Dymodetron

Dyemm O'Detron

Jul 16, 2021

CONTENTS:

DY NAMIC **MODE** L **TRON**

Dymodetron is for building and analyzing dynamic models efficiently.

CHAPTER

ONE

QUICK START

The figure below illustrates the Dymodetron user workflow. There are two loops oriented around the `Build Model` activity.

In this section, we'll take a pre-built example model, and do the following:

1.1 Install Dymodetron

```
$ pip install git+https://github.com/InstituteforDiseaseModeling/dymodetron.git
↪#egg=dymodetron
```

If you've already installed an earlier version, you can upgrade to a more recent version by specifying the version number at the end, as follows:

```
$ pip install git+https://github.com/InstituteforDiseaseModeling/dymodetron.git
↪#egg=dymodetron==0.2.11
```

1.2 Generate Model Diagrams

This bash command generates diagrams for *the model below*.

```
$ python -m dymodetron.generators.state_machine_diagrams --model_definition_
↪file=examples/mm1_queue.py --overwrite_existing=1
state_machine_diagrams
Generating state machine diagram for state machine [mm1_queue_arrivals]
Outputting to [generated/state_machine_diagrams/mm1_queue_arrivals.html]
Generating state machine diagram for state machine [mm1_queue_service]
Outputting to [generated/state_machine_diagrams/mm1_queue_service.html]
```

1.3 View Model Diagrams

The command above listed out the locations of the generated diagram files. Take note of the folder that the files are generated into. Open the listed `.html` files in a browser.

This example model has two state machines, each one gets its own diagram file:

`mm1_queue_arrivals.html`:

`mm1_queue_service.html`:

The diagrams provide a good picture of the "bones" of the model, but they don't provide all the details. For that you have to look at *the model definition*. It's often useful to have the model diagrams open side-by-side with the model definition, and look at both together.

1.4 Generate Model Simulation Code

This bash command generates simulation code for the same model.

```
$ python -m dymodetron.generators.python_arrayified --model_definition_file=examples/mm1_
↪queue.py --overwrite_existing=1
python_arrayified
Generating code for entity type [MM1_Queue]
Outputting to [generated/python_arrayified/mm1_queue.py]
```

The command above listed the path of the generated simulation code. Take note of that path, that is the code you will use next to run the simulation.

1.5 Run Simulation

The following command illustrates how to run the simulation with two simulated entities for 100 time 'ticks'.

Note: Dymodetron measures time in units of 'ticks'. Time 'ticks' are whatever units of time you want them to be. Just be sure to be consistent in your use of time units throughout your model.

```
$ LOGLEVEL=INFO python generated/python_arrayified/mm1_queue.py --enable_logging=1 --end_
↪time_ticks=100 --progress_interval_ticks=10 --num_entities=2
```

The simulation produces the output below. Some highlighted lines are described in the next section.

```
Sim Arguments:
{
  'enable_logging': 1,
  'end_time_ticks': 100.0,
  'num_entities': 2,
  'progress_interval_ticks': 10.0,
  'rand_seed': 1,
}

INFO:__main__:t = 0.0
INFO:__main__:t = 10.016675542743428
INFO:__main__:t = 20.027835278247828
INFO:__main__:t = 30.055466614166935
INFO:__main__:t = 40.07540843143107
INFO:__main__:t = 50.12466144061003
INFO:__main__:t = 60.12856024231663
INFO:__main__:t = 70.14824380249689
INFO:__main__:t = 80.15701700954823
INFO:__main__:t = 90.25112781277566

Runtime: [1.460378399999172 s]

----------------------------------
State Tracking
----------------------------------
instance(Entities_MM1_Queue_state_tracking):
  mm1_queue_arrivals__arriving: StateTracking(track_cumulative=None),
  mm1_queue_arrivals__idle: StateTracking(track_cumulative=None),
  mm1_queue_service__idle: StateTracking(track_cumulative=None),
  mm1_queue_service__servicing: StateTracking(track_cumulative=None)

Writing entity attribute history to [entity_attribute_history.nc]

----------------------------------
Entity Attribute History Summary over ['time']
----------------------------------

-------------------- mean --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
Data variables:
    num_waiting  (entity_id) float64 0.3952 0.3241
------------------------------------------------------------

-------------------- std --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
```

(continues on next page)

(continued from previous page)

```
Data variables:
    num_waiting  (entity_id) float64 0.7688 0.6215
------------------------------------------------------------

-------------------- min --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
    quantile     float64 0.0
Data variables:
    num_waiting  (entity_id) float64 0.0 0.0
------------------------------------------------------------

-------------------- 0.25 quantile --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
    quantile     float64 0.25
Data variables:
    num_waiting  (entity_id) float64 0.0 0.0
------------------------------------------------------------

-------------------- 0.50 quantile --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
    quantile     float64 0.5
Data variables:
    num_waiting  (entity_id) float64 0.0 0.0
------------------------------------------------------------

-------------------- 0.75 quantile --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
    quantile     float64 0.75
Data variables:
    num_waiting  (entity_id) float64 1.0 0.5
------------------------------------------------------------

-------------------- max --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
    quantile     float64 1.0
Data variables:
    num_waiting  (entity_id) float64 5.0 4.0
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------



-----------------------------------
Entity Attribute History Summary over ['time', 'entity_id']
-----------------------------------

-------------------- mean --------------------
<xarray.Dataset>
Dimensions:      ()
Data variables:
    num_waiting  float64 0.3597
------------------------------------------------------------

-------------------- std --------------------
<xarray.Dataset>
Dimensions:      ()
Data variables:
    num_waiting  float64 0.6999
------------------------------------------------------------

-------------------- min --------------------
<xarray.Dataset>
Dimensions:      ()
Coordinates:
    quantile     float64 0.0
Data variables:
    num_waiting  float64 0.0
------------------------------------------------------------

-------------------- 0.25 quantile --------------------
<xarray.Dataset>
Dimensions:      ()
Coordinates:
    quantile     float64 0.25
Data variables:
    num_waiting  float64 0.0
------------------------------------------------------------

-------------------- 0.50 quantile --------------------
<xarray.Dataset>
Dimensions:      ()
Coordinates:
    quantile     float64 0.5
Data variables:
    num_waiting  float64 0.0
------------------------------------------------------------

-------------------- 0.75 quantile --------------------
<xarray.Dataset>
Dimensions:      ()
Coordinates:
```

(continues on next page)

(continued from previous page)

```
    quantile     float64 0.75
Data variables:
    num_waiting  float64 1.0
------------------------------------------------------------

-------------------- max --------------------
<xarray.Dataset>
Dimensions:      ()
Coordinates:
    quantile     float64 1.0
Data variables:
    num_waiting  float64 5.0
------------------------------------------------------------
```

1.6 Analyze Results

1.6.1 Entity Attribute History Summary

The model defines a single attribute named `num_waiting`, the number of items waiting in the queue at any one time. The model defines this attribute with `probe=True`, which causes it to be recorded at each event time while the simulation runs.

The highlighted lines in the `mean` and `std` blocks above illustrate the time-averaged mean and standard deviation of `num_waiting` for each entity in the simulation. We've run the simulation with two entities, so there are two entries for `entity_id` and `num_waiting` in each block.

Our model is configured with an interarrival rate of 3 and a service rate of 12. M/M/1 Queue theory says that we should expect the mean number of waiting queue requests to be `0.33`, with a standard deviation of `0.66`. You can see in the entity attribute history summary listed above that the simulated results match the theory well.

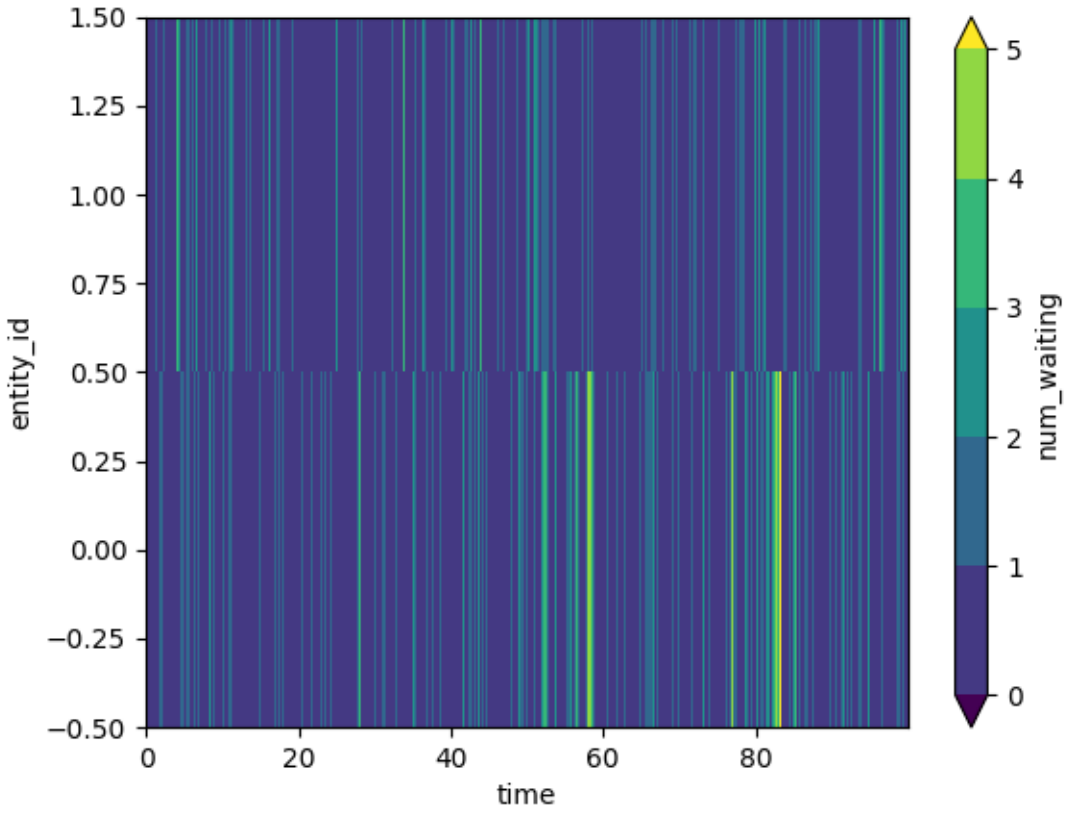
There are two `Entity Attribute History Summary` sections. One displays the results for each entity, aggregated over all simulated time. The other aggregates over all time and all entities.

Another highlighted output indicates that entity attribute data is written to the file `entity_attribute_history.nc` This is a NetCDF file containing the time-series history of the `num_waiting` entity attribute.

The section labeled `State Tracking` has a number of items listed as `None`, because this model doesn't have state tracking enabled.

1.6.2 Entity Attribute History Plot

The simulation also generates a file named `num_waiting_vs_entities_time.png`. This is a 2-d plot of the value of the `num_waiting` attribute. The y axis is entity ID. The x axis is time. We ran this example with 2 entities, their ids are 0 and 1 in the plot below.



1.6.3 Output Data File

The simulation generates an output file named `entity_attribute_history.nc`. This is a NetCDF file. There are a lot of tools and libraries for accessing and using this data.

One example is the python library netCDF4. Below is an example of programmatically examining the file generated by this example.

```
$ pip install netCDF4
$ python -c 'import netCDF4; d = netCDF4.Dataset("entity_attribute_history.nc", "r");␣
↪print(d); print(d.variables["num_waiting"]);'

<class 'netCDF4._netCDF4.Dataset'>
root group (NETCDF4 data model, file format HDF5):
    dimensions(sizes): time(3039), entity_id(2)
    variables(dimensions): float64 time(time), int64 entity_id(entity_id), int64 num_
↪waiting(time, entity_id)
    groups:
<class 'netCDF4._netCDF4.Variable'>
int64 num_waiting(time, entity_id)
unlimited dimensions:
current shape = (3039, 2)
filling on, default _FillValue of -9223372036854775806 used
```

The highlighted lines above tell you that this data has two dimensions, `time` and `entity_id`, a variable named `num_waiting` along those dimensions, that there are `30258` time coordinates, and `2` entities.

You can use netCDF4 or one of the many other libraries in various languages listed here to access the data for analysis.

The list of tools also has a number of GUI applications that you can use to explore and plot the simulation data in the NetCDF4 file. An especially useful one is Panoply.

1.7 Iterate on Model Parameters

We skipped this *before*, but you can ask the simulation to tell you what command line arguments it accepts by passing `--help`:

```
$ python generated/python_arrayified/mm1_queue.py --help

usage: mm1_queue.py [-h] [--num_entities NUM_ENTITIES] [--enable_logging ENABLE_LOGGING]␣
↪[--rand_seed RAND_SEED] [--end_time_ticks END_TIME_TICKS] [--progress_interval_ticks␣
↪PROGRESS_INTERVAL_TICKS]
                    [--PARAM_interarrival_time_distribution_rate PARAM_INTERARRIVAL_TIME_
↪DISTRIBUTION_RATE] [--PARAM_service_time_distribution_rate PARAM_SERVICE_TIME_
↪DISTRIBUTION_RATE]

optional arguments:
  -h, --help            show this help message and exit
  --num_entities NUM_ENTITIES
                        Number of entity instances.
  --enable_logging ENABLE_LOGGING
                        Enable logging, or not. Set environment variable LOGLEVEL=DEBUG␣
↪or INFO.
  --rand_seed RAND_SEED
                        Random number seed.
  --end_time_ticks END_TIME_TICKS
                        Simulation end time (ticks).
  --progress_interval_ticks PROGRESS_INTERVAL_TICKS
                        Interval between reporting sim progress (ticks).
  --PARAM_interarrival_time_distribution_rate PARAM_INTERARRIVAL_TIME_DISTRIBUTION_RATE
  --PARAM_service_time_distribution_rate PARAM_SERVICE_TIME_DISTRIBUTION_RATE
```

You will see that a couple of the arguments above are prefixed with `PARAM`. These arguments are used to configure parameters of the model that are defined *in the model definition*. (The rest of the arguments are for configuring other aspects of the simulation of the model.)

Below, we run the simulation again with different model parameters by providing values for the `PARAM_...` command line arguments. All the other arguments are specified the same as the previous example above.

The model parameters we're setting here configure the random distributions used to determine the times of queue arrival and service events.

```
$LOGLEVEL=INFO python generated/python_arrayified/mm1_queue.py --enable_logging=1 --end_
↪time_ticks=1000 --num_entities=2 --PARAM_interarrival_time_distribution_rate 5 --PARAM_
↪service_time_distribution_rate 10
```

The highlighted lines below indicate what has changed in the simulation when running with the new `PARAM_...` arguments. Some of the output has been truncated.

The first highlighted lines are reading back to us the parameter values that we've provided. Lower down, you'll see that the num_waiting values have changed relative to the previous example, which was using the default values for these parameters, which were different from the values that we've specified for this run.

If you check the M/M/1 Queue theory you'll see that the highlighted results match what we'd expect given the new PARAM_... values.

```
Sim Arguments:
{
  'PARAM_interarrival_time_distribution_rate': 5.0,
  'PARAM_service_time_distribution_rate': 10.0,
  'enable_logging': 1,
  'end_time_ticks': 100.0,
  'num_entities': 2,
  'progress_interval_ticks': 10.0,
  'rand_seed': 1,
}

INFO:__main__:t = 0.0
INFO:__main__:t = 10.01873127619964
INFO:__main__:t = 20.084890768112547
INFO:__main__:t = 30.094167821482863
INFO:__main__:t = 40.108556547423525
INFO:__main__:t = 50.11398855853502
INFO:__main__:t = 60.149593728731936
INFO:__main__:t = 70.17381362676747
INFO:__main__:t = 80.18713329960623
INFO:__main__:t = 90.21419988309019

Runtime: [1.4623036999983015 s]

-----------------------------------
State Tracking
-----------------------------------
instance(Entities_MM1_Queue_state_tracking):
  mm1_queue_arrivals__arriving: StateTracking(track_cumulative=None),
  mm1_queue_arrivals__idle: StateTracking(track_cumulative=None),
  mm1_queue_service__idle: StateTracking(track_cumulative=None),
  mm1_queue_service__servicing: StateTracking(track_cumulative=None)

Writing entity attribute history to [entity_attribute_history.nc]

-----------------------------------
Entity Attribute History Summary over ['time']
-----------------------------------

-------------------- mean --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
Data variables:
    num_waiting  (entity_id) float64 1.206 0.9456
------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
-------------------- std --------------------
<xarray.Dataset>
Dimensions:      (entity_id: 2)
Coordinates:
  * entity_id    (entity_id) int64 0 1
Data variables:
    num_waiting  (entity_id) float64 1.626 1.321
---------------------------------------------------------
```

1.8 Peruse Example Model Definition

The model below implements an M/M/1 Queue. This is the model fed to the diagram generator and the simulation code generator in the previous sections.

