulation's output:

```
...
At time 10746.862481297383: Machine 9 starts working.
At time 10746.862481297383: Machine 7 repair starts.
At time 10748.673383437574: Machine 7 is repaired.
At time 10760.598516359223: Machine 10 stops working.
At time 10760.598516359223: Machine 7 starts working.
At time 10760.598516359223: Machine 10 repair starts.
At time 10761.127926380934: Machine 10 is repaired.
At time 10763.742027509461: Machine 1 stops working.
At time 10763.742027509461: Machine 10 starts working.
At time 10763.742027509461: Machine 1 repair starts.
At time 10763.940397277867: Machine 12 stops working.
At time 10763.940397277867: Machine 2 starts working.
At time 10764.498080704856: Machine 4 stops working.
At time 10764.498080704856: Machine 6 starts working.
```

```
At time 10764.703085034163: Machine 6 stops working.
11685.41156141544
```

CHAPTER 5

API Reference

The API reference provides detailed descriptions of SimJulia's classes and functions. It should be helpful if you plan to extend SimJulia with custom components.

5.1 SimJulia

The `SimJulia` module aggregates SimJulia's most used components into a single namespace.

The following tables list all of the available types in this module.

5.1.1 Environment

`AbstractEnvironment`	Parent type for an environment.
`Environment`	Execution environment for a simulation.

5.1.2 Events

`AbstractEvent`	Parent type for all events.
`Event`	An event that may happen at some point in time.
`Timeout`	An event that is triggered after a *delay* has passed.
`EventOperator`	An event that is triggered if an *eval* functions returns true on a tuple of events.

5.1.3 Processes

`Process`	A model that is implemented by a process function yielding events.
	An event that is triggered if the process function returns.
`Initialize`	An event that is triggered automatically to start the process function.
`Interrupt`	An event that is triggered immediately and that interrupts another process.
`Interruption`	An event that is triggered with priority and has an `InterruptException` value.

5.1.4 Resources

Resource	Resource with a *capacity* of usage slots that can be requested by processes.
Container	Resource containing up to a *capacity* of matter which may either be continuous or discrete.
Store	Resource with a *capacity* of slots for storing arbitrary objects.
PutEvent	An event that is triggered if the *put* action of a resource has been executed.
GetEvent	An event that is triggered if the *get* action of a resource has been executed.
Preempted	A type that contains the *cause* and the *usage time* of a preemption on a *Resource*.

5.1.5 Exceptions

EmptySchedule	An exception that is thrown if the scheduler contains no events.
StopSimulation	An exception that stops the simulation when it is thrown.
EventTriggered	An exception that is thrown if an already triggered event is triggered again.
EventProcessed	An exception that is thrown if a *callback* is added to a processed event.
InterruptException	An exception that is thrown if an *interrupt* occurs.

5.2 Environment

5.2.1 AbstractEnvironment

abstract AbstractEnvironment

Parent type for event processing environments.

An implementation must at least provide the means to access the current time of the environment (see now), to process events (see step and peek) and to give a reference to the active process (see active_process).

The class is meant to be subclassed for different execution environments. For example, SimJulia defines a *Environment* for simulations with a virtual time.

run (*env::AbstractEnvironment*) → nothing

Executes the step function until there are no further events to be processed.

run (*env::AbstractEnvironment*, *until::Float64*) → nothing

Executes the step function until the environment's time reaches *until*.

run (*env::AbstractEnvironment*, *until::AbstractEvent*) → Any

Executes the step function until the *until* event has been triggered and will return its *value*.

stop_simulation (*env::AbstractEnvironment*, *value=nothing*)

Stops the simulation, optionally providing an alternative return value to the run function.

5.2.2 Environment

Environment <: AbstractEnvironment

Execution environment for a simulation. The passing of time is simulated by stepping from event to event.

Environment (*initial_time::Float64=0.0*) → Environment

Constructor of *Environment*. An *initial_time* for the environment can be specified. By default, it starts at 0.0.

now(*env::Environment*) → Float64

Returns the current simulation time.

active_process(*env::Environment*) → Process

Returns the active process. If no process is active throws a `NullException`.

5.3 Events

5.3.1 AbstractEvent

abstract AbstractEvent

The parent type for all events is `AbstractEvent`.

An event:

- may happen (`triggered` returns `false`),
- is going to happen (`triggered` returns `true`),
- is happening (`processed` returns `false`) or
- has happened (`processed` returns `true`).

Every event is bound to an environment and is initially not triggered. Events are scheduled for processing by the environment after they are triggered by either `succeed`, `fail` or `trigger`. These methods also set the value of the event.