Note: This section is focused on how to use and iterate on a model, but we're not describing all the elements of the model here. The concepts and constructs used to build this model are described in the *Overview* and *Modeling Constructs* sections.

```
import numpy
from dymodetron import \
    EntityType, \
    EntityAttribute, \
    StateMachine, \
    ModelDescription, \
    Params, \
    State, \
    Event, \
    Transition, \
    Transitions, \
    dymaction, \
    initial_state, \
    initial_event, \
    types, \
    random as dyrandom, \
    action

# This is the model description. The name of the class is the name of the model.
# The docstring is a place to summarize the model.
class mm1_queue(ModelDescription):
    """An mm1 queue model. See https://en.wikipedia.org/wiki/M/M/1_queue"""

# This model has a single entity type. The name of this class is the name of the entity␣
↪type.
class MM1_Queue(EntityType):

    # We'll track how many items are waiting in the queue to be serviced.
```

(continues on next page)

(continued from previous page)

```
    num_waiting = EntityAttribute(attribute_type=types.scalar_int, initial_value=0.0, probe=True)

# Model parameters go here.
class ModelParameters(Params):

    # We'll sample from this distribution for interarrival times. Note that the distribution parameter
    # is in terms of rate (1/time), rather than time directly. The samples will be interarrival times, though.
    interarrival_time_distribution = dyrandom.ExponentialDistribution(rate=3.0)

    # We'll sample from this distribution for time spent waiting to be serviced by the queue. Same comment
    # as above about rate vs. time.
    service_time_distribution = dyrandom.ExponentialDistribution(rate=12.0)

# These events are used in the state machines below.
class arrival_event(Event):
    pass

class done_arriving_event(Event):
    pass

class service_event(Event):
    pass

class done_servicing_event(Event):
    pass

# This is one of two state machines in this model.
class mm1_queue_arrivals(StateMachine):
    """Handles arrivals for mm1 queue."""

    # The model has to specify what entity type is associated with this state machine.
    # This state machine models a portion of the behavior of an 'MM1_Queue'.
    entity_type = MM1_Queue()

    # This is how states are defined within a state machine.
    class idle(State):

        # These statements are executed when entities enter this state.
        @dymaction
        def entry_action(queue: MM1_Queue):
            """Kick off the next arrival."""

            # Declare an entity variable to store the next arrival time. There's one value in this
            # variable for each entity in the entity set 'queue'.
            t_next_arrival = action.declare(
                entity_set=queue,
                var_type=numpy.float64
```

(continues on next page)

(continued from previous page)

```
            )

            # Sample from the random distribution, and use those samples to populate the values of the
            # entity variable 't_next_arrival'. One value is sampled and stored for each entity in the
            # entity set 'queue'.
            action.assign(
                entity_set=queue,
                entity_var=t_next_arrival,
                dist=ModelParameters.interarrival_time_distribution
            )

            # Generate 'arrival_event' on each entity in the entity set 'queue'. The times for the events
            # are taken from the entity variable that we declared and assigned to above.
            action.generate_event_rel(
                entity_type=MM1_Queue(),
                entity_set=queue,
                event=arrival_event(),
                time_ticks_rel=t_next_arrival.value()
            )

    class arriving(State):

        @dymaction
        def entry_action(queue: MM1_Queue):
            """Count the arrival, and kick off transition back to mm1_queue_arrivals.idle state."""

            # This is how we track how many are waiting in the queue to be serviced.
            queue.num_waiting = queue.num_waiting + 1

            # This will kick the state machine back to state 'idle' (see transition table below),
            # for each entity in the entity set 'queue'.
            action.generate_event_rel(
                entity_type=MM1_Queue(),
                entity_set=queue,
                event=done_arriving_event(),
                time_ticks_rel=0
            )

    # The transition table defines how the state machine moves between states when it receives events.
    # The information in this table is also visualized in the generated state machine diagram,
    # which is arguably easier to interpret.
    transitions = Transitions([
        Transition(event_type=initial_event(),   source_state=initial_state(), target_state=idle()),
        Transition(           arrival_event(),               idle(),                arriving()),
```

(continues on next page)

(continued from previous page)

```
        Transition(           done_arriving_event(),          arriving(),                 idle()),
    ])

# This is the other state machine for this model. A model can have as many state machines
# as you want.
class mm1_queue_service(StateMachine):
    """Handles servicing mm1 queue."""

    entity_type = MM1_Queue()

    class idle(State):

        @dymaction
        def entry_action(queue: MM1_Queue):
            """Kick off the next service."""

            # Declare entity variable to store the next service time.
            t_next_service = action.declare(
                entity_set=queue,
                var_type=numpy.float64
            )

            # Sample from the random distribution, and use those samples to populate the values
            # of 't_next_service'. One value per entity in entity set 'queue'.
            action.assign(
                entity_set=queue,
                entity_var=t_next_service,
                dist=ModelParameters.service_time_distribution
            )

            # Generate 'service_event' on each entity in the entity set 'queue', using the times we
            # sample above.
            action.generate_event_rel(
                entity_type=MM1_Queue(),
                entity_set=queue,
                event=service_event(),
                time_ticks_rel=t_next_service.value()
            )

    class servicing(State):

        @dymaction
        def entry_action(queue: MM1_Queue):
            """Count the service, and kick off transition back to mm1_queue_service.idle state."""

            # If the queue has more than zero waiting, we'll decrement.
            with action.entity_match(
                entity_type=MM1_Queue(),
```

(continues on next page)

(continued from previous page)

```
167                 entity_set=queue,
168                 criteria=lambda q: q.num_waiting > 0
169             ) as non_empty_queue:
170
171                 # Update how many are waiting in the queue to be serviced.
172                 non_empty_queue.num_waiting = non_empty_queue.num_waiting - 1
173
174             # This will kick the state machine back to state 'idle' (see transition table␣
    ↪below),
175             # for each entity in the entity set 'queue'.
176             action.generate_event_rel(
177                 entity_type=MM1_Queue(),
178                 entity_set=queue,
179                 event=done_servicing_event(),
180                 time_ticks_rel=0
181             )
182
183     # The transition table defines how the state machine moves between states when it␣
    ↪receives events.
184     # The information in this table is also visualized in the generated state machine␣
    ↪diagram,
185     # which is arguably easier to interpret.
186     transitions = Transitions([
187         Transition(event_type=initial_event(),   source_state=initial_state(), target_
    ↪state=idle()),
188         Transition(          service_event(),               idle(),                ␣
    ↪   servicing()),
189         Transition(          done_servicing_event(),        servicing(),           ␣
    ↪   idle()),
190     ])
191
```

1.9 Iterate On Model Definition

Now we'll change the model to capture some more real-life effects. We'll leave the 'arrivals' part of the model alone, and modify the 'service' portion to model queue processors being on and off shift, i.e. services are being handled or not. Here's the new state machine diagram:

`shifty_mm1_queue_service.html`:

The changes in the model definition, relative to the previous example model, are highlighted below.

What we've changed:

- New model name and summary description (lines 21-22).
- New random distributions for modeling on/off shift times (lines 39-43).
- New events for transitioning on/off shift (lines 58-62).
- New top-level state machine `shifty_mm1_queue_service` to define how queues are serviced (starting on line 129).

- If you look closely, you'll see that the original `mm1_queue_service` state machine is now a sub-state-machine within the top-level `shifty_mm1_queue_service`. The top-level state machine toggles back and forth between the old behavior happening (on shift), vs. nothing happening (off shift).
- We've also modified the `mm1_queue_service` state machine (which is now a sub-state-machine) to have an entry action that schedules the next `shift_end_event` to take us back to state `off_shift`.
- The `off_shift` state entry action schedules the next time the processor is servicing requests again.

```
import numpy
from dymodetron import \
    EntityType, \
    EntityAttribute, \
    StateMachine, \
    SubStateMachine, \
    ModelDescription, \
    Params, \
    Param, \
    State, \
    Event, \
    Transition, \
    Transitions, \
    dymaction, \
    initial_state, \
    initial_event, \
    types, \
    random as dyrandom, \
    action

class mm1_queue_with_shifts(ModelDescription):
    """The example mm1 queue model, modified for queue processing happening only during␣
↪certain periods of time."""

class MM1_Queue(EntityType):

    # We'll track how many items are waiting in the queue to be serviced.
    num_waiting = EntityAttribute(attribute_type=types.scalar_int, initial_value=0.0,␣
↪probe=True)

class ModelParameters(Params):

    # We'll sample from this distribution for interarrival times. Note that the␣
↪distribution parameter
    # is in terms of rate (1/time), rather than time directly. The samples will be inter-
↪arrival times, though.
    interarrival_time_distribution = dyrandom.ExponentialDistribution(rate=3.0)

    # We'll sample from this distribution for time spent waiting to be serviced by the␣
↪queue. Same comment
    # as above about rate vs. time.
    service_time_distribution = dyrandom.ExponentialDistribution(rate=12.0)

    # How many hours a queue processor stays on shift.
    shift_on_distribution = dyrandom.NormalDistribution(mean=8.0, std=0.1)
```

(continues on next page)

(continued from previous page)

```
    # How many hours a queue processor stays off shift.
    shift_off_distribution = dyrandom.NormalDistribution(mean=16.0, std=0.5)

# These events are used in the state machines below.
class arrival_event(Event):
    pass

class done_arriving_event(Event):
    pass

class service_event(Event):
    pass

class done_servicing_event(Event):
    pass

class shift_start_event(Event):
    pass

class shift_end_event(Event):
    pass


class mm1_queue_arrivals(StateMachine):
    """Handles arrivals for mm1 queue."""

    entity_type = MM1_Queue()

    class idle(State):

        @dymaction
        def entry_action(queue: MM1_Queue):
            """Kick off the next arrival."""

            # Declare an entity variable to store the next arrival time. There's one value in this
            # variable for each entity in the entity set 'queue'.
            t_next_arrival = action.declare(
                entity_set=queue,
                var_type=numpy.float64
            )

            # Sample from the random distribution, and use those samples to populate the values of the
            # entity variable 't_next_arrival'. One value is sampled and stored for each entity in the
            # entity set 'queue'.
            action.assign(
                entity_set=queue,
                entity_var=t_next_arrival,
                dist=ModelParameters.interarrival_time_distribution
```

(continues on next page)

(continued from previous page)

```
            )

            # Generate 'arrival_event' on each entity in the entity set 'queue'. The times for the events
            # are taken from the entity variable that we declared and assigned to above.
            action.generate_event_rel(
                entity_type=MM1_Queue(),
                entity_set=queue,
                event=arrival_event(),
                time_ticks_rel=t_next_arrival.value()
            )

    class arriving(State):

        @dymaction
        def entry_action(queue: MM1_Queue):
            """Count the arrival, and kick off transition back to mm1_queue_arrivals. idle state."""

            # This is how we track how many are waiting in the queue to be serviced.
            queue.num_waiting = queue.num_waiting + 1

            # This will kick the state machine back to state 'idle' (see transition table below),
            # for each entity in the entity set 'queue'.
            action.generate_event_rel(
                entity_type=MM1_Queue(),
                entity_set=queue,
                event=done_arriving_event(),
                time_ticks_rel=0
            )

    # The transition table defines how the state machine moves between states when it receives events.
    # The information in this table is also visualized in the generated state machine diagram,
    # which is arguably easier to interpret.
    transitions = Transitions([
        Transition(event_type=initial_event(),   source_state=initial_state(), target_state=idle()),
        Transition(          arrival_event(),               idle(),                arriving()),
        Transition(          done_arriving_event(),         arriving(),            idle()),
    ])


class shifty_mm1_queue_service(StateMachine):
    """Handles servicing mm1 queue when on shift, otherwise idle."""

    entity_type = MM1_Queue()

```

(continues on next page)

(continued from previous page)

```
    class off_shift(State):
        @dymaction
        def entry_action(queue: MM1_Queue):
            """Schedule the next 'on' shift."""

            t_next_shift = action.declare(
                entity_set=queue,
                var_type=numpy.float64
            )

            action.assign(
                entity_set=queue,
                entity_var=t_next_shift,
                dist=ModelParameters.shift_off_distribution
            )

            action.generate_event_rel(
                entity_type=MM1_Queue(),
                entity_set=queue,
                event=shift_start_event(),
                time_ticks_rel=t_next_shift.value()
            )

    class mm1_queue_service(SubStateMachine):
        """Handles servicing mm1 queue."""

        @dymaction
        def entry_action(queue: MM1_Queue):
            """Schedule the next 'off' shift."""

            t_shift_ends = action.declare(
                entity_set=queue,
                var_type=numpy.float64
            )

            action.assign(
                entity_set=queue,
                entity_var=t_shift_ends,
                dist=ModelParameters.shift_on_distribution
            )

            action.generate_event_rel(
                entity_type=MM1_Queue(),
                entity_set=queue,
                event=shift_end_event(),
                time_ticks_rel=t_shift_ends.value()
            )


        class idle(State):

            @dymaction
```

(continues on next page)

(continued from previous page)

```
            def entry_action(queue: MM1_Queue):
                """Kick off the next service."""

                # Declare entity variable to store the next service time.
                t_next_service = action.declare(
                    entity_set=queue,
                    var_type=numpy.float64
                )

                # Sample from the random distribution, and use those samples to populate the values
                # of 't_next_service'. One value per entity in entity set 'queue'.
                action.assign(
                    entity_set=queue,
                    entity_var=t_next_service,
                    dist=ModelParameters.service_time_distribution
                )

                # Generate 'service_event' on each entity in the entity set 'queue', using the times we
                # sample above.
                action.generate_event_rel(
                    entity_type=MM1_Queue(),
                    entity_set=queue,
                    event=service_event(),
                    time_ticks_rel=t_next_service.value()
                )

        class servicing(State):

            @dymaction
            def entry_action(queue: MM1_Queue):
                """Count the service, and kick off transition back to mm1_queue_service.idle state."""

                # If the queue has more than zero waiting, we'll decrement.
                with action.entity_match(
                    entity_type=MM1_Queue(),
                    entity_set=queue,
                    criteria=lambda q: q.num_waiting > 0
                ) as non_empty_queue:

                    # Update how many are waiting in the queue to be serviced.
                    non_empty_queue.num_waiting = non_empty_queue.num_waiting - 1

                # This will kick the state machine back to state 'idle' (see transition table below),
                # for each entity in the entity set 'queue'.
                action.generate_event_rel(
                    entity_type=MM1_Queue(),
                    entity_set=queue,
                    event=done_servicing_event(),
```

(continues on next page)

(continued from previous page)

```
                    time_ticks_rel=0
                )

        # The transition table defines how the state machine moves between states when it receives events.
        # The information in this table is also visualized in the generated state machine diagram,
        # which is arguably easier to interpret.
        transitions = Transitions([
            Transition(event_type=initial_event(),   source_state=initial_state(), target_state=idle()),
            Transition(          service_event(),               idle(),              servicing()),
            Transition(          done_servicing_event(),        servicing(),         idle()),
        ])

    # This is the transition table for the top-level state machine 'shifty_mm1_queue_service'.
    transitions = Transitions([
        Transition(event_type=initial_event(),   source_state=initial_state(), target_state=off_shift()),
        Transition(          shift_start_event(),           off_shift(),             mm1_queue_service()),
        Transition(          shift_end_event(),             mm1_queue_service(),     off_shift()),
    ])

```

We can generate the code for this as follows:

```
$ python -m dymodetron.generators.python_arrayified --model_definition_file=examples/mm1_queue_with_shifts.py --overwrite_existing=1
python_arrayified
Generating code for entity type [MM1_Queue]
Outputting to [generated/python_arrayified/mm1_queue_with_shifts.py]
```

If we ask the generated code for `--help`, it will tell us about the command line options to configure the new parameters:

```
$ python generated/python_arrayified/mm1_queue_with_shifts.py --help
usage: mm1_queue_with_shifts.py [-h] [--num_entities NUM_ENTITIES] [--enable_logging ENABLE_LOGGING] [--rand_seed RAND_SEED] [--end_time_ticks END_TIME_TICKS] [--progress_interval_ticks PROGRESS_INTERVAL_TICKS]
                                [--PARAM_interarrival_time_distribution_rate PARAM_INTERARRIVAL_TIME_DISTRIBUTION_RATE] [--PARAM_service_time_distribution_rate PARAM_SERVICE_TIME_DISTRIBUTION_RATE]
                                [--PARAM_shift_on_distribution_mean PARAM_SHIFT_ON_DISTRIBUTION_MEAN] [--PARAM_shift_on_distribution_std PARAM_SHIFT_ON_DISTRIBUTION_STD]
                                [--PARAM_shift_off_distribution_mean PARAM_SHIFT_OFF_DISTRIBUTION_MEAN] [--PARAM_shift_off_distribution_std PARAM_SHIFT_OFF_DISTRIBUTION_STD]
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  --num_entities NUM_ENTITIES
                        Number of entity instances.
  --enable_logging ENABLE_LOGGING
                        Enable logging, or not. Set environment variable LOGLEVEL=DEBUG␣
↪or INFO.
  --rand_seed RAND_SEED
                        Random number seed.
  --end_time_ticks END_TIME_TICKS
                        Simulation end time (ticks).
  --progress_interval_ticks PROGRESS_INTERVAL_TICKS
                        Interval between reporting sim progress (ticks).
  --PARAM_interarrival_time_distribution_rate PARAM_INTERARRIVAL_TIME_DISTRIBUTION_RATE
  --PARAM_service_time_distribution_rate PARAM_SERVICE_TIME_DISTRIBUTION_RATE
  --PARAM_shift_on_distribution_mean PARAM_SHIFT_ON_DISTRIBUTION_MEAN
  --PARAM_shift_on_distribution_std PARAM_SHIFT_ON_DISTRIBUTION_STD
  --PARAM_shift_off_distribution_mean PARAM_SHIFT_OFF_DISTRIBUTION_MEAN
  --PARAM_shift_off_distribution_std PARAM_SHIFT_OFF_DISTRIBUTION_STD
```

Below we run with the arrival and service rates such that the processor is just keeping up with demand while on shift. Note that we're running with 5 queues now. We've configured so that queues processors on average run for 8 hours with 16 hour breaks.

```
$ LOGLEVEL=DEBUG python generated/python_arrayified/mm1_queue_with_shifts.py --enable_
↪logging=1 --end_time_ticks=100 --num_entities=5 --PARAM_interarrival_time_distribution_
↪rate=10 --PARAM_service_time_distribution_rate=20 --PARAM_shift_on_distribution_mean=8␣
↪--PARAM_shift_off_distribution_mean=16 | grep -E "shift_(start|end)_event" | grep␣
↪Processing
```

We've enabled `LOGLEVEL=DEBUG` above, and are applying some `bash` and `grep` magic to filter the output in order to look at shift start and end events:

```
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=15.839870402347834, event_
↪id=7, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7a30>, entity_ids=array([2]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=16.411418394602425, event_
↪id=8, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7a30>, entity_ids=array([0]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=16.48581232444172, event_
↪id=9, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7a30>, entity_ids=array([4]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=16.516611992313333, event_
↪id=10, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7a30>, entity_ids=array([3]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=16.587737090994786, event_
↪id=11, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7a30>, entity_ids=array([1]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=24.00950271981257, event_
↪id=1642, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f77f0>, entity_ids=array([2]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=24.34930968829747, event_
↪id=1719, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7b50>, entity_ids=array([0]))]
```

(continues on next page)

(continued from previous page)

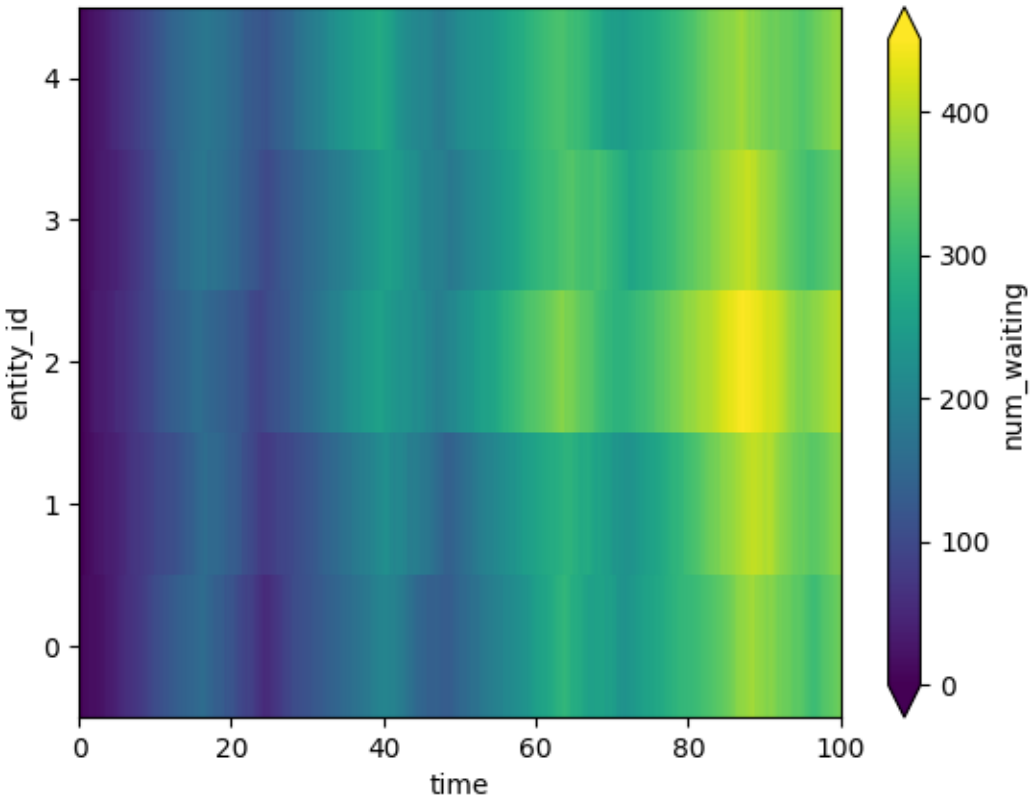
```
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=24.389206084520275, event_
↪id=1751, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7af0>, entity_ids=array([3]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=24.506864686216 39, event_
↪id=1738, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7ca0>, entity_ids=array([4]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=24.71094645767817, event_
↪id=1784, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7940>, entity_ids=array([1]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=39.39334645813642, event_
↪id=4160, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7a60>, entity_ids=array([4]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=39.58149606852187, event_
↪id=4051, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7b80>, entity_ids=array([2]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=40.290511055040334, event_
↪id=4130, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f79a0>, entity_ids=array([0]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=40.43694464902083, event_
↪id=4195, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7fa0>, entity_ids=array([1]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=40.86773007523732, event_
↪id=4143, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7b20>, entity_ids=array([3]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=47.32726868032039, event_
↪id=5670, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7a00>, entity_ids=array([4]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=47.58026972021337, event_
↪id=5691, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7f10>, entity_ids=array([2]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=48.14047961449103, event_
↪id=5830, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f77f0>, entity_ids=array([0]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=48.36089674700067, event_
↪id=5859, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7d00>, entity_ids=array([1]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=48.86332882350327, event_
↪id=5970, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7be0>, entity_ids=array([3]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=63.355878950906636, event_
↪id=8018, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7ee0>, entity_ids=array([2]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=63.54038311499454, event_
↪id=7961, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f78e0>, entity_ids=array([4]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=63.636522730745384, event_
↪id=8129, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7f70>, entity_ids=array([0]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=64.25318121513186, event_
↪id=8170, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f79a0>, entity_ids=array([1]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=64.8634887873948, event_
↪id=8253, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7bb0>, entity_ids=array([3]))]
```

(continues on next page)

(continued from previous page)

```
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=71.54804911256912, event_
↪id=9702, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7e20>, entity_ids=array([2]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=71.60075531927286, event_
↪id=9745, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7ee0>, entity_ids=array([4]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=71.62732639263827, event_
↪id=9762, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7b80>, entity_ids=array([0]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=72.14682560697014, event_
↪id=9901, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7f70>, entity_ids=array([1]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=72.83586559363486, event_
↪id=10052, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7f40>, entity_ids=array([3]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=87.01674377115683, event_
↪id=12080, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7850>, entity_ids=array([4]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=87.22284147193372, event_
↪id=12067, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7fd0>, entity_ids=array([2]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=88.04646050718927, event_
↪id=12275, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7e20>, entity_ids=array([3]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=88.41109984957293, event_
↪id=12085, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7bb0>, entity_ids=array([0]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=88.53995986997506, event_
↪id=12176, event_obj=<examples.mm1_queue_with_shifts.shift_start_event object at␣
↪0x7fb0a56f7ca0>, entity_ids=array([1]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=94.99741638388537, event_
↪id=13696, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7f70>, entity_ids=array([4]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=95.21440310860663, event_
↪id=13721, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f78e0>, entity_ids=array([2]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=95.9447754901954, event_
↪id=13876, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7a90>, entity_ids=array([3]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=96.44793863168 62, event_
↪id=13945, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7f40>, entity_ids=array([0]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=96.56639366711914, event_
↪id=13978, event_obj=<examples.mm1_queue_with_shifts.shift_end_event object at␣
↪0x7fb0a56f7850>, entity_ids=array([1]))]
```

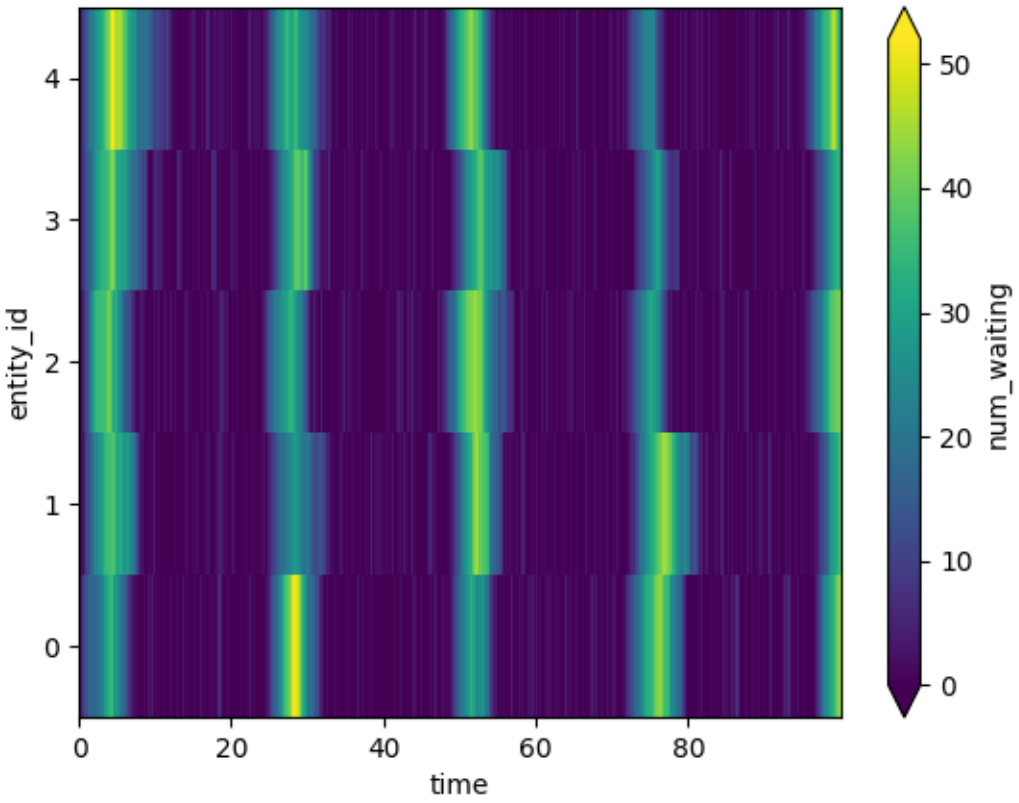
The summary plot shows us that the queues are having trouble keeping up with demand:



Now we'll modify the shift times, so that queue processors run for 20 hours alternating with 4 hour breaks.

```
$ LOGLEVEL=DEBUG python generated/python_arrayified/mm1_queue_with_shifts.py --enable_
↪logging=1 --end_time_ticks=100 --num_entities=5 --PARAM_interarrival_time_distribution_
↪rate=10 --PARAM_service_time_distribution_rate=20 --PARAM_shift_on_distribution_
↪mean=20 --PARAM_shift_off_distribution_mean=4 | grep -E "shift_(start|end)_event" |␣
↪grep Processing
```

The summary plot shows that the queue processors are now able to manage a little better. Note that the color scale is not the same in this plot as the last one.


entity_id
4
3
2
1
0
0
20
40
60
80
time
num_waiting
50
40
30
20
10
0

CHAPTER

TWO

OVERVIEW

DY NAMIC **MODE** L **TRON**

Dymodetron is for building and analyzing dynamic models efficiently.

2.1 What do you mean by "dynamic models"?

- Models involving time-varying states/attributes and rules/logic/equations for changing state and attributes.
- Models having a set of identifiable entities, usually interacting with one another.
- Models where you want to quantify how those entities interact with one another and how their states and attributes progress and change over time.
- Often with some random processes involved in the model.

2.2 How does one use Dymodetron?

The figure below illustrates the Dymodetron user workflow. There are two loops oriented around the `Build Model` activity. Each piece is described in the subsequent sections.

2.2.1 1. Build Model

- A Dymodetron user builds or modifies a model using the Dymodetron modeling constructs.
- The user does this in a python source code file, using a number of modeling constructs provided by Dymodetron.
- This is done in order to ask the model some questions.

Note: If you are using a python integrated development environment (IDE), all the features of auto-complete, code navigation, identifier renaming, code refactoring, etc., are there to help while building your model. This is why we've 'borrowed' python as the Dymodetron model definition language.

2.2.2 2. Generate Model Simulation Code

- The user invokes the Dymodetron code generator to create a simulation of the model.

2.2.3 3. Generate Model Visualizations

- Optional, recommended: the user invokes the Dymodetron diagram generator(s) to create visualizations of the model, and uses these visualizations as a central element of communication/collaboration with others around the questions and answers associated with #1 and #5. These are not visualizations of the results of simulations of the model. Instead, these are visualizations of the model structure and behavior.

2.2.4 4. Run Simulations

- The user runs the simulation from the command line or IDE.
- The simulation creates result data files.

2.2.5 5. Analyze Results

- The user views the result data or writes additional scripts to process/analyze the results.
- This is where the user gets some answers to those questions in #1.

2.2.6 6. Iterate

- GOTO #1.

Note: When the model involves one or more random processes, then steps #2,4,5 are complicated by the need to run many copies of the simulation each having different random number seeds, and then collect, aggregate, analyze the results across the many simulation runs. For now, the user has to do all that work by themselves, but Dymodetron has *aspirations to make this easier for the user*.

2.3 What does a dymodetron model look like?

Dymodetron enables modeling of behavior in the form of state machines. Entities being modeled are described as existing in one or more states at any given time. The model generates events based on certain criteria being satisfied; entities then react to the arrival of those events by transitioning between states. Model logic is executed upon entry to or exit from a state, often resulting in generation of additional events, which continues the propagation of the model. The user builds a model like this because they want to ask questions about e.g. how many entities enter a particular state over a certain window of time, or how long entities stay in certain states, and other questions of this nature.

We discuss the concepts underpinning Dymodetron state machines models in *another section*.

2.3.1 Example Model Diagram (generated)

Here's a visualization of an example Dymodetron state machine model. The model captures a crude description of the behavior of cats. The boxes are states. The arrows are transitions. The labels on the transition arrows indicate the event that triggers the transition.

This diagram is generated by the Dymodetron state machine diagram generator, which takes the code for a model as its input. The code for this particular model is *below*. The diagram generator *is described here*. Generally speaking, whenever you think you're done with some model definition changes, you'll want to re-run the diagram generator to review the latest diagram(s). This increases the likelihood, but doesn't guarantee, that Dymodetron understands what you had in your head. It's a good idea to check the diagrams in along with your code, so that other people can more rapidly come up to speed on your model.

The logic for the state machine model shown above is as follows:

1. The state machine starts at the black dot on the left-hand side of the diagram.
    - The black dot is the 'initial state' for the state machine.
2. Immediately, the 'initial_event' transition is taken, putting all the cats in the simulation of the model into state 'sleeping'.
3. When a 'not_tired_event' happens to a cat, it will transition to 'awake'.
4. The 'awake' state has a sub-state-machine with additional logic.
5. An 'awake' cat immediately transitions to 'staring_out_window'.
6. Additional events take the cat in and out of 'eating' and 'drinking' states while the cat is 'awake'.
7. At some point, a 'tired_event' may occur, taking the cat back into state 'sleeping'.
    - In this model, a cat may transition to 'sleeping' while 'staring_out_window', or while 'eating', or while 'drinking'.
    - If a cat subsequently transitions again back to 'awake', it will again immediately transition to 'staring_out_window'.

2.3.2 Multiple interacting state machines

A typical model will have multiple state machines, describing different aspects of the behavior of the entities that you care about. Our 'cat' model above might be augmented with another, separate, state machine that describes e.g. the internal biological processes of our cats as they go about their daily activities. This separate state machine will have a completely separate set of states. It can use none, some, or all of the same events used in the state machine above. The 2nd state machine can generate events that cause transitions in 1st state machine, and vice versa. For example, the 2nd state machine might generate 'hungry_event' and 'thirsty_event', triggering the associated transitions in the 1st state machine shown above. Or, the 2nd state machine might have transitions also triggered by the 'tired_event' and 'not_tired_event', which means that the same events can cause state transitions to occur simultaneously in both state machines.

In this way, the behaviors of the modeled entities can be decomposed into separate pieces but also linked together where necessary.

2.3.3 Multiple interacting entities

A typical model will have a number of entities. Each entity has its own 'copy' of the state machine(s) in the model. For example, our 'cat' model may have 100,000 cats, each of which is 'running' its own 'copy' of the state machine shown above. At any point in simulated time, each individual cat will have its own individual state. Individual entities are called entity instances.

Individual entities can generate events targeted at themselves or at other entities. In this way, the modeled entities can interact with one another.

2.3.4 Where do events come from?

Not shown in the diagram above are the mechanisms by which the events are generated. Those happen within state entry actions, which can be seen in the *model code*, where you will see action statements similar to the following:

```
# This action statement generates the named event on all the cats in the named entity␣
↪set.
# The time of the event will be 7 time ticks in in the future, relative to the simulated␣
↪time
# that the action statement is executed.
#
# Not shown in this example: how to construct the entity set 'cat_with_odd_times_staring_
↪out_window'.
# See the section on entity sets for more information on that.
#
action.generate_event_rel(
    entity_type=Cat(),
    entity_set=cat_with_odd_times_staring_out_window,
    event=thirsty_event(),
    time_ticks_rel=7
)
```

2.3.5 Targeting events at entities

When you run a model simulation, it will typically have a large number of entity instances 'running' in the simulation. In the example above, those will all represent cats. Each individual cat has its own "copy" of the state machine running, and so all the cats may be in different states at any given simulated time. Individual events are typically targeted to a subset of the cat population: anything from one cat, to a handful of cats, to all the cats.

Subsets of the entities in the model are gathered into *entity sets*, by specifying filter criteria to pick out the entities that should belong to the set. The targeting of events in your model is designed around entity filter criteria that you provide.

In our cat example, the 'cat criteria' can be based on what *state* a cat is in. For example we might have some logic that does something special just for cats that are currently in the 'eating' state.

In the example below illustrates two different mechanisms for selecting subsets of entities to be targeted for events. First, we select all the cats currently in the 'eating' state. This is done using *a construct for filtering entity sets based on criteria you provide*.

Then we model that cats currently in state 'eating' will have a 30% chance of getting tired in 5 seconds. The example below does this using *a construct for generating events on a randomly chosen subset of an entity set*.

```
# This 'entity_match' statement filters down from the set of all cats, to the set of all␣
↪cats
# currently in state 'eating'.
with action.entity_match(
    # Type of entity to match.
    entity_type=Cat(),

    # The entity instance(s) to look through. This comes from the context in which
    # this 'entity_match' statement is being executed.
    entity_set=cat,

    # The criteria to filter on: cats that are currently in the 'eating' sub-state
    # of the 'awake' state within the 'cat_sleeping_eating_drinking' state machine.
    # The 'lambda' syntax is how we build an expression describing the criteria we are␣
↪looking for.
    # Each entity in the entity_set 'cat' will be evaluated against this criteria, to␣
↪determine if it
    # should be included in the entity_set 'eating_cats' used below.
    criteria= lambda c: action.in_state(c, cat_sleeping_eating_drinking.awake.eating())
) as eating_cats:

    # In this section, we can now reference 'eating_cats', which is the set of all␣
↪entities
    # in the entity_set 'cat' that are also in state 'eating'.
    #
    # The action statement below randomly selects entities from entity set 'eating_cats'.
    #
    # Each entity instance has a 30% chance of being selected.
    #
    # The event 'tired_event' will be generated on the selected entities, at simulation␣
↪time 5 ticks
    # in the future relative to the simulated time at which the action statement is␣
↪executed.
    action.generate_event_binomial(
        entity_type=Cat(),
        entity_set=eating_cats,
        event_a=tired_event(),
        probability_event_a=0.3,
        time_ticks_rel=5
    )
```

The 'cat criteria' can also be based on cat *attributes*. Attributes are ways of quantifying the measurable aspects of entities, separate from the state(s) that an entity is in. (Attributes are not shown in the diagram above). For example, a cat might have attributes such as weight, height, eye color.

The 'cat criteria' can also be based on a random selection process. You can use *random distributions* to model random processes that result in some entities becoming part of an entity set, and others not. Sometimes the 'cat criteria' is a combination of states, attributes, and random processes.

Entity criteria are described further *in the section on entity sets*.