An event has a list of callbacks. A callback can be any function as long as it accepts an instance of type *Event* as its first argument. Once an event gets processed, all callbacks will be invoked. Callbacks can do further processing with the value it has produced.

Failed events are never silently ignored and will raise an exception upon being processed.

triggered(*ev::AbstractEvent*) → Bool

Returns `true` if the event has been triggered and its callbacks are about to be invoked.

processed(*ev::AbstractEvent*) → Bool

Returns `true` if the event has been processed (i.e., its callbacks have been invoked).

value(*ev::AbstractEvent*) → Any

Returns the `value` of the event if it is available, otherwise returns `nothing`. The value is available when the event has been triggered.

append_callback(*ev::AbstractEvent*, *callback::Function*, *args...*)

Adds a process function to the event. The first argument of the function `callback` is an `AbstractEnvironment`. Optional arguments can be specified by `args...`. If the event is already processed an `EventProcessed` exception is thrown.

succeed(*ev::AbstractEvent*, *value=nothing*) → AbstractEvent

Sets the event's *value* and schedule it for processing by the environment. Returns the event instance. Throws an `EventTriggered` exception if this event has already been triggered.

fail(*ev::AbstractEvent*, *exc::Exception*) → AbstractEvent

Sets the exception as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance. Throws an `EventTriggered` exception if this event has already been triggered.

trigger(*cause::AbstractEvent*, *ev::AbstractEvent*) → AbstractEvent

Schedules the event with the state and value of the *cause* event. Returns the event instance. Throws an `EventTriggered` exception if this event has already been triggered. This method can be used directly as a callback function to trigger chain reactions.

5.3.2 Event

Event <: AbstractEvent

An event that may happen at some point in time.

Event(*env::AbstractEnvironment*) → Event

Constructor of *Event* with one argument `env`, the environment where the event lives in.

5.3.3 Timeout

Timeout <: AbstractEvent

An event that gets triggered after a `delay` has passed.

Timeout(*env::AbstractEnvironment*, *delay::Float64*, *value=nothing*) → Timeout

This event is automatically triggered when it is created. The `value` argument is optional.

5.3.4 EventOperator

EventOperator <: AbstractEvent

An event that gets triggered once the condition function `eval` returns `true` on the given list of `events`.

The value of an Eventoperator is an instance of `Dict{AbstractEvent, Any}` which allows convenient access to the input events and their values. The value will only contain entries for those events that occurred before the condition is processed. If one of the events fails, the condition also fails and forwards the exception of the failing event.

EventOperator(*eval::Function*, *events...*) → EventOperator

The `eval` function receives a tuple of target events: `eval(events...)`. If it returns `true`, the event is triggered.

AllOf(*events...*) → EventOperator

Constructor for an *EventOperator* that is triggered if all of a list of events have been successfully triggered. Fails immediately if any of `events` failed.

AnyOf(*events...*) → EventOperator

Constructor for an *EventOperator* that is triggered if any of a list of events has been successfully triggered. Fails immediately if any of `events` failed.

(&)(ev1::AbstractEvent, ev2::AbstractEvent) -> EventOperator

Shortcut for *AllOf(ev1, ev2)*.

(|)(ev1::AbstractEvent, ev2::AbstractEvent) -> EventOperator

Shortcut for *AnyOf(ev1, ev2)*.

5.4 Processes

5.4.1 Process

Process <: AbstractEvent

A *Process* is an abstraction for an event yielding function, a process function.

The process function can suspend its execution by yielding an `AbstractEvent`. The *Process* will take care of resuming the process function with the value of that event once it has happened. The exception of failed events is also thrown into the process function.

A *Process* itself is an event, too. It is triggered, once the process functions returns or raises an exception. The value of the process is the return value of the process function or the exception, respectively.

Process(*env::AbstractEnvironment*, *func::Function*, *args...*) → Process

Process(*env::AbstractEnvironment*, *name::ASCIIString*, *func::Function*, *args...*) → Process

Constructs a *Process*. The argument `func` is the process function and has the following signature :func:`func(env::AbstractEnvironment, args...)` <func>. If the `name` argument is missing, the name of the process is a combination of the name of the process function and the event id of the process. An `Initialize` event is scheduled immediately to start the process function.

DelayedProcess(*env::AbstractEnvironment*, *delay::Float64*, *func::Function*, *args...*) → Process

Constructs a delayed *Process*. A *Timeout* event is scheduled with the specified `delay`. The process function is started from a callback of the timeout event.

yield(*ev::AbstractEvent*) → Any

Passes the control flow back to the simulation. If the yielded event is triggered , the simulation will resume the function after this statement. The return value is the value from the yielded event.

is_process_done(*proc::Process*) → Bool

Returns `true` if the process function returned or an exception was thrown.