2.3.6 Example Model Source

This model associated with the diagram above is listed below. There are a number of things about the model that are not represented in the state chart diagram. The various elements of a Dymodetron model are described in *the section on modeling constructs*.

```
from dymodetron import \
    EntityType, \
    EntityAttribute, \
    StateMachine, \
    SubStateMachine, \
    ModelDescription, \
    Params, \
    State, \
    Event, \
    Transition, \
    Transitions, \
    types, \
    action, \
    dymaction, \
    dymodetrack, \
    initial_state, \
    initial_event

import numpy

#############################################################################################
#
# Model overview.
#
# This model is a simple example of a Dymodetron state machine. We describe
# the behaviors of a cat. The main point of the model is to illustrate how some of
# the Dymodetron model definition elements work.
#
# A Dymodetron model consists of entity types, entity attributes, event types,
# state machines, states, and transition tables consisting of individual transitions.
#
# We use python classes as the atomic unit for many of the elements of the model definition.
# These should not be interpreted as python classes in the usual sense. Instead, we are
# co-opting the python language as a model definition language. (Instead of creating a new
# model definition language). We use python classes in many places as an atomic unit for the
# construction of our model definition.
#
# You will notice that we use a mixture of capitalization conventions for classes, some of
# which are not conventional. Specifically, we use CamelCase for the names of the Dymodetron
# model constructs such as StateMachine, Event, etc. We also use CamelCase for the entity
# type 'Cat'. We use a lower_case_naming_convention for names of events, state machines,
# states, entity attributes.
```

(continues on next page)

(continued from previous page)

```
#
# The unconventional capitalizations arguably make certain parts of the model definition
# easier on the eyes when the class names are used. You can use whatever capitalization
# conventions you want.
#############################################################################################
#

#############################################################################################
#
# Provide the model description.
#############################################################################################
#


# The model description is defined by creating a class that sub-classes 'ModelDescription'.
class cat_model(ModelDescription):
    # A model description just has a docstring explaining what you are up to with this model.
    """We model the behaviors of a certain kind of cat."""


#############################################################################################
#
# Define entity types.
#############################################################################################
#

# An entity is defined by creating a class that sub-classes 'EntityType'.
class Cat(EntityType):
    # You can have docstrings on any of the elements of your model.
    """Cats are a type of animal that is neither dog, duck, nor lizard."""

    # Here we declare the entity attributes of Cat.
    # The entity name is on the left-hand side.
    # On the right-hand side, we specify the type of the attribute, and the default
    # initial value.
    height_inches = EntityAttribute(attribute_type=types.scalar_float, initial_value=9.0)
    weight_pounds = EntityAttribute(attribute_type=types.scalar_float, initial_value=10.0)


#############################################################################################
#
# Define model parameters.
#############################################################################################
#

# Your model parameters live inside a class that sub-classes 'Params'.
class ModelParameters(Params):
    # No parameters for this model.
    pass
```

(continues on next page)

(continued from previous page)

```

#############################################################################################
# Define event types.  It is recommended, but not required, to end event names
# with '_event'.
#
# The meaning of each event type is defined by its usage in the state machines
# further below in the model. The event name is a useful label, but it's just a label.
# If you really want to know what an event "means", you have to look at where it is used
# to cause transitions in the model.
#############################################################################################

# An event type is defined by creating a class that sub-classes Event.
class tired_event(Event):
    pass

class not_tired_event(Event):
    pass

class hungry_event(Event):
    pass

class not_hungry_event(Event):
    pass

class thirsty_event(Event):
    pass

class not_thirsty_event(Event):
    pass


#############################################################################################
# Define state machines.
#############################################################################################
#
# The Cat model has one state machine.  (You can have more than one if you want).
#
# State machines can be hard to visualize solely by looking at this model code. What you want
# to do is use the dymodetron state machine diagram generator to create the diagram for this
# state machine.
#
# Do this by running the following command, from the dymodetron root folder:
#
#     python -m dymodetron.generators.state_machine_diagrams --model_definition_file=examples/cat.py --overwrite_existing=1
#
```

(continues on next page)

(continued from previous page)

```
# That will create a file:
#
#     generated/state_machine_diagrams/cat_sleeping_eating_drinking.html
#
# Open that file in a browser, and you will see the diagram for this state machine.
#
# A state machine is defined by creating a class that sub-classes StateMachine.
#
class cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping, eating, and drinking patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
    # In this model, there is only one entity type, 'Cat'.
    entity_type = Cat()

    # A state is defined by creating a nested class that sub-classes State.
    class sleeping(State):
        # We're going to define an entry action for this state. All entry action functions
        # need to be annotated with '@dymaction'.
        @dymaction
        def entry_action(cat: Cat):
            # The entry action takes an argument 'cat' of type 'Cat'. 'Cat' is the entity type we
            # defined above.

            # We want to model all cats as remaining in state 'sleeping' for 10 time ticks.
            # So we schedule the 'not_tired_event' to fire in 10 time ticks.
            # Time 'ticks' are whatever unit you want them to be. They could be seconds, minutes,
            # hours, days. You just have to make sure that you are consistently treating them as
            # whatever unit you have chosen.

            # This action statement generates an event at a time relative to the 'current' time
            # when the action is executed. For an entry action, the 'current' time is the time that
            # this state is entered.
            action.generate_event_rel(
                entity_type=Cat(),          # The type of entity to schedule the event on.
                entity_set=cat,                 # The entity instance(s) to schedule the event on.
                event=not_tired_event(),    # The event to schedule.
                time_ticks_rel=10           # The time in the future, relative to 'now', that the event should occur.
            )

    # A state can contain its own sub-state-machine. Whenever the state is entered,
    # the sub-state-machine will start executing. If you want a state to contain a state machine,
```

(continues on next page)

(continued from previous page)

```
    # you subclass from SubStateMachine instead of from State.
    class awake(SubStateMachine):

        # States within the sub-state-machine are defined in the same way all states are defined.
        class staring_out_window(State):
            @dymaction
            def entry_action(cat: Cat):

                # The logic we want to implement is:
                #
                #   Cats who have been staring out the window an even number of times get hungry in 5 ticks.
                #   Cats who have been staring out the window an odd number of times get thirsty in 7 ticks.

                # This action statement retrieves the number of times that this state has been entered by
                # the entity instance(s) represented by 'cat'.
                staring_out_window_count = action.get_state_entry_counter(cat, cat_sleeping_eating_drinking.awake.staring_out_window())

                # Generally speaking, you use the library of 'action.xyz()' statements to implement a lot
                # of the logic of your state machine model. The states, events, and transition tables provide
                # the skeleton of the model. The action statements are the meat on the bones, and fill in the
                # detail of how the state machine should propagate.

                # This 'entity_match' statement picks out the cats with an even 'staring_out_window_count'.
                with action.entity_match(
                    entity_type=Cat(),  # Type of entity to match.
                    entity_set=cat,         # The entity instance(s) to look through.
                    criteria=           # The criteria to match on.  It will be passed entity instance(s) to look at,
                                        # i.e. the cat(s).  Note that for this criteria, we don't actually look at the
                                        # cat(s), we are just looking at the 'staring_out_window_count'.
                        lambda c: numpy.mod(staring_out_window_count, 2) == 0
                ) as cat_with_even_times_staring_out_window:

                    # The 'with' statement above picked out all the cats such that their 'staring_out_window_count'
                    # is an even number. The 'as' clause labeled all those as 'cat_with_even_times_staring_out_window'.
                    # We use that label next in the 'entity' parameter, to target just those specific cats with the
                    # event we are generating.

```

(continues on next page)

(continued from previous page)

```
                    action.generate_event_rel(
                        entity_type=Cat(),
                        entity_set=cat_with_even_times_staring_out_window,
                        event=hungry_event(),
                        time_ticks_rel=5
                    )

                # This 'entity_match' statement picks out the cats with an odd 'staring_
↪out_window_count'.
                with action.entity_match(
                    entity_type=Cat(),
                    entity_set=cat,
                    criteria=lambda c: numpy.mod(staring_out_window_count, 2) == 1
                ) as cat_with_odd_times_staring_out_window:

                    action.generate_event_rel(
                        entity_type=Cat(),
                        entity_set=cat_with_odd_times_staring_out_window,
                        event=thirsty_event(),
                        time_ticks_rel=7
                    )

        @dymodetrack(track_cumulative=True)
        class eating(State):

            @dymaction
            def entry_action(cat: Cat):
                # In 10 ticks the cat isn't hungry anymore.
                action.generate_event_rel(
                    entity_type=Cat(),
                    entity_set=cat,
                    event=not_hungry_event(),
                    time_ticks_rel=10
                )

            @dymaction
            def exit_action(cat: Cat):
                # Cats gain a little weight every time they are done eating.
                cat.weight_pounds += 0.5


        class drinking(State):

            @dymaction
            def entry_action(cat: Cat):
                # We are modeling a world of giant cats: cats get a little taller every
                # time they take a drink of water.
                cat.height_inches += 0.1

                # In 3 ticks the cat isn't thirsty anymore.
                action.generate_event_rel(
                    entity_type=Cat(),
```

(continues on next page)

(continued from previous page)

```
                    entity_set=cat,
                    event=not_thirsty_event(),
                    time_ticks_rel=3
                )

        # Here, we define the transition table for the sub-state-machine within state 'awake'.
        # Transition tables describe how events cause transitions between states.
        #
        # You define a transition table with a class variable named 'transitions', of type 'Transitions'.
        # As you can see below, 'Transitions' takes an array of 'Transition' objects.
        #
        # Each row in the transition table is a 'Transition' object defining the event type that triggers
        # the transition, and the source/target states for the transition.
        #
        # This 'awake' sub-state-machine initializes into the 'staring_out_window' state. Then, events toggle it
        # back and forth to/from the 'eating' and 'drinking' states.
        transitions = Transitions([
            # Every state machine has a magic 'initial_event' and 'initial_state' automatically defined for you.
            # You don't have to write them into the model definition above, but you do have to include
            # them in the transition table here, so that the simulation knows what state to initialize the state
            # machine into.
            #
            # The 'initial_event' fires when the state machine starts executing. At that time, the state machine
            # transitions from the magic 'initial_state' to whichever state you tell it to transition to here
            # in the transition table.
            Transition(event_type=initial_event(),   source_state=initial_state(), target_state=staring_out_window()),
            Transition(          hungry_event(),                staring_out_window(),  eating()),
            Transition(          not_hungry_event(),            eating(),              staring_out_window()),
            Transition(          thirsty_event(),               staring_out_window(),  drinking()),
            Transition(          not_thirsty_event(),           drinking(),            staring_out_window())
        ])

    # Here, we define the transition table for the top-level state-machine 'cat_sleeping_eating_drinking'.
    #
    # The state machine initializes into the 'sleeping' state. Then, events toggle it back and forth
    # between 'sleeping' and 'awake'.
```

(continues on next page)

(continued from previous page)

```
    transitions = Transitions([
        Transition(event_type=initial_event(),   source_state=initial_state(), target_
↪state=sleeping()),
        Transition(           not_tired_event(),              sleeping(),              ␣
↪   awake()),
        Transition(           tired_event(),                  awake(),                 ␣
↪   sleeping())
    ])
```

2.3.7 Model Simulation Code (generated)

If you want to use Dymodetron to build and simulate models, you may not ever need to look at the generated simulation code. The exception might be for debugging the simulation of your model. In any case, this section illustrates what the generated simulation code for a model looks like.

The Dymodetron *code generator* consumes a model definition, and generates code to simulate the model. The generated code for the model above is listed below. The model entry actions embedded within the model definition above are also invoked during the simulation execution. You will see that the generated code also relies upon some Dymodetron package pieces.

```

import numpy
import numpy_indexed
from scipy.interpolate import interp1d
from dataclasses import dataclass, field
from typing import Any, Type, Union, List, Dict
from heapq import heappop, heappush
from functools import partial
import logging
import beeprint
import sys
import os
import argparse

from dymodetron import \
    Event, \
    State, \
    StateMachine, \
    action, \
    ActionStateHooks, \
    ActionEntityTypeHooks, \
    ActionLookupTableHooks, \
    ActionDistributionHooks, \
    TimedValue, \
    StateTracking


sys.path.append(os.path.join('..','..'))
import examples.cat as model


```

(continues on next page)

(continued from previous page)

```
log = logging.getLogger(__name__)


@dataclass(order=True)
class ScheduledEvent:
    # The heapq will sort on time_ticks and event_id.
    # Uniqueness and monotonicity (handled elsewhere) of scheduled_event.event_id
    # gives us the tie-breaker we want, so we get time order and then
    # schedule order.
    time_ticks: numpy.float64 = numpy.float64(0.0)
    event_id: numpy.uint64 = 0
    event_obj: Any = field(compare=False, default=None)
    # entity_ids = None means all
    entity_ids: 'numpy.array[numpy.uint64]' = field(compare=False, default=None)


@dataclass
class SourceTargetPair:
    source: 'numpy.array[numpy.int64]'
    target: 'numpy.array[numpy.int64]'
    target_state_last_entry_time_ticks: 'numpy.array[numpy.float64]' =␣
↪field(compare=False, default=None)
    source_state_definition: State = field(default=None) #TODO: default None is temporary
    target_state_definition: State = field(default=None) #TODO: default None is temporary


@dataclass
class StateData:
    state: Type
    state_entry_counters: 'numpy.array[numpy.uint64]' = field(compare=False,␣
↪default=None)
    last_state_entry_time_ticks: 'numpy.array[numpy.float64]' = field(compare=False,␣
↪default=None)
    state_tracking: StateTracking = field(compare=False, default=None)
    is_state_machine: bool = field(default=None)
    substate_initialization_event_type: Type = field(default=None)


########################################################################
# State machine initial event types.
########################################################################


class initial_event_types:
    class initial_event_cat_sleeping_eating_drinking(Event):
        pass
    class initial_event_cat_sleeping_eating_drinking__awake(Event):
        pass

########################################################################
# Entity attributes.
# Entity type: Cat
```

(continues on next page)

(continued from previous page)

```
################################################################################


class Entities_Cat_attributes:
    def __init__(self, num_entities):
        self.num_entities = num_entities
        self.height_inches = numpy.full(self.num_entities, 9.0, numpy.float32)
        self.weight_pounds = numpy.full(self.num_entities, 10.0, numpy.float32)


################################################################################
# Entity states.
# Entity type: Cat
################################################################################


class Entities_Cat_states:
    def __init__(self, num_entities):
        self.num_entities = num_entities
        self.cat_sleeping_eating_drinking = Entities_Cat_states_cat_sleeping_eating_
↪drinking(num_entities=self.num_entities)
        self.cat_sleeping_eating_drinking__awake = Entities_Cat_states_cat_sleeping_
↪eating_drinking__awake(num_entities=self.num_entities)


class Entities_Cat_states_cat_sleeping_eating_drinking:
    def __init__(self, num_entities):
        self.num_entities = num_entities
        self.initial_state = numpy.full(self.num_entities, 1, dtype=numpy.int64)
        self.awake = numpy.full(self.num_entities, 0, dtype=numpy.int64)
        self.sleeping = numpy.full(self.num_entities, 0, dtype=numpy.int64)


class Entities_Cat_states_cat_sleeping_eating_drinking__awake:
    def __init__(self, num_entities):
        self.num_entities = num_entities
        self.initial_state = numpy.full(self.num_entities, 1, dtype=numpy.int64)
        self.drinking = numpy.full(self.num_entities, 0, dtype=numpy.int64)
        self.eating = numpy.full(self.num_entities, 0, dtype=numpy.int64)
        self.staring_out_window = numpy.full(self.num_entities, 0, dtype=numpy.int64)


class Entities_Cat_state_tracking:
    def __init__(self):
        self.cat_sleeping_eating_drinking__awake__eating = StateTracking(
            track_n=None,
            track_new=None,
            track_cumulative=0
        )
        self.cat_sleeping_eating_drinking__awake = StateTracking(
            track_n=None,
            track_new=None,
```

(continues on next page)

(continued from previous page)

```
            track_cumulative=None
        )
        self.cat_sleeping_eating_drinking__awake__drinking = StateTracking(
            track_n=None,
            track_new=None,
            track_cumulative=None
        )
        self.cat_sleeping_eating_drinking__awake__staring_out_window = StateTracking(
            track_n=None,
            track_new=None,
            track_cumulative=None
        )
        self.cat_sleeping_eating_drinking__sleeping = StateTracking(
            track_n=None,
            track_new=None,
            track_cumulative=None
        )


################################################################################
# Entity last state entry times.
# Entity type: Cat
################################################################################


class Entities_Cat_last_state_entry_times_cat_sleeping_eating_drinking:
    def __init__(self, num_entities):
        self.num_entities = num_entities
        self.awake = numpy.full(self.num_entities, numpy.nan, dtype=numpy.int64)
        self.initial_state = numpy.full(self.num_entities, numpy.nan, dtype=numpy.int64)
        self.sleeping = numpy.full(self.num_entities, numpy.nan, dtype=numpy.int64)


class Entities_Cat_last_state_entry_times_cat_sleeping_eating_drinking__awake:
    def __init__(self, num_entities):
        self.num_entities = num_entities
        self.drinking = numpy.full(self.num_entities, numpy.nan, dtype=numpy.int64)
        self.eating = numpy.full(self.num_entities, numpy.nan, dtype=numpy.int64)
        self.initial_state = numpy.full(self.num_entities, numpy.nan, dtype=numpy.int64)
        self.staring_out_window = numpy.full(self.num_entities, numpy.nan, dtype=numpy.
↪int64)


class Entities_Cat_last_state_entry_times:
    def __init__(self, num_entities):
        self.num_entities = num_entities
        self.cat_sleeping_eating_drinking = Entities_Cat_last_state_entry_times_cat_
↪sleeping_eating_drinking(num_entities=self.num_entities)
        self.cat_sleeping_eating_drinking__awake = Entities_Cat_last_state_entry_times_
↪cat_sleeping_eating_drinking__awake(num_entities=self.num_entities)


```

(continues on next page)

(continued from previous page)

```
180 ##########################################################################
181 # Entity state entry counters.
182 # Entity type: Cat
183 ##########################################################################
184
185
186 class Entities_Cat_state_entry_counters_cat_sleeping_eating_drinking:
187     def __init__(self, num_entities):
188         self.num_entities = num_entities
189         self.awake = numpy.full(self.num_entities, 0, dtype=numpy.uint64)
190         self.initial_state = numpy.full(self.num_entities, 0, dtype=numpy.uint64)
191         self.sleeping = numpy.full(self.num_entities, 0, dtype=numpy.uint64)
192
193
194 class Entities_Cat_state_entry_counters_cat_sleeping_eating_drinking__awake:
195     def __init__(self, num_entities):
196         self.num_entities = num_entities
197         self.drinking = numpy.full(self.num_entities, 0, dtype=numpy.uint64)
198         self.eating = numpy.full(self.num_entities, 0, dtype=numpy.uint64)
199         self.initial_state = numpy.full(self.num_entities, 0, dtype=numpy.uint64)
200         self.staring_out_window = numpy.full(self.num_entities, 0, dtype=numpy.uint64)
201
202
203 class Entities_Cat_state_entry_counters:
204     def __init__(self, num_entities):
205         self.num_entities = num_entities
206         self.cat_sleeping_eating_drinking = Entities_Cat_state_entry_counters_cat_
    ↪sleeping_eating_drinking(num_entities=self.num_entities)
207         self.cat_sleeping_eating_drinking__awake = Entities_Cat_state_entry_counters_cat_
    ↪sleeping_eating_drinking__awake(num_entities=self.num_entities)
208
209
210 ##########################################################################
211 # Entity instances.
212 # Entity type: Cat
213 ##########################################################################
214
215
216 class Entities_Cat:
217
218     def __init__(self, num_entities: numpy.uint64, enable_logging: bool=None, info_
    ↪interval_ticks: numpy.float64=None):
219         if enable_logging is None:
220             enable_logging = False
221         if info_interval_ticks is None:
222             info_interval_ticks = 100.0
223         self.enable_logging = enable_logging
224         self.info_interval_ticks = info_interval_ticks
225         self.last_info_time_ticks = numpy.finfo(numpy.float64).min
226         self.num_entities = num_entities
227         self.initialize_q()
228         self.initialize_rng()
```

(continues on next page)

(continued from previous page)

```
        self.initialize_state()
        self.initialize_event_type_transition_lookup()
        self.initialize_action_mappings()
        self.initialize_entity_type()
        self.initialize_entity_type_mappings()
        self.initialize_lookup_tables()
        self.register_lookup_tables()
        self.initialize_distributions()
        self.register_distributions()
        return

    def initialize_state(self):
        self.initialize_identity()
        self.initialize_entity_attributes()
        self.initialize_entity_states()
        self.initialize_entity_state_tracking()
        self.initialize_time()
        self.initialize_state_machines()
        self.initialize_state_data_hooks()

    def initialize_identity(self):
        self.entity_id = numpy.arange(self.num_entities, dtype=numpy.uint64)
        pass

    def initialize_time(self):
        min_time = numpy.finfo(numpy.float64).min
        self.set_time_ticks(time_ticks=min_time)

    def initialize_q(self):
        self.q = []
        self.next_event_id = numpy.uint64(0)

    def set_seed(self, seed):
        self.initialize_rng(seed=seed)

    def get_seed(self):
        # TODO: this might not be the best value for 'high'.
        return self.seed_rng.integers(low=0, high=numpy.iinfo(numpy.int32).max, size=1)

    def initialize_rng(self, seed=12345):
        self.seed_rng = numpy.random.default_rng(seed=seed)

        # The events rng is used internally in schedule_event_zzz() methods.
        events_seed = self.get_seed()
        self.events_rng = numpy.random.default_rng(events_seed)

        # Now that the seed rng is set up, we have to re-initialize all the distribution␣
↪rngs,
        # they will take new seeds from the seed rng. And then they have to be re-
↪registered
        # with the action hooks.
        self.initialize_distributions()
```

(continues on next page)

(continued from previous page)

```
        self.register_distributions()

    def get_next_event_id(self):
        event_id = self.next_event_id
        self.next_event_id = self.next_event_id + numpy.uint64(1)
        return event_id

    def schedule_event_rel(self, event_obj: Event, time_ticks_rel: numpy.float64 = numpy.float64(0.0), entity_ids: 'numpy.array[numpy.uint64]' = None):
        event_time_ticks_abs = self.get_time_ticks() + time_ticks_rel
        self.schedule_event_abs(event_obj=event_obj, time_ticks_abs=event_time_ticks_abs, entity_ids=entity_ids)

    def schedule_events_abs(self,
                            event_obj: Event,
                            times_ticks_abs: 'numpy.array[numpy.float64]',
                            entity_ids: 'numpy.array[numpy.uint64]' = None):
        # Break the entity_ids into sub-groups that share the same times_ticks_abs.
        # And then schedule the sub-groups together.

        # This creates effectively a list of arrays of indices, where each array contains indices that
        # have the same times_ticks_abs.
        #
        # [ array([5, 13, 16]), array([1, 8]), array([0, 3, 12, 19, 21]), ... ]
        #
        # Subsequent .split() operations pick out the values at these indices.
        time_ticks_group_by = numpy_indexed.group_by(times_ticks_abs)

        # The two lists created below by contract will have the same number of entries.
        #
        # The unique time ticks values.
        time_ticks = time_ticks_group_by.unique
        # The entity ids for each corresponding time ticks value.
        entity_id_groups = time_ticks_group_by.split(entity_ids)

        # Schedule the appropriate entities for each unique time ticks value.
        for i, (time, this_time_entity_ids) in enumerate(zip(time_ticks, entity_id_groups)):
            self.schedule_event_abs(event_obj=event_obj,
                                    time_ticks_abs=time,
                                    entity_ids=this_time_entity_ids)

    def _schedule_event_abs(self, event_obj: Event, time_ticks_abs: numpy.float64 = numpy.float64(0.0), entity_ids: 'numpy.array[numpy.uint64]' = None):
        # We ignore an empty list for entity_ids.
        if entity_ids is not None and len(entity_ids) == 0:
            if self.enable_logging: log.debug(f'Event was scheduled with no target entity_ids: [{event_obj} @ {time_ticks_abs}]')
            return

        if entity_ids is None:
```

(continues on next page)

(continued from previous page)

```
            # We pass it along as None; this is handled as a special case in dispatch_event().
            pass

        event_id = self.get_next_event_id()
        scheduled_event = ScheduledEvent(time_ticks=numpy.float64(time_ticks_abs),
                                         event_id=event_id,
                                         event_obj=event_obj,
                                         entity_ids=entity_ids)
        if self.enable_logging: log.debug(f'Scheduling event: [{scheduled_event}]')
        heappush(self.q, scheduled_event)

    def schedule_event_abs(self,
                           event_obj: Event,
                           time_ticks_abs: numpy.float64 = numpy.float64(0.0),
                           entity_ids: 'numpy.array[numpy.uint64]' = None):

        if isinstance(time_ticks_abs, numpy.ndarray):
            self.schedule_events_abs(event_obj=event_obj,
                                     times_ticks_abs=time_ticks_abs,
                                     entity_ids=entity_ids)
        else:
            self._schedule_event_abs(event_obj=event_obj,
                                     time_ticks_abs=time_ticks_abs,
                                     entity_ids=entity_ids)

    def schedule_event_binomial(self,
                                time_ticks_abs: Union[numpy.float64, 'numpy.array[numpy.float64]'],
                                event_a: Event,
                                event_b: Event,
                                p_a: numpy.float64,
                                entity_ids: 'numpy.array[numpy.uint64]'):

        # An empty list for 'entity_ids' here is ignored.
        if entity_ids is not None and len(entity_ids) == 0:
            if self.enable_logging: log.debug(f'Binomial event was scheduled with no target entity_ids: [({event_a}, {event_b}) @ {time_ticks_abs}]')
            return

        # None is interpreted as all.
        if entity_ids is None:
            entity_ids = numpy.arange(self.num_entities)

        if event_a is None and event_b is None:
            raise RuntimeError('At least one of of event_a or event_b must be provided to schedule_event_binomial()')

        trials = self.events_rng.binomial(1, p_a, len(entity_ids))
        entity_inds_a = numpy.nonzero(trials == 1)[0]
        entity_inds_b = numpy.nonzero(trials == 0)[0]
        entity_ids_a = entity_ids[entity_inds_a]
```

(continues on next page)

(continued from previous page)

```
        entity_ids_b = entity_ids[entity_inds_b]

        if len(entity_ids_a) > 0 and event_a is not None:
            if isinstance(time_ticks_abs, numpy.ndarray):
                time_ticks_abs_a = time_ticks_abs[entity_inds_a]
                self.schedule_events_abs(event_obj=event_a,
                                         times_ticks_abs=time_ticks_abs_a,
                                         entity_ids=entity_ids_a)
            else:
                self.schedule_event_abs(event_obj=event_a,
                                        time_ticks_abs=time_ticks_abs,
                                        entity_ids=entity_ids_a)

        if len(entity_ids_b) > 0 and event_b is not None:
            if isinstance(time_ticks_abs, numpy.ndarray):
                time_ticks_abs_b = time_ticks_abs[entity_inds_b]
                self.schedule_events_abs(event_obj=event_b,
                                         times_ticks_abs=time_ticks_abs_b,
                                         entity_ids=entity_ids_b)
            else:
                self.schedule_event_abs(event_obj=event_b,
                                        time_ticks_abs=time_ticks_abs,
                                        entity_ids=entity_ids_b)

    def schedule_event_trials(self,
                              time_ticks_abs: Union[numpy.float64, 'numpy.array[numpy.
↪float64]'],
                              event: Event,
                              p: 'numpy.array[numpy.float64]',
                              entity_ids: 'numpy.array[numpy.uint64]'):

        if entity_ids is not None and len(entity_ids) == 0:
            if self.enable_logging: log.debug(f'Trials event was scheduled with no␣
↪target entity_ids: [({event}) @ {time_ticks_abs}]')
            return

        # None is interpreted as all.
        if entity_ids is None:
            entity_ids = numpy.arange(self.num_entities)

        if event is None:
            raise RuntimeError('The event parameter must be provided to schedule_event_
↪trials()')

        draws = self.events_rng.random(size=len(entity_ids))
        trials = draws < p
        # Indices into entity_ids for trials resulting in the outcome as specified by␣
↪the probability.
        outcome_trial_indices = trials.nonzero()[0]
        if len(outcome_trial_indices) > 0:
            outcome_trial_entity_ids = entity_ids[outcome_trial_indices]

```

(continues on next page)

(continued from previous page)

```
            if isinstance(time_ticks_abs, numpy.ndarray):
                # Pull the times associated with each entity_id, which we find by outcome_trial_indices.
                time_ticks_abs_a = time_ticks_abs[outcome_trial_indices]
                self.schedule_events_abs(event_obj=event,
                                         times_ticks_abs=time_ticks_abs_a,
                                         entity_ids=outcome_trial_entity_ids)
            else:
                self.schedule_event_abs(event_obj=event,
                                        time_ticks_abs=time_ticks_abs,
                                        entity_ids=outcome_trial_entity_ids)

    def set_time_ticks(self, time_ticks: numpy.int64):
        action.time_ticks = time_ticks

    def get_time_ticks(self) -> numpy.float64:
        return action.time_ticks

    def get_events_at_next_time(self, q: List[ScheduledEvent]) -> Dict[Type, List[ScheduledEvent]]:
        '''Returns a map of lists of events that are co-scheduled for the next scheduled event time, keyed by event
        type.'''
        event_map = dict()
        if len(q) == 0:
            return None

        next_event = heappop(q)
        next_event_type = type(next_event.event_obj)
        event_map[next_event_type] = [next_event]
        while len(q) > 0 and q[0].time_ticks == next_event.time_ticks:
            event: ScheduledEvent = heappop(q)
            event_type = type(event.event_obj)
            if event_type in event_map.keys():
                event_map[event_type].append(event)
            else:
                event_map[event_type] = [event]

        return event_map


    def step(self, to_time_ticks: numpy.float64):
        '''Run state machines up to time to_time_ticks. Returns False if there are no more events left in the
        queue, otherwise True.'''
        self.check_multiprocessing_shenanigans()

        if len(self.q) == 0:
            return False

        next_event_time = self.q[0].time_ticks
        while next_event_time <= to_time_ticks:
```

(continues on next page)

(continued from previous page)

```
            if next_event_time - self.last_info_time_ticks >= self.info_interval_ticks:
                if self.enable_logging: log.info(f't = {next_event_time}')
                self.last_info_time_ticks = next_event_time

            next_events = self.get_events_at_next_time(self.q)

            next_event: ScheduledEvent
            for next_event_type in next_events.keys():
                events = next_events[next_event_type]
                # Merge all the events of the same type together.
                if len(events) > 1:
                    if self.enable_logging: log.debug(f'Merging event ids: [{f"{[e.event_
↪id for e in events]},"}]')
                    merge_target = events[0]
                    if merge_target.entity_ids is None:
                        # This indicates 'all entity_ids', so we don't need to merge, just␣
↪leave it alone.
                        pass
                    else:
                        for merge_source in events[1:]:
                            merge_target.entity_ids = numpy.append(merge_target.entity_
↪ids, merge_source.entity_ids)
                    next_event = merge_target
                    if next_event.entity_ids is not None:
                        # TODO: should we be using a set for entity_ids ?
                        numpy.ndarray.sort(next_event.entity_ids)
                else:
                    next_event = events[0]

                if self.enable_logging: log.debug(f'Processing event: [{next_event}]')

                event_time_ticks = next_event.time_ticks
                self.set_time_ticks(time_ticks=event_time_ticks)

                self.dispatch_event(next_event)

            if len(self.q) == 0:
                return False

            next_event_time = self.q[0].time_ticks

        return True

    def dispatch_event(self, scheduled_event: ScheduledEvent):

        time_ticks = scheduled_event.time_ticks

        # Figure out which applicable transitions.
        all_transitions = self.event_type_transition_lookup
        event_obj = scheduled_event.event_obj
        applicable_transitions = all_transitions[type(event_obj)]
```

(continues on next page)

(continued from previous page)

```
518
519         # For each applicable transition:
520         transition: SourceTargetPair
521         for transition in applicable_transitions:
522
523             # Build mask for targeting scheduled_event.entity_ids
524             if scheduled_event.entity_ids is None:
525                 entity_id_mask = numpy.full(self.num_entities, 1)
526             else:
527                 entity_id_mask = numpy.full(self.num_entities, 0)
528                 # This masking relies on entity_id is also the index into the arrays.
529                 entity_id_mask[scheduled_event.entity_ids] = 1
530
531             # Figure out which entity ids are currently in the appropriate source state,
532             # and are in the entity_id_mask.
533             applicable_entity_ids = numpy.nonzero((transition.source == 1)
534                                                   &
535                                                   (entity_id_mask == 1)
536                                                   )[0]
537
538             # Get the target state data hooks.
539             target_state_data_hooks: StateData = self.state_data_hooks[type(transition.
    ↪target_state_definition)]
540             target_state_entry_counters = target_state_data_hooks.state_entry_counters
541             target_state_last_entry_time_ticks = target_state_data_hooks.last_state_
    ↪entry_time_ticks
542             target_state_tracking = target_state_data_hooks.state_tracking
543
544             # Transition state.  In addition to updating the state tracking variables,␣
    ↪we also:
545             #   - Run the source state exit actions, with the appropriate entity ids␣
    ↪that are transitioning.
546             #   - Run the target state entry actions, again with the appropriate entity␣
    ↪ids.
547             #   - Update the associated state hooks.
548
549             transitioning_entities: action.entity_match
550
551             with action.entity_match(entity_type=self.entity_type(),
552                                      entity_set=applicable_entity_ids,
553                                      criteria=None
554             ) as transitioning_entities:
555
556                 # Take the applicable entities out of the source state.
557                 transition.source_state_definition.exit_action(transitioning_entities)
558                 transition.source[transitioning_entities.filtered_entity_ids] = 0
559
560                 # Put the applicable entities into the target state.
561                 transition.target[transitioning_entities.filtered_entity_ids] = 1
562
563                 # Update state hooks.
564                 target_state_entry_counters[transitioning_entities.filtered_entity_ids]␣
    ↪+= 1
```

(continues on next page)

(continued from previous page)

```
                target_state_last_entry_time_ticks[transitioning_entities.filtered_
↪entity_ids] = time_ticks

                # Update state tracking
                if target_state_tracking.track_n is not None:
                    entities_in_target_state_now = numpy.nonzero(transition.target==1)[0]
                    num_entities_in_target_state_now = len(entities_in_target_state_now)
                    target_state_tracking.track_n.append(TimedValue(
                        time_ticks=time_ticks,
                        value=num_entities_in_target_state_now
                    ))

                if target_state_tracking.track_new is not None:
                    num_new_entities_in_state = len(transitioning_entities.filtered_
↪entity_ids)
                    target_state_tracking.track_new.append(TimedValue(
                        time_ticks=time_ticks,
                        value=num_new_entities_in_state
                    ))

                if target_state_tracking.track_cumulative is not None:
                    num_new_entities_in_state = len(transitioning_entities.filtered_
↪entity_ids)
                    target_state_tracking.track_cumulative += num_new_entities_in_state

                # Execute target state entry action.
                transition.target_state_definition.entry_action(transitioning_entities)

                # If the target state has a sub-state-machine, schedule the␣
↪initialization event for it now.
                if target_state_data_hooks.is_state_machine:
                    self.schedule_event_rel(event_obj=target_state_data_hooks.substate_
↪initialization_event_type(),
                                            time_ticks_rel=numpy.float64(0.0),
                                            entity_ids=transitioning_entities.filtered_
↪entity_ids)

    def check_multiprocessing_shenanigans(self):
        # This is a workaround for module-level stuff doesn't survive sporking on windows
        # when using multiprocessing.
        # We're using the item in the 'if' statement as the 'canary' to detect that things␣
↪need to be re-initialized.
        if len(action.action_state_hooks.keys()) == 0:
            self.initialize_action_mappings()
            self.register_lookup_tables()
            self.initialize_time()
            self.register_distributions()
            self.initialize_entity_type_mappings()

    def initialize_entity_attributes(self):
        self.attributes = Entities_Cat_attributes(num_entities=self.num_entities)

```

(continues on next page)

(continued from previous page)

```
    def initialize_entity_states(self):
        self.states = Entities_Cat_states(num_entities=self.num_entities)
        self.last_state_entry_time_ticks = Entities_Cat_last_state_entry_times(num_entities=self.num_entities)
        self.state_entry_counters = Entities_Cat_state_entry_counters(num_entities=self.num_entities)


    def initialize_entity_state_tracking(self):
        self.state_tracking = Entities_Cat_state_tracking()

    def initialize_state_data_hooks(self):
        self.state_data_hooks = dict()

        # State machine: cat_sleeping_eating_drinking
        self.state_data_hooks[model.cat_sleeping_eating_drinking.awake] = StateData(
            state=model.cat_sleeping_eating_drinking.awake,
            state_entry_counters=self.state_entry_counters.cat_sleeping_eating_drinking.awake,
            last_state_entry_time_ticks=self.last_state_entry_time_ticks.cat_sleeping_eating_drinking.awake,
            state_tracking=self.state_tracking.cat_sleeping_eating_drinking__awake,
            is_state_machine=True,
            substate_initialization_event_type=initial_event_types.initial_event_cat_sleeping_eating_drinking__awake
        )

        # State machine: cat_sleeping_eating_drinking.awake
        self.state_data_hooks[model.cat_sleeping_eating_drinking.awake.drinking] = StateData(
            state=model.cat_sleeping_eating_drinking.awake.drinking,
            state_entry_counters=self.state_entry_counters.cat_sleeping_eating_drinking__awake.drinking,
            last_state_entry_time_ticks=self.last_state_entry_time_ticks.cat_sleeping_eating_drinking__awake.drinking,
            state_tracking=self.state_tracking.cat_sleeping_eating_drinking__awake__drinking,
            is_state_machine=False,
            substate_initialization_event_type=None
        )
        self.state_data_hooks[model.cat_sleeping_eating_drinking.awake.eating] = StateData(
            state=model.cat_sleeping_eating_drinking.awake.eating,
            state_entry_counters=self.state_entry_counters.cat_sleeping_eating_drinking__awake.eating,
            last_state_entry_time_ticks=self.last_state_entry_time_ticks.cat_sleeping_eating_drinking__awake.eating,
            state_tracking=self.state_tracking.cat_sleeping_eating_drinking__awake__eating,
            is_state_machine=False,
            substate_initialization_event_type=None
        )
```