5.4.2 Initialize

Initialize <: AbstractEvent

Start a process function. Only used internally by *Process*. This event is automatically triggered when it is created.

5.4.3 Interrupt

Interrupt <: AbstractEvent

Interrupt(*proc::Process*, *cause::Any=nothing*) → Interruption

Immediately schedules an `Interruption` event with as value an instance of `InterruptException`. The process function of `proc` is added to its callbacks. An *Interrupt* event is returned. This event is automatically triggered when it is created.

5.4.4 Interruption

Interruption <: AbstractEvent

Constructs an interruption event. Only used internally by *Interrupt*. This event is automatically triggered with priority when it is created.

5.5 Resources

SimJulia implements three types of resources that can be used to synchronize processes or to model congestion points:

- *Resource*: shared resources supporting priorities and preemption.
- Container: resource for sharing homogeneous matter between processes, either continuous or discrete.
- Store: shared resources for storing a possibly unlimited amount of objects supporting requests for specific objects.

5.5.1 AbstractResource

abstract AbstractResource

All resource types are derived from the abstract type AbstractResource. Common functions to all types of resources can be found in *resources\base.jl*. These are also meant to support the implementation of custom resource types.

capacity(*res::AbstractResource*)

Returns the capacity of the resource.

5.5.2 ResourceEvent

abstract ResourceEvent <: AbstractEvent

All events related to resources are derived from the abstract type ResourceEvent.

abstract PutEvent <: ResourceEvent

Abstract event for requesting to put something into the resource.

cancel(*ev::PutEvent*)

Cancel the put request ev.

This method has to be called if the put request must be aborted, for example if a process needs to handle an exception like an Interruption.

abstract GetEvent <: ResourceEvent

Generic event for requesting to get something from the resource.

cancel(*ev::GetEvent*)

Cancel this get request ev.

This method has to be called if the get request must be aborted, for example if a process needs to handle an exception like an Interruption.

5.5.3 Resource

`Resource <: AbstractResource`

Shared resources supporting priorities and preemption.

These resources can be used to limit the number of processes using them concurrently. A process needs to *request* the usage right to a resource. Once the usage right is not needed anymore it has to be *released*. A gas station can be modelled as a resource with a limited amount of fuel-pumps. Vehicles arrive at the gas station and request to use a fuel-pump. If all fuel-pumps are in use, the vehicle needs to wait until one of the users has finished refueling and releases its fuel-pump.

These resources can be used by a limited number of processes at a time. Processes request these resources to become a *user* and have to release them once they are done. For example, a gas station with a limited number of fuel pumps can be modeled with a Resource. Arriving vehicles request a fuel-pump. Once one is available they refuel. When they are done, the release the fuel-pump and leave the gas station.

Requesting a resource is modelled as "putting a process' token into the resources" and releasing a resources correspondingly as "getting a process' token out of the resource". Note, that releasing a resource will always succeed immediately, no matter if a process is actually using a resource or not.

`Resource`(*env::AbstractEnvironment*, *capacity::Int=1*) → Resource

Resource with `capacity` of usage slots that can be requested by processes. If all slots are taken, requests are enqueued. Once a usage request is released, a pending request will be triggered. The `env` argument is the `AbstractEnvironment` instance the resource is bound to.

`count`(*res::Resource*) → Int

Returns the number of users currently using `res`.

5.5.4 ResourcePut

`ResourcePut <: PutEvent`

Subtype of `PutEvent` for requesting to put something in a *`Resource`*.

`Put`(*res::Resource*, *priority::Int=0*, *preempt::Bool=false*) → ResourcePut

`Request`(*res::Resource*, *priority::Int=0*, *preempt::Bool=false*) → ResourcePut

Request usage of the *`Resource`* with a given `priority`. The event is triggered once access is granted.

If the maximum capacity of users has not yet been reached, the request is triggered immediately. If the maximum capacity has been reached, the request is triggered once an earlier usage request on the resource is released. If `preempt` is `true` other usage requests of the resource may be preempted.

5.5.5 ResourceGet

`ResourceGet <: GetEvent`

Subtype of `GetEvent` for requesting to get something from a *`Resource`*.

`Get`(*res::Resource*) → ResourceGet

`Release`(*res::Resource*) → ResourceGet

Releases the usage of `resource` by the active process. This event is triggered immediately.