(continues on next page)

(continued from previous page)

```
649         self.state_data_hooks[model.cat_sleeping_eating_drinking.awake.staring_out_
    ↪window] = StateData(
650             state=model.cat_sleeping_eating_drinking.awake.staring_out_window,
651             state_entry_counters=self.state_entry_counters.cat_sleeping_eating_drinking__
    ↪awake.staring_out_window,
652             last_state_entry_time_ticks=self.last_state_entry_time_ticks.cat_sleeping_
    ↪eating_drinking__awake.staring_out_window,
653             state_tracking=self.state_tracking.cat_sleeping_eating_drinking__awake__
    ↪staring_out_window,
654             is_state_machine=False,
655             substate_initialization_event_type=None
656         )
657         self.state_data_hooks[model.cat_sleeping_eating_drinking.sleeping] = StateData(
658             state=model.cat_sleeping_eating_drinking.sleeping,
659             state_entry_counters=self.state_entry_counters.cat_sleeping_eating_drinking.
    ↪sleeping,
660             last_state_entry_time_ticks=self.last_state_entry_time_ticks.cat_sleeping_
    ↪eating_drinking.sleeping,
661             state_tracking=self.state_tracking.cat_sleeping_eating_drinking__sleeping,
662             is_state_machine=False,
663             substate_initialization_event_type=None
664         )
665 
666     def initialize_event_type_transition_lookup(self):
667         self.event_type_transition_lookup: Dict[type, List[SourceTargetPair]] = dict()
668 
669         self.event_type_transition_lookup[model.not_tired_event] = [
670             SourceTargetPair(source=self.states.cat_sleeping_eating_drinking.sleeping,
671                              target=self.states.cat_sleeping_eating_drinking.awake,
672                              source_state_definition=model.cat_sleeping_eating_drinking.
    ↪sleeping(),
673                              target_state_definition=model.cat_sleeping_eating_drinking.
    ↪awake()
674             ),
675         ]
676 
677         self.event_type_transition_lookup[model.tired_event] = [
678             SourceTargetPair(source=self.states.cat_sleeping_eating_drinking.awake,
679                              target=self.states.cat_sleeping_eating_drinking.sleeping,
680                              source_state_definition=model.cat_sleeping_eating_drinking.
    ↪awake(),
681                              target_state_definition=model.cat_sleeping_eating_drinking.
    ↪sleeping()
682             ),
683         ]
684 
685         self.event_type_transition_lookup[model.hungry_event] = [
686             SourceTargetPair(source=self.states.cat_sleeping_eating_drinking__awake.
    ↪staring_out_window,
687                              target=self.states.cat_sleeping_eating_drinking__awake.
    ↪eating,
688                              source_state_definition=model.cat_sleeping_eating_drinking.
    ↪awake.staring_out_window(),
```

(continues on next page)

(continued from previous page)

```
                                target_state_definition=model.cat_sleeping_eating_drinking.awake.eating()
            ),
        ]

        self.event_type_transition_lookup[model.not_hungry_event] = [
            SourceTargetPair(source=self.states.cat_sleeping_eating_drinking__awake.eating,
                                target=self.states.cat_sleeping_eating_drinking__awake.staring_out_window,
                                source_state_definition=model.cat_sleeping_eating_drinking.awake.eating(),
                                target_state_definition=model.cat_sleeping_eating_drinking.awake.staring_out_window()
            ),
        ]

        self.event_type_transition_lookup[model.not_thirsty_event] = [
            SourceTargetPair(source=self.states.cat_sleeping_eating_drinking__awake.drinking,
                                target=self.states.cat_sleeping_eating_drinking__awake.staring_out_window,
                                source_state_definition=model.cat_sleeping_eating_drinking.awake.drinking(),
                                target_state_definition=model.cat_sleeping_eating_drinking.awake.staring_out_window()
            ),
        ]

        self.event_type_transition_lookup[model.thirsty_event] = [
            SourceTargetPair(source=self.states.cat_sleeping_eating_drinking__awake.staring_out_window,
                                target=self.states.cat_sleeping_eating_drinking__awake.drinking,
                                source_state_definition=model.cat_sleeping_eating_drinking.awake.staring_out_window(),
                                target_state_definition=model.cat_sleeping_eating_drinking.awake.drinking()
            ),
        ]

        self.event_type_transition_lookup[initial_event_types.initial_event_cat_sleeping_eating_drinking__awake] = [
            SourceTargetPair(source=self.states.cat_sleeping_eating_drinking__awake.initial_state,
                                target=self.states.cat_sleeping_eating_drinking__awake.staring_out_window,
                                source_state_definition=model.cat_sleeping_eating_drinking.awake.initial_state(),
                                target_state_definition=model.cat_sleeping_eating_drinking.awake.staring_out_window()
            ),
```

(continues on next page)

(continued from previous page)

```
        ]

        self.event_type_transition_lookup[initial_event_types.initial_event_cat_sleeping_eating_drinking] = [
            SourceTargetPair(source=self.states.cat_sleeping_eating_drinking.initial_state,
                             target=self.states.cat_sleeping_eating_drinking.sleeping,
                             source_state_definition=model.cat_sleeping_eating_drinking.initial_state(),
                             target_state_definition=model.cat_sleeping_eating_drinking.sleeping()
            ),
        ]

        self.event_type_transition_lookup[initial_event_types.initial_event_cat_sleeping_eating_drinking__awake] = [
            SourceTargetPair(source=self.states.cat_sleeping_eating_drinking__awake.initial_state,
                             target=self.states.cat_sleeping_eating_drinking__awake.staring_out_window,
                             source_state_definition=model.cat_sleeping_eating_drinking.awake.initial_state(),
                             target_state_definition=model.cat_sleeping_eating_drinking.awake.staring_out_window()
            ),
        ]

    def initialize_action_mappings(self):

        # State machine: cat_sleeping_eating_drinking
        # State: cat_sleeping_eating_drinking.awake
        action.register_action_state_hooks(ActionStateHooks(
            state=model.cat_sleeping_eating_drinking.awake(),
            generated_state=self.states.cat_sleeping_eating_drinking.awake,
            generated_last_entry_time_ticks=self.last_state_entry_time_ticks.cat_sleeping_eating_drinking.awake,
            generated_state_entry_counters=self.state_entry_counters.cat_sleeping_eating_drinking.awake,
            state_tracking=self.state_tracking.cat_sleeping_eating_drinking__awake
        ))

        # State machine: cat_sleeping_eating_drinking.awake
        # State: cat_sleeping_eating_drinking.awake.drinking
        action.register_action_state_hooks(ActionStateHooks(
            state=model.cat_sleeping_eating_drinking.awake.drinking(),
            generated_state=self.states.cat_sleeping_eating_drinking__awake.drinking,
            generated_last_entry_time_ticks=self.last_state_entry_time_ticks.cat_sleeping_eating_drinking__awake.drinking,
            generated_state_entry_counters=self.state_entry_counters.cat_sleeping_eating_drinking__awake.drinking,
            state_tracking=self.state_tracking.cat_sleeping_eating_drinking__awake__drinking
```

(continues on next page)

(continued from previous page)

```
        ))
        # State: cat_sleeping_eating_drinking.awake.eating
        action.register_action_state_hooks(ActionStateHooks(
            state=model.cat_sleeping_eating_drinking.awake.eating(),
            generated_state=self.states.cat_sleeping_eating_drinking__awake.eating,
            generated_last_entry_time_ticks=self.last_state_entry_time_ticks.cat_
↪sleeping_eating_drinking__awake.eating,
            generated_state_entry_counters=self.state_entry_counters.cat_sleeping_eating_
↪drinking__awake.eating,
            state_tracking=self.state_tracking.cat_sleeping_eating_drinking__awake__
↪eating
        ))
        # State: StateMachine.initial_state
        action.register_action_state_hooks(ActionStateHooks(
            state=model.cat_sleeping_eating_drinking.awake.initial_state(),
            generated_state=self.states.cat_sleeping_eating_drinking__awake.initial_
↪state,
            generated_last_entry_time_ticks=self.last_state_entry_time_ticks.cat_
↪sleeping_eating_drinking__awake.initial_state,
            generated_state_entry_counters=self.state_entry_counters.cat_sleeping_eating_
↪drinking__awake.initial_state,
            state_tracking=None
        ))
        # State: cat_sleeping_eating_drinking.awake.staring_out_window
        action.register_action_state_hooks(ActionStateHooks(
            state=model.cat_sleeping_eating_drinking.awake.staring_out_window(),
            generated_state=self.states.cat_sleeping_eating_drinking__awake.staring_out_
↪window,
            generated_last_entry_time_ticks=self.last_state_entry_time_ticks.cat_
↪sleeping_eating_drinking__awake.staring_out_window,
            generated_state_entry_counters=self.state_entry_counters.cat_sleeping_eating_
↪drinking__awake.staring_out_window,
            state_tracking=self.state_tracking.cat_sleeping_eating_drinking__awake__
↪staring_out_window
        ))
        # State: StateMachine.initial_state
        action.register_action_state_hooks(ActionStateHooks(
            state=model.cat_sleeping_eating_drinking.initial_state(),
            generated_state=self.states.cat_sleeping_eating_drinking.initial_state,
            generated_last_entry_time_ticks=self.last_state_entry_time_ticks.cat_
↪sleeping_eating_drinking.initial_state,
            generated_state_entry_counters=self.state_entry_counters.cat_sleeping_eating_
↪drinking.initial_state,
            state_tracking=None
        ))
        # State: cat_sleeping_eating_drinking.sleeping
        action.register_action_state_hooks(ActionStateHooks(
            state=model.cat_sleeping_eating_drinking.sleeping(),
            generated_state=self.states.cat_sleeping_eating_drinking.sleeping,
            generated_last_entry_time_ticks=self.last_state_entry_time_ticks.cat_
↪sleeping_eating_drinking.sleeping,
            generated_state_entry_counters=self.state_entry_counters.cat_sleeping_eating_
↪drinking.sleeping,
```

(continues on next page)

(continued from previous page)

```
            state_tracking=self.state_tracking.cat_sleeping_eating_drinking__sleeping
        ))

    def initialize_entity_type_mappings(self):
        action.register_action_entity_type_hooks(ActionEntityTypeHooks(
            entity_type=model.Cat(),
            entity_attributes=self.attributes,
            schedule_event_method=self.schedule_event_abs,
            schedule_event_binomial_method=self.schedule_event_binomial,
            schedule_event_trials_method=self.schedule_event_trials
        ))

    def initialize_entity_type(self):
        self.entity_type = model.Cat

    def initialize_state_machines(self):
        """Initialize the top-level state machines. Note that we do not initialize sub-
↪state-machines here."""
        self.schedule_event_abs(
            event_obj=initial_event_types.initial_event_cat_sleeping_eating_drinking(),
            time_ticks_abs=0,
            entity_ids=None
        )

    def initialize_lookup_tables(self):
        """Initialize lookup tables."""

    def register_lookup_tables(self):
        """Register lookup tables w/ action handling."""

    def initialize_distributions(self):
        """Initialize distributions."""

    def register_distributions(self):
        """Register distributions with action handling."""

        def uniform(numpy_rng, low, high, num_samples):
            return numpy_rng.uniform(low,high,num_samples)

        def exponential(numpy_rng, rate, num_samples):
            scale = 1 / rate
            return numpy_rng.exponential(scale, num_samples)

        def poisson(numpy_rng, rate, num_samples):
            return numpy_rng.poisson(rate, num_samples)

        def normal(numpy_rng, mean, std, num_samples):
            return numpy_rng.normal(mean, std, num_samples)

        def normal_positive(numpy_rng, mean, std, num_samples):
            return numpy.abs(numpy_rng.normal(mean, std, num_samples))

```

(continues on next page)

(continued from previous page)

```
851         def normal_int(numpy_rng, mean, std, num_samples):
852             return numpy.round(numpy_rng.normal(mean, std, num_samples))
853 
854         def normal_positive_int(numpy_rng, mean, std, num_samples):
855             return numpy.abs(numpy.round(numpy_rng.normal(mean, std, num_samples)))
856 
857         def calculate_lognormal_underlying_parameters(mean, std):
858             """Given the mean and sigma (std deviation) for a lognormal distribution,␣
    ↪calculates the mean
859             and sigma of the underlying normal distribution."""
860             underlying_mean = numpy.log((mean ** 2) / (numpy.sqrt(mean ** 2 + std ** 2)))
861             underlying_sigma = numpy.sqrt(numpy.log(1 + (std ** 2 / mean ** 2)))
862             return underlying_mean, underlying_sigma
863 
864         def log_normal(numpy_rng, mean, std, num_samples):
865             underlying_mean, underlying_sigma = calculate_lognormal_underlying_
    ↪parameters(mean, std)
866             return numpy_rng.lognormal(underlying_mean, underlying_sigma, num_samples)
867 
868         def log_normal_int(numpy_rng, mean, std, num_samples):
869             underlying_mean, underlying_sigma = calculate_lognormal_underlying_
    ↪parameters(mean, std)
870             return numpy.round(numpy_rng.lognormal(underlying_mean, underlying_sigma,␣
    ↪num_samples))
871 
872 
873     def finalize(self):
874         applicable_entity_ids = numpy.arange(self.num_entities)
875         with action.entity_match(entity_type=self.entity_type(),
876                                  entity_set=applicable_entity_ids,
877                                  criteria=None
878         ) as finalizing_entities:
879             model.cat_sleeping_eating_drinking.finalize_action(finalizing_entities)
880 
881 
882         action.log('-------------------------------------')
883         action.log('State Tracking')
884         action.log('-------------------------------------')
885         action.log(beeprint.pp(self.state_tracking, output=False))
886 
887 if __name__ == '__main__':
888 
889     logging.basicConfig(stream=sys.stdout, level=os.environ.get("LOGLEVEL", "INFO"))
890 
891     arg_parser = argparse.ArgumentParser()
892     arg_parser.add_argument("--num_entities", type=int, default=100, help='Number of␣
    ↪entity instances.')
893     arg_parser.add_argument("--enable_logging", type=int, default=0, help='Enable␣
    ↪logging, or not.')
894     arg_parser.add_argument("--rand_seed", type=int, default=1, help='Random number seed.
    ↪')
895     arg_parser.add_argument("--end_time_ticks", type=float, default=10, help='Simulation␣
    ↪end time (ticks).')
```

(continues on next page)

(continued from previous page)

```
    arg_parser.add_argument("--progress_interval_ticks", type=float, default=100, help=
↪'Interval between reporting sim progress (ticks).')
    args = arg_parser.parse_args()

    enable_logging = True if args.enable_logging > 0 else False

    if enable_logging:
        print('Sim Arguments:')
        beeprint.pp(vars(args))

    entities_Cat = Entities_Cat(
        num_entities=args.num_entities,
        enable_logging=enable_logging,
        info_interval_ticks=args.progress_interval_ticks
    )

    entities_Cat.set_seed(args.rand_seed)

    entities_Cat.step(to_time_ticks=args.end_time_ticks)

    entities_Cat.finalize()
```

CHAPTER

THREE

MODELS OF COMPUTATION FOR DYMODETRON SIMULATIONS

3.1 What are "models of computation"?

When you build a model, you use a certain set of underlying concepts to express the model. These underlying concepts are your model of computation. In other words, what are the kinds of things that will be computed, what are the kinds of relationships that they have with one another? What elemental concepts make up the set of widgets you can use to build your model?

For example, you might build a model that consists of a set of coupled first order ordinary differential equations (ODEs). The underlying widgets in the toolbox for constructing that model are:

1. A continuous notion of time.
2. A set of quantities that you'll use to quantify different aspects of the system. Sometimes these quantities are called states.
3. Equations to express the rate of change of each state quantity as a function of the other quantities in the system, and time.
4. Usually, a number of parameters are present within the equations.
5. You'll have a value or range of values for each parameter.
6. For each quantity in the system, you'll have an initial value or range of values for the state quantity and its initial rate of change.

Those concepts together make up a model of computation for ODE systems. Depending on the system you are modeling and the questions that you want to ask your model, this underlying model of computation may be perfectly appropriate, or it may be completely useless.

Another example of a model of computation is the one that Dymodetron supports, state machines, *described below*.

3.2 State Machines

Dymodetron supports a single model of computation: state machines. This section briefly summarizes this model of computation. More details about each piece of the puzzle are available *in the section on modeling constructs*.

Note: The word 'state' in 'state machine' does not mean exactly the same thing it means in the *ODE example we discussed earlier*. But, it is similar in the sense that states are a way of describing/quantifying a simulated entity. In Dymodetron state machines, we describe/quantify simulated entities using two distinct concepts: states, and entity attributes. Both are discussed in this section.

Dymodetron state machines are an interpretation of the concept of Harel statecharts, which is a formalism for describing complex systems. They are also similar to UML state machines, which themselves were built on Harel's statechart ideas. Dymodetron supports hierarchical state machines but does not support Harel's "orthogonal regions", nor does it support "history" states (although with some work *it could*).

There's a description of an example Dymodetron state machine *in the overview*. We repeat the illustration of the example diagram below.

3.2.1 Entities

Entities are the individual nouns in your model, which will be described as having attributes and states. Your model may have any number of entities, within computing resource constraints.

For example, your entities might be cats.

3.2.2 Entity Attributes

Entity attributes are the quantifiable, measurable characteristics of your entities. Each individual entity has its own value for each entity attribute.

For example, you might measure the weight and height of your cats.

3.2.3 States

States are the atomic unit of the description of the behaviors of your entities. An individual entity is described as being in a discrete set of states at any given time. An individual state is a binary proposition: an entity is either in that state, or it's not. The model might have any number of states that an entity can be in. As time goes on, the set of states that an entity is in will change. This happens via state transitions triggered by events. Events can be generated by state entry and exit actions. When entities transition between states, the entry and exit of states can trigger follow-on transitions; this ongoing process propagates the behavior of the model.

States are organized into state machines. There can be multiple state machines, each describing different aspects of the behavior of the entities. The multiple state machines execute concurrently.

For example, you might have a state machine to describe the conscious behavior of the cat performing its daily routine of sleeping, eating, and drinking. Separately, you might have a state machine describing the evolution of the cat's inner dialog, which goes on irrespective of the conscious daily cat routine, and which may be alternating back and forth between instinctive ancestral memories of ancient egypt and ruminations on better places to hide a lizard in the house where you'll be most surprised and disgusted when you find it.

In the example state machine diagram above, the states a cat can be in are: 'sleeping', 'awake', 'staring_out_window', 'eating', and 'drinking'. Due to the hierarchical nature of this particular state machine, if a cat is in state 'staring_out_window', 'eating', or 'drinking', it is also simultaneously in state 'awake'.

3.2.4 Sub-state-machines

A state can have a state machine within it. When the state is entered, the initial transition of the sub-state-machine fires, and the enclosed state machine can begin responding to events by transitioning between states in the sub-state-machine. This is a key feature for managing the complexity of the model while still enabling interesting, useful models.

In the example state machine above, we model a cat as either "asleep' or "awake". While it's awake, it has additional states that it can transition in and out of as events happen.

3.2.5 Events

Events cause transitions to trigger, causing entities to transition between states. Events are generated by action statements within state entry and exit actions. Events are targeted at a subset of entities in the system (one, many, or all).

In the example state machine above, the events are: 'initial_event', 'tired_event', 'not_tired_event', 'hungry_event', 'not_hungry_event', 'thirsty_event', and 'not_thirsty_event'. Note that 'initial_event' is a special event that is present in every state machine, that takes the state machine from the initial state (black dot) to a modeler-defined state.

3.2.6 Transitions

Transitions are how entities can leave one state and enter another state. Transitions are triggered by a particular event, and define the source and target states.

When a particular event "arrives" at an entity, at a given time, the entity may or may not have any valid transitions to be triggered at that time. In this case, the event is ignored.

In the example state machine above, if the cat is in state 'sleeping', then when 'not_tired_event' arrives at a given cat entity instance, that cat will transition to state 'awake'. If the same event arrives at a cat while it is in state 'awake', no transitions will occur, and the event is ignored.

3.2.7 Action statements

Action statements are sequences of logic and calculations that run within state entry and exit actions. Action statements can implement numerical calculations to update the entity attributes. They can also be used to generate events on subsets (1, many, or all) of the entities in the system.

3.2.8 State Entry and Exit Actions

A state can be defined to have action statements executed when the state is entered, or when it is exited, or both. Those actions can result in updates to entity attribute values, and/or the generation of additional events.

3.2.9 Time

State machines have a discrete time model. Events are scheduled to happen at a certain time. Once all the events at that time have occurred, the simulation time progresses to the time of the next event. So time skips along from one event time to the next, until some simulation termination condition is met.

Time is measured in "ticks". "Ticks" are whatever you want them to be: years, days, hours, minutes, seconds, nanoseconds, etc. Just make sure to be consistent!

3.3 Heterogeneous models of computation

Dymodetron doesn't support this yet, but has *aspirations to support heterogeneous models of computation*. This would enable the construction of models that use a heterogeneous mix of models of computation, integrated together. This is useful in that it would allow user models to express different kind of system behavior in the manner that is most useful and efficient for the question/topic at hand.

Mixing models of computation together requires addressing two key integration problems:

1. How to integrate the different notions of time present in the different models of computation?
    - For example, if integrating a *"Gillespie" stochastic simulation algorithm (SSA)* model of compute with an *ODE* model of compute, one must tie together the SSA time propagation algorithm with the ODE solver time propagation algorithm.
2. How to integrate behaviors together between the two models of computation, so that models can be built of elements that use these different widgets but are still able to interact with each other?
    - For example, if integrating an ODE model of compute with a state machine model of compute, how can state machine events be used to change ODE model parameters? Similarly, how can ODE state (or rate) thresholds be used to trigger events in a state machine?

This idea of models relying on a heterogeneous mix of models of computation isn't new, and it has been explored. For example, here's a book on the topic. And here's a widely used commercial tool that implements the idea.

CHAPTER

FOUR

MODELING CONSTRUCTS

This section describes the constructs that can be used to build Dymodetron models.

The modeling constructs are summarized in the table below.

Table 1: Modeling Constructs Summary

Modeling construct	Description
Model	An entire model containing other model constructs.
Model Description	The name of the model and a textual summary.
Model Parameters	The scalar values, lookup table, and parameterized random distributions that affect the behavior of the model.
Scalar Parameter	A scalar value that affects the behavior of the model.
Lookup Table	A lookup table that affects the behavior of the model.
Random Distribution	A parameterized random distribution that affects the behavior of the model.
Entity Type	A kind of object described by the model.
Entity Attribute	A measurable characteristic of an entity type.
Entity Instance	An individual, identifiable realization of an entity type, with concrete values of each entity attribute and state.
Entity Set	A collection of entity instances.
State Machine	A description of the behavior of an entity type, in the form of individual states and reactions to events.
State	A binary (true, false) description of an entity as having some condition or being in some mode or circumstance.
Event Type	A type of event that can occur, causing a state machine to transition between states.
Event	An individual realization of an event type, that can cause specific entity instances to transition between states.
Transition	A reaction to an event, that causes an entity to stop being in one state, and start being in another state.
Entry Action	A sequence of action statements that occurs upon entry to a state.
Exit Action	A sequence of action statements that occurs prior to exit from a state.
Action Statement	An expression that generates events on entity instances, or modifies entity attributes, or intermediate logic leading up to one of those outcomes.

4.1 Overview

The figure below provides an overview of how the modeling constructs fit together. The boxes are dymodetron constructs. The lines indicate relationships between constructs. The "diamonds" indicate containment: the construct on the "diamond" side contains the construct on the other side of the relationship. Where the arrows have labels, read from the base of the arrow, across the relationship, and to the other side. For example `State Machine describes behavior of EntityType`. Quantifiers such as `0..*`, `1`, `2`, indicate the multiplicity of the relationship: one-to-one, one-to-many, zero-or-more, etc.

4.2 Model

A Dymodetron model is a single self-contained set of Dymodetron model constructs used to describe some system in order to ask some questions.

A Dymodetron Model is a python file containing instances of all the rest of the Dymodetron constructs. *Here's an example*.

4.3 Entity Type

An entity type is a kind of thing that your model describes. Measurable characteristics of an entity type are quantified by one or more *entity attributes*. The behavior of an entity type is described by one or more *state machines*. Individual realizations of an entity type are called *entity instances*. A simulation of a model can have many instances of the same entity type. Each instance has its own unique identifier, its own set of values for each of the entity type's entity attributes, and its own "running copy" of each of the entity type's state machines.

You create an entity type by creating a class that sub-classes the Dymodetron `EntityType` class.

```
from dymodetron import EntityType

# An entity is defined by creating a class that sub-classes 'EntityType'.
class Cat(EntityType):
    # It doesn't hurt to put a docstring on your new entity type.
    """Cats are a type of animal that is neither dog, duck, nor lizard."""
```

4.4 Entity Instance

An entity instance is an individual realization of a given entity type. Each entity instance has its own unique identifier, its own copy of each of the entity attributes for the entity type, and its own "running copy" of each of the state machines for the entity type.

Normally you don't need to create entity instances yourself. The Dymodetron engine will take care of this, after you tell it how many you want.

4.5 Entity Attribute

An entity attribute quantifies some measurable aspect of an entity.

You declare the entity attributes within your entity type class, by using `EntityAttribute`. An entity attribute has a name, a type, and an initial value, as shown in the example below. Here we are adding an entity attribute named `height_inches`, of type `scalar_float`, with default initial value `9.0`.

The available types are 'scalar_float', 'scalar_int', and 'scalar_obj'.

```
from dymodetron import EntityType, EntityAttribute, types

# An entity is defined by creating a class that sub-classes 'EntityType'.
class Cat(EntityType):
    # You can have docstrings on any of the elements of your model.
    """Cats are a type of animal that is neither dog, duck, nor lizard."""

    # Here we declare the entity attributes of Cat.
    # The entity attribute name is on the left-hand side.
    # On the right-hand side, we specify the type of the attribute, and the default
    # initial value.
    height_inches = EntityAttribute(attribute_type=types.scalar_float, initial_value=9.0)
```

4.6 Entity Set

An entity set is a collection of entity instances that is 'calculated' at runtime, determined by applying a selection criteria to a larger set of entity instances, to choose a subset of them.

A common thing you will do in a Dymodetron model is select and gather up subsets of all the entities in your model, so that you can generate events targeting just those selected entities, or so that you can update their entity attribute values. You do this by filtering down entity sets into subsets, based on criteria that you provide. The criteria examine whether or not an individual entity instance should be in the resultant entity set or not.

This process is illustrated below. Every state entry/exit action is passed a set of entities that are entering/exiting the state. Using the `action.entity_match(...)` expression, you can filter this set down to a smaller set of filtered entities. Then, you can generate events on those filtered entities. Or, you can modify the attributes of the filtered entities.

For example, you might select all `Cat` entities that have their `height_inches` attribute larger than some threshold. Then you might generate events that target just those cats.

Entity sets are created within action statements by using `with action.entity_match(...) as name_of_entity_set: ...` expressions. In the example below, we select all cats more than 24 inches tall, and generate an event targeted at them.

```
from dymodetron import action

# Select all the cats that are more than 24 inches tall.
# The 'cat' entity set is provided to the action statement in which
# this entity_match() is used, and now we are filtering down the cats within that entity␣
↪set
# based on the value of the entity attribute 'height_inches'.
with action.entity_match(
    entity_type=Cat(),                              # What entity type?
    entity_set=cat,                                 # What entity set to start with?
```

(continues on next page)

(continued from previous page)

```
    criteria=lambda c: c.height_inches > 24.0      # Which entity instances should be␣
↪included in the resultant entity set?
) as giant_cat:                                    # The name of the resultant entity set␣
↪follows the 'as' keyword.

    # In this section, you can refer to 'giant_cat' anywhere an entity set is called for.
    # It will refer to all entities matching the criteria you provided above.

    # Generate hungry event on giant cats at 5 time ticks in the future.
    action.generate_event_rel(
        entity_type=Cat(),
        entity_set=giant_cat,
        event=hungry_event(),
        time_ticks_rel=5
    )
```

4.6.1 Naming an entity set

Notice above that the `with ... as` expression is assigning a name `giant_cat` to all the cats that match the specified criteria. The `as` clause is how the entity set is named.

That entity set name is then used within the `with ... as` block to generate an event that targets just the cats in the `giant_cat` entity set. You might instead manipulate the `giant_cat` attributes, or filter them down even further with additional `entity_match()` expressions.

4.6.2 Entity match criteria expressions

Within the `entity_match()` expression above, you'll notice a `criteria` argument. This is how individual entity instances are evaluated to determine if they belong in the entity set.

The `criteria` is an expression that takes an entity as an argument, and evaluates the entity's attributes and/or states in order to return a true/false condition indicating whether or not it should be included in the set. True = include in set, False = exclude from set.

Note: What is this `lambda` thing?

The python `lambda` syntax is a way of defining functions in-line in the code. This makes it so that we can specify the `criteria` for the entity set right there in the definition of the entity set. You can learn more about lambda expressions in the python documentation. The short version is lambdas are "syntactic sugar for a normal function definition".

The value of the `criteria` argument should be a lambda expression, or else a separate function that takes entity sets as an argument.

The example below applies the same criteria as the last example, but instead of using a `lambda`, we pull the criteria out into a standalone function named `my_criteria`. This has the benefit that you can write unit tests against the standalone criteria function, and you can re-use it in other places. The drawback is that it's not as easy to look at the `action.entity_match()` expression and know what the criteria are.

```
from dymodetron import action

def my_criteria(c):
```

(continues on next page)

(continued from previous page)

```
    result = c.height_inches > 24.0
    return result

with action.entity_match(
    entity_type=Cat(),
    entity_set=cat,
    criteria=my_criteria                          # Reference a standalone function,␣
↪instead of using a lambda.
) as giant_cat:

    # ... etc ...
```

Both approaches are equivalent, so do what works best for you. Generally speaking, if the criteria are pretty simple, and you can be pretty sure it's correct by visual inspection, then the inline `lambda` approach works well. If the criteria are complicated, or if you have the same criteria that you will use in multiple places, or if you want to gain additional confidence by writing unit tests against your criteria, then it's probably worth pulling the criteria out into a standalone function.

4.6.3 Entity match criteria expressions - a closer look

The `criteria` argument for `entity_match` has the following form, where you replace the portions indicated with angle brackets < > (there should be no angle brackets in your resulting expression).

```
criteria=lambda <entity_argument_name>: <entity-argument-boolean-expression>
```

entity_argument_name

This is any name of your choosing. You will refer to this name in the `entity-argument-boolean-expression`.

entity-argument-boolean-expression

This is a function that evaluates the entity attributes and/or states of an entity instance, to determine whether or not the entity should be included in the entity set.

Using `entity_argument_name`, you access entity attributes using 'dot' notation, and construct boolean expressions bsaed on the entity attributes. For example: `entity_argument_name.entity_attribute_name > 0` would select entity instances where the `entity_attribute_name` is larger than zero.

The types of expressions you can use are listed in this table and described in the subsequent sections.

expression type	Description
value comparisons	Use python value comparison operators.
numpy numerical expressions	Use numpy numerical expressions to perform calculations.
dymodetron action expressions	Use dymodetron action expressions to evaluate expressions involving state machines.

4.6.4 Entity match criteria expressions - examples

Value comparisons expressions using entity attributes

You can use expressions involving python value comparison operators.

Here we look for cats in the entity set `cat` having `height_inches > 24.0`.

```
from dymodetron import action

# Here suppose that entity type 'Cat' has an entity attribute 'height_inches'.
with action.entity_match(
    entity_type=Cat(),
    entity_set=cat,
    criteria=lambda c: c.height_inches > 24.0
) as giant_cat:
    # ... do something with 'giant_cat'
```

Naming the lambda expression argument

In the 'criteria', the lambda argument name can be whatever you want. It's best to make it either meaningful or a single letter. The *entity_match* block below will end up with the identical *giant_cat* entity set as the last example, we've just used a stranger name for the lambda expression argument.

```
from dymodetron import action

with action.entity_match(
    entity_type=Cat(),
    entity_set=cat,
    criteria=lambda whatever_name_you_want_to_use: whatever_name_you_want_to_use.height_
↪inches > 24.0
) as giant_cat:
    # ... do something with 'giant_cat'
```

Using a numpy expression

You can use expressions involving numpy mathematical functions in order to do calculations. Below, we are creating a set of cats that have taken an even number of naps.

```
from dymodetron import action
import numpy

# Here suppose that entity type 'Cat' has an entity attribute 'naps_taken'.
with action.entity_match(
    entity_type=Cat(),
    entity_set=cat,
    criteria=lambda c: numpy.mod(c.naps_taken) == 0
) as evenly_napped_cat:
    # ... do something with 'evenly_napped_cat'
```

Using Dymodetron 'action' expressions

Dymodetron has a number of 'action' expressions which support evaluation of entity instances in your entity set `criteria`.

action	Description	Example
in_state(entity, state_machine_name.state_name())	Determine if `entity` is currently in the referenced state. You must include the state machine name and the state name in the second argument. Also, note the closed parens `()` at the end of the state name.	# Find cats in state `eating` of state machine # `cat_sleeping_eating_drinking.action.in_state(c, cat_sleeping_eating_drinking.eating())`
not_in_state(entity, state_machine_name.state_name())	Determine if `entity` is not currently in the referenced state. You must include the state machine name and the state name in the second argument. Also, note the closed parens `()` at the end of the state name.	# Find cats not in state `eating` of state machine # `cat_sleeping_eating_drinking.action.not_in_state(c, cat_sleeping_eating_drinking.eating())`
get_state_entry_counter(entity, state_machine_name.state_name())	Determine the number of times `entity` has entered the referenced state.	# Find cats that have never eaten. `action.get_state_entry_counter(c, cat_sleeping_eating_drinking.eating()) == 0`

Matching multiple criteria

You can match on multiple criteria by using the & and | operators for logical-and and logical-or, respectively.

For example, the `entity_match` below picks out all cats that are currently in state `sleeping`, and have never been in state `eating`.

```
from dymodetron import action

with action.entity_match(
    entity_type=Cat(),
    entity_set=cat,
    criteria=lambda c:
        action.in_state(c, be_a_cat.sleeping())
        &
        action.get_state_entry_counter(c, be_a_cat.eating()) == 0
) as hungry_sleeping_cat:
    # ... do something with 'hungry_sleeping_cat'
```

4.6.5 Entity set nested filtering

By using nested `entity_match()` blocks, you can apply model logic to pick out different subsets of the entity instance population and then manipulate their entity attributes and/or generate events on them depending on which criteria they satisfy.

This process is illustrated below.

For example, below, we include two sub-filtering `entity_match` blocks nested inside the top-level `entity_match`. We generate an event on all the entity instances matching the top-level criteria. Then we filter down further into two groups, depending on additional criteria, to manipulate the entity attributes differently for different subsets of entities.

```
from dymodetron import action

# Select all the cats that are more than 24 inches tall.
with action.entity_match(
    entity_type=Cat(),
    entity_set=cat,
    criteria=lambda c: c.height_inches > 24.0
) as giant_cat:

    # All the giant cats will be seen by a predator in 10 time ticks.
    action.generate_event_rel(
        entity_type=Cat(),
        entity_set=giant_cat,
        event=observed_by_predator(),
        time_ticks_rel=10
    )

    # Here we are filtering down giant_cat, to pick out heavy giant cats.
    with action.entity_match(
        entity_type=Cat(),
        entity_set=giant_cat,
        criteria=lambda c: c.weight_lbm > 40.0
    ) as giant_heavy_cat:
        # Suppose we have a 'maximum_velocity_mph' entity attribute defined on Cats.
        # Giant heavy cats don't move very fast.
        giant_heavy_cat.maximum_velocity_mph = 1

    # Here we are filtering down giant_cat, to pick out giant cats that aren't as heavy.
    with action.entity_match(
        entity_type=Cat(),
        entity_set=giant_cat,
        criteria=lambda c: c.weight_lbm <= 40.0
    ) as giant_lighter_cat:
        # Giant lighter cats can move a little bit faster.
        giant_lighter_cat.maximum_velocity_mph = 15
```

4.7 Model Description

The model description construct contains a textual description of the model. You can put whatever you want in it.