5.5.6 Preempted

`Preempted`

Cause of a preemption `Interruption` containing information about the preemption.

`by` (*pre::Preempted*) → Process

Returns the preempting *`Process`*.

`usage_since` (*pre::Preempted*) → Float64

Returns the simulation time at which the preempted process started to use the resource.

5.5.7 Container

`Container{T<:Number} <: AbstractResource`

Resource for sharing homogeneous matter between processes, either continuous (like water) or discrete (like apples).

A `Container` can be used to model the fuel tank of a gasoline station. Tankers increase and refuelled cars decrease the amount of gas in the station's fuel tanks.

`Container{T}` (*env::Environment*, *capacity::T=typemax(T)*, *level::T=zero(T)*) → Container{T}

Resource containing up to capacity of matter which may either be continuous (like water) or discrete (like apples). It supports requests to put or get matter into/from the container.

The `env` argument is the `AbstractEnvironment` instance the container is bound to.

The `capacity` defines the size of the container.The initial amount of matter is specified by `level` and defaults to `zero(T)`.

`level` (*cont::Container*) → T

Returns the current amount of the matter in the container.

5.5.8 ContainerPut

`ContainerPut <: PutEvent`

Subtype of `PutEvent` for requesting to put something in a `Container`.

`Put{T<:Number}` (*cont::Container{T}*, *amount::T*, *priority::Int=0*) → ContainerPut

Request to put `amount` of matter into the container with a given `priority`. The request will be triggered once there is enough space in the container available.

5.5.9 ContainerGet

`ContainerGet <: GetEvent`

Subtype of `GetEvent` for requesting to get something from a `Container`.

`Get{T<:Number}` (*cont::Container{T}*, *amount::T*, *priority::Int=0*) → ContainerGet

Request to get `amount` of matter from the container with a given `priority`. The request will be triggered once there is enough matter available in the container.

5.5.10 Store

`Store{T} <: AbstractResource`

Shared resources for storing a possibly unlimited amount of objects supporting requests for specific objects.

The `Store` operates in a FIFO (first-in, first-out) order. Objects are retrieved from the store in the order they were put in. The get requests can be customized by a filter to only retrieve objects matching a given criterion.

`Store{T}`(*env::Environment*, *capacity::Int=typemax(Int)*) → Store{T}

Resource with capacity slots for storing arbitrary objects. By default, the capacity is unlimited and objects are put and retrieved from the store in a first-in first-out order.

The env argument is the `AbstractEnvironment` instance the store is bound to.

5.5.11 StorePut

`StorePut{T} <: PutEvent`

Subtype of `PutEvent` for requesting to put something in a `Store`.

`Put{T}`(*sto::Store{T}*, *item::T*, *priority::Int=0*) → StorePut

Request to put `item` into the store with a given `priority`. The request is triggered once there is space for the item in the store.

5.5.12 StoreGet

`StoreGet <: GetEvent`

Subtype of `GetEvent` for requesting to get something from a `Store`.

`Get{T}`(*sto::Store{T}*, *filter::Function=(item::T)* → true, priority::Int=0)

Request to get an item from the store matching the `filter` with a `priority`. The request is triggered once there is such an item available in the store.

`filter` is a function receiving one item. It should return `true` for items matching the filter criterion. The default function returns `true` for all items

5.6 Exceptions

5.6.1 EmptySchedule

`EmptySchedule <: Exception`

An exception that is thrown if the scheduler contains no events. Only used internally.

5.6.2 StopSimulation

`StopSimulation <: Exception`

An exception that stops the simulation when it is thrown.

5.6.3 EventTriggered

`EventTriggered <: Exception`

An exception that is thrown if an already triggered event is triggered again. Only used internally.

5.6.4 EventProcessed

`EventProcessed <: Exception`

An exception that is thrown if a *callback* is added to a processed event. Only used internally.

5.6.5 InterruptException

`InterruptException <: Exception`

An exception that is thrown if an *interrupt* occurs.

`cause`(*inter::InterruptException*) → Any

Returns the cause of the interrupt exception.

5.7 Low Level API

`step`(*env::Environment*)

Processes the next event.

`peek`(*env::Environment*) → Float64

Returns the next event time.

`schedule`(*ev::AbstractEvent*, *priority::Bool*, *delay::Float64*, *value=nothing*)

`schedule`(*ev::AbstractEvent*, *priority::Bool*, *value=nothing*)

`schedule`(*ev::AbstractEvent*, *delay::Float64*, *value=nothing*)

`schedule`(*ev::AbstractEvent*, *value=nothing*)

Schedules an event with a `value`, a `delay` and `priority`.