You define the model description by creating a class that sub-classes the Dymodetron `ModelDescription` class. Use whatever name you want for the class. Add a docstring within the class to describe your model.

The name of the model description class will be taken to be the name of the model. This means that the name of python file that contains the model definition is not necessarily the name of the model. The model name is used when generating code and diagrams.

```
from dymodetron import ModelDescription

# The model description is defined by creating a class that sub-classes 'ModelDescription
↪'.
class cat_model_description(ModelDescription):
    # A model description just has a docstring explaining what you are up to with this␣
↪model.
    """We model the behaviors of a certain kind of cat."""
```

4.8 Model Parameters

The model parameters construct contains all of the scalar values, random distributions, and lookup tables that influence the behavior of the model simulation when executed.

You create the model parameters construct by defining a class that sub-classes the Dymodetron `Params` class. It is recommended that you name your model parameters class `ModelParameters`, although you can name it whatever you want.

The example model parameters below contain a random distribution, a lookup table, and a scalar.

You should put all the values that influence the behavior of your model into the ModelParameters construct.

```
from dymodetron import Params, random as dyrandom, LookupTable, LookupMethod

#############################################################################################
↪#
# Define model parameters.
#############################################################################################
↪#
class ModelParameters(Params):

    # A random distribution to be used for event generation times:
    # cat nap durations.
    nap_length_hours_distribution = dyrandom.NormalDistribution(
        mean=12.0,
        std=2.0
    )

    # Data for a lookup table:
    # cat max jumping height lookup table data.
    cat_max_jumping_height_data = dict(
        age_cutoffs       = numpy.array([0,        1,      2,          3,        4,     ␣
↪ 5,      10,     15,     20]),
```

(continues on next page)

(continued from previous page)

```
        jump_height_feet  = numpy.array([3.0,      8.0,    10.0,   12.00,    13.00,   14.
↪00,   12.00,   3.00,   1.00])
    )

    # Lookup table to be used for intermediate calculations in model:
    # cat max jumping height lookup table.
    cat_max_jump_height_lookup = LookupTable(
        breakpoints=cat_max_jumping_height_data['age_cutoffs'],
        values=cat_max_jumping_height_data['jump_height_feet'],
        lookup_method=LookupMethod.Previous
    )

    # For this model, we have all cats take the same amount of time to drink water.
    cat_drinking_time_hours = 0.0167
```

4.8.1 Scalar Model Parameter

A scalar model parameter is a value with a name.

You create a scalar model parameter by writing `scalar_param_name = value` in your model parameters block. For example, see `cat_drinking_time_hours` in the last section.

You use a scalar model parameter by referencing it in an action statement. In the example below, we have an entity set named `drinking_cats`, and we want to generate events on them at some time in the future. We use a model scalar parameter to specify the time in the future (relative to now) at which the events should be scheduled.

```
from dymodetron import action

# Generate event on 'drinking_cats' at a time in the future.
# We get the time from a model parameter.
action.generate_event_rel(entity_type=Cat(),
                          entity_set=drinking_cats,
                          event=not_thirsty_event(),
                          time_ticks_rel=ModelParameters.cat_drinking_time_hours
                          )
```

4.8.2 Lookup Table

A lookup table is a function `y = f(x)`, where given an input value `x`, the output value `y` is determined based on the lookup table data.

4.8.2.1 Defining a lookup table

You define a lookup table by providing 3 pieces of information:

- breakpoints - a monotonically increasing array of lookup input values.
- values - the lookup output values associated with each lookup input value.
- lookup_method - a rule for how to calculate the output value when an input value is in between two breakpoints.

The number of entries in `breakpoints` and `values` must be identical.

Below is an example of a lookup table definition. This illustrates the syntax for creating the breakpoint and value arrays. You must define lookup tables within your model parameters block.

```
from dymodetron import LookupTable, LookupMethod, Params

#############################################################################################
#
# Define model parameters.
#############################################################################################
#
class ModelParameters(Params):

    # Lookup table to be used for intermediate calculations in model:
    # cat max jumping height lookup table.
    cat_max_jump_height_feet_lookup_by_age_years = LookupTable(
        breakpoints   = numpy.array([0,        1,      2,          3,        4,       5,
    10,     15,     20]),
        values        = numpy.array([3,        8,     10,         12,       13,      14,
    12,      3,      1]),
        lookup_method = LookupMethod.Previous
    )
```

The available options for `lookup_method` are described below.

lookup_method	Description
LookupMethod.NearestLow	Uses the nearest breakpoint to the input value. Rounds down when interpolating half-integers.
LookupMethod.NearestHigh	Uses the nearest breakpoint to the input value. Rounds up when interpolating half-integers.
LookupMethod.Previous	Use the next lowest breakpoint.
LookupMethod.Next	Use the next largest breakpoint.

4.8.2.2 Using a lookup table

You use a lookup table within an entry action by invoking the `action.lookup` action statement.

In the example below, recall that `cat` is the entity set passed to the entry action. What the `action.lookup` statement is doing is looking up the 'max jump height' for each entity in the entity set `cat`. Then, the lookup output value is being written to the entity attribute `altitude_feet`, again one for each entity in the entity set `cat`.

```
from dymodetron import State, dymaction, action

# In this example model, we have a state that a cat gets into when it jumps.
class jump(State):
    @dymaction
    # This is the entry action for the state 'jump'.
    def entry_action(cat: Cat):
        # Remember that 'cat' is an entity set. We are looking up a value for each cat in␣
↪the entity set,
        # and assigning the entity attribute on the left-hand side of the expression,␣
↪for each cat.
        cat.altitude_feet = action.lookup(cat.age_years, ModelParameters.cat_max_jump_
↪height_feet_lookup_by_age_years)

        # Now, each entity in the entity set 'cat' has an altitude_feet value that was␣
↪determined by the lookup
        # table and the cat's 'age_years' entity attribute.
```

If you look up an input value that is outside the range of the provided breakpoints, the output value will be extrapolated per the behavior of scipy.interpolate.interp1d.

4.8.3 Random Distribution

A random distribution is used to model a random process by giving random values to quantities in your model.

4.8.3.1 Defining a random distribution

Below is an example of a random distribution being defined. You must define your random distributions within the model parameters block.

Note that when importing the `dymodetron.random` package, we alias it to `dyrandom`. You don't have to do this but it helps to clarify that we're working with Dymodetron's `random` package and not one of the many other random packages one might encounter.

```
from dymodetron import random as dyrandom

#############################################################################################
↪#
# Define model parameters.
#############################################################################################
↪#
class ModelParameters(Params):

    # A random distribution to be used for event generation times:
    # cat nap durations.
```

(continues on next page)

(continued from previous page)

```
    nap_length_hours_distribution = dyrandom.NormalDistribution(
        mean=12.0,
        std=2.0
    )
```

The random distribution types available in Dymodetron, and their parameters, are listed below.

random distribution type	Description	Parameters
UniformDistribution	Random real number uniformly distributed in the provided interval.	low, high
NormalDistribution	Random real number normally distributed with provided mean and standard deviation.	mean, std
NormalPositiveDistribution	Absolute value of NormalDistribution.	mean, std
NormalIntDistribution	NormalDistribution, rounded to produce integers.	mean, std
NormalPositiveIntDistribution	Absolute value of NormalDistribution, rounded to produce integers.	mean, std
ExponentialDistribution	Random real number Exponential distributed with provided rate parameter.	rate
PoissonDistribution	Random real number Poisson distributed with provided rate parameter.	rate
LogNormalDistribution	Random real number whose logarithm is normally distributed. The provided mean and standard deviation are of the log-normal distribution itself, not the underlying normal distribution.	mean, std
LogNormalIntDistribution	LogNormalDistribution, rounded to produce integers.	mean, std

4.8.3.2 Using a random distribution

You use random distributions by assigning the values sampled from them to *entity variables*, and then using those entity variables.

In the example below, whenever a set of cats begins to sleep, we calculate the time at which they are not tired anymore as a random process using the random distribution defined above.

We first declare an entity variable, which is a set of values, one for each entity in the entity set `cat`. Then we assign to those values from the random distribution. Then we use those values to determine the event times for new events that we generate on the entities.

```
from dymodetron import action, dymaction, State

class sleeping(State):
```

(continues on next page)

(continued from previous page)

```
    @dymaction
    def entry_action(cat: Cat):

        # Declare entity variable: one variable value for each entity in entity set 'cat'.
        t_nap_finished_hours = action.declare(entity_set=cat, var_type=numpy.float)

        # Assign entity variable from random distribution: each entity in entity set 'cat'
        # gets a sample from 'ModelParameters.nap_length_hours_distribution', stored
        # in the entity variable we declared above.
        action.assign(entity_set=cat,
                      entity_var=t_nap_finished_hours,
                      dist=ModelParameters.nap_length_hours_distribution)

        # Use random data in entity variable: each entity in entity set 'cat' gets a not_
↪tired_event(),
        # at a time determined by the random distribution we just used above.
        action.generate_event_rel(entity_type=Cat(),
                                  entity_set=cat,
                                  event=not_tired_event(),
                                  time_ticks_rel=t_nap_finished_hours.value())
```

4.9 State Machine

The state machine model of computation is *described here*.

A state machine describes behaviors of an *entity type*. The state machine consists of a number of *states*. Every state machine is associated with a single *entity type*. Every entity instance of that type has its own running 'copy' of the state machine. For a given entity instance, each state is either active or inactive at any given simulated time. When a state is active, we say that the entity instance is 'in' the state. When the state is inactive, we say that the entity instance is 'not in' the state.

States become activated and de-activated by transitions, which are triggered by events. As events occur, entity instances go in and out of various states.

After modeling entity behavior using state machines, the user can ask questions of the model in terms of counting how many times certain states are entered/exited, or how long entities stay in certain states, etc.

State machines can be hierarchical. A state 'S' in a top-level state machine can contain its own sub-state-machine. When such a state 'S' becomes active, the sub-state-machine begins execution. *Here's an example* of a hierarchical state machine.

4.9.1 Define state machine

These are the steps to defining a state machine:

1. *Declare state machine*
2. *Specify state machine entity type*
3. *Define state machine event types*
4. *Define state machine states*
5. *Define state machine transition table*

6. *(Optional) Define state machine initialization time*

These steps are described in the next sections.

4.9.1.1 Declare state machine

To declare a state machine, create a class that subclasses from Dymodetron's class `StateMachine`.

The name of the class is the name of the state machine. It is recommended that you provide a docstring for the state machine.

It is recommended that you give your state machine a meaningful name that describes the behaviors that it models. In our example below, we've provided a state machine name that is indicative of the behaviors that it captures.

Your model can have multiple state machines.

```
from dymodetron import StateMachine

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""
```

4.9.1.2 Specify state machine entity type

Every state machine is associated with a single *entity type*. You indicate which one with an expression of the form `entity_type = XYZ()` within your state machine class. Replace XYZ with the name of the entity type you want the state machine to be associated with.

```
from dymodetron import StateMachine

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()
```

4.9.1.3 Define state machine initialization time

By default every state machine initializes at simulation time zero. You can change this by specifying the initial time for the state machine.

You do this by including a variable `t_initial` within your state machine. In the example below, we specify an initial time for the state machine at `t_initial = -30.0`. Using negative initial times can be useful for building state machines that implement initialization logic for your model, to arrange for the simulation to be in a particular state at t=0.

```
from dymodetron import StateMachine

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    t_initial = -30.0
```

(continues on next page)

(continued from previous page)

```
    # Every state machine must declare the type of entity it is associated with.
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()
```

4.9.2 State

The concept of a state in a state machine is *described here*.

4.9.2.1 Define state machine states

Every state machine consists of a number of states. You populate the state machine class with states by placing a nested class inside the state machine. The name of the nested class is the name of the state. The nested state class needs to sub-class the Dymodetron `State` class.

In the example below, we have two states `sleeping` and `awake`.

```
from dymodetron import Event, State, StateMachine

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()

    class sleeping(State):
        pass

    class awake(State):
        pass
```

4.9.3 State Entry and Exit Actions

State entry and exit actions are used to generate events, modify entity attributes, or measure and report on the state of the simulation.

When an entity instance enters a state, the entry action is executed. Similarly for exit actions when a state is exited.

A state entry action is defined by placing a function named `entry_action` in the state class, as shown in the example below. You must also decorate the function with the `@dymaction` decorator.

```
from dymodetron import Event, State, StateMachine, dymaction

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()
```

(continues on next page)

(continued from previous page)

```
    class sleeping(State):
        @dymaction
        def entry_action(cat: Cat):
            time_ticks = action.get_time_ticks()
            action.log(f'cats sleeping at time = {time_ticks}')

    class awake(State):
        pass
```

State exit actions are defined similarly, but using a function named `exit_action`.

Within an entry or exit action, you use *action statements* to select sets of entities and then generate events on them and/or modify their entity attributes.

The argument to `entry_action` is an entity set. This is because events, which are what cause a transition resulting in a state being entered, are targeted at an entity set. The `entry_action` argument is the set of all entity instances entering a state together at some simulation time. It may contain one, many, or all of the entity instances in the simulation.

The syntax for the `entry_action` argument is as follows (similar for `exit_action`):

```
def entry_action(<ENTITY_SET_NAME>: <ENTITY_TYPE>):
```

You can use whatever name you want for `ENTITY_SET_NAME`. Then reference that name in the body of the `entry_action`. The `ENTITY_TYPE` should be the same type that you set with the state machine's `entity_type` declaration.

Including the `: <ENTITY_TYPE>` is optional but recommended. You could instead do the following:

```
def entry_action(<ENTITY_SET_NAME>):
```

4.9.4 Event Type

Events are *described here*. In the model definition you define types of events. Using action statements you define the model logic that results in generation of events when the simulation runs. Each individual instance of an event is a realization of the event type.

4.9.4.1 Define state machine event types

State machines react to events. Part of the state machine model is a set of events that it will react to.

You define an event by creating a class that sub-classes from the Dymodetron `Event` class.

In the example below, we're defining two event types `tired_event` and `not_tired_event`.

```
from dymodetron import Event, State, StateMachine

# An event type is defined by creating a class that sub-classes Event.
class tired_event(Event):
    pass

class not_tired_event(Event):
    pass
```

(continues on next page)

(continued from previous page)

```
class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()

    class sleeping(State):
        pass

    class awake(State):
        pass
```

4.9.5 State Transition

State transitions are *described here*.

A state transition is defined by the following:

- event_type - the type of event that causes the transition.
- source_state - if an entity is in this state when the event arrives, the transition will occur.
- target_state - the state that an entity will go into after the transition.

Transitions are defined within the state transition table, see next section.

When a transition occurs, an entity leaves the source state and enters the target state. If the target state has a sub-state-machine, then that sub-state-machine will initialize and start responding to events.

4.9.6 State Transition Table

Every state machine has one state transition table. The state transition table is a list of state transitions.

4.9.6.1 Define state machine transition table

A transition table is defined within the state machine class by declaring a variable `transitions` and then using the Dymodetron `Transitions` and `Transition` constructs as illustrated in the example below.

```
from dymodetron import Event, State, StateMachine, Transition, Transitions

# An event type is defined by creating a class that sub-classes Event.
class tired_event(Event):
    pass

class not_tired_event(Event):
    pass

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
```

(continues on next page)

(continued from previous page)

```
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()

    class sleeping(State):
        pass

    class awake(State):
        pass

    # Here, we define the transition table for state-machine 'cat_sleeping_eating_drinking
↪'.
    #
    # The state machine initializes into the 'sleeping' state. Then, events toggle it␣
↪back and forth
    # between 'sleeping' and 'awake'.
    transitions = Transitions([
        Transition(event_type=initial_event(),   source_state=initial_state(), target_
↪state=sleeping()),
        Transition(           not_tired_event(),              sleeping(),              ␣
↪   awake()),
        Transition(           tired_event(),                  awake(),                 ␣
↪   sleeping())
    ])
```

Initial state, initial event, initial transition

Every state machine has an initial state. You don't define this state explicitly in your model, it is implied by the existence of the state machine. The initial state is represented by a black dot in a state machine diagram.

When a state machine initializes, it is in the initial state. At the time of state machine initialization, a special event fires, of type `initial_event`.

In your transition table, you include an entry that describes what state the state machine should go into the `initial_event` fires. This is called the initial transition. The initial transition should have `event_type=initial_event()` and `source_state=initial_state()`.

See the transition table in the example above for an illustration of how to specify the initial transition.

4.9.7 State Tracking

When you simulate a model, you'll often want to count how many entities enter a particular state over the course of the simulation, or examine how the number of entities in a state changes over time.

You can configure the model on a per-state basis to enable tracking of the metrics listed in the table below. When the simulation completes, it will write out a state tracking report describing the state tracking metrics for the enabled states.

Table 2: State Tracking Options

State Tracking Option	Description
`track_n`	Track the number of entities in a state at each event time.
`track_new`	Track the number of entities newly entering a state at each event time.
`track_cumulative`	Track the total number of entities that entered a state over the course of a simulation.

You configure state tracking on a state by using the `@dymodetrack` decorator. You pass it the arguments in the table listed above in order to enable to respective state tracking feature.

In the example below, we have indicated that we want to enable `track_cumulative` and `track_n` on state `awake`.

```
from dymodetron import State, StateMachine, dymodetrack

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()

    class sleeping(State):
        pass

    @dymodetrack(track_cumulative=True, track_n=True)
    class awake(State):
        pass
```

4.9.8 Actions

Action statements are used to implement model logic in state entry and exit actions.

One key characteristic of Dymodetron action statements is that they operate on *entity sets*. For example, when you generate events, you generate events on all the entity instances an entity set. When you declare a variable, the variable contains a separate value for each entity instance in an entity set. When you assign to that variable, you can assign a different value for each entity instance. When you calculate expressions using that variable, the calculation is performed for all the entity instances, using the appropriate value for each one.

The table below lists the available action commands.

Table 3: Dymodetron action statement commands

Action statement name	Action statement command	Description
Entity Match	`action.entity_match()`	Filter an entity set down to a subset of entities based on evaluation criteria.
Declare Entity Variable	`action.declare()`	Declare a variable that will hold a separate value for each entity instance in an entity set.
Assign to Entity Variable	`action.assign()`	Assign a value to an entity variable; can be a different value for each entity instance in the associated entity set.
Generate Events	`action.generate_event_...()`	Generate events targeted to entity instances in an entity set.
In State or Not?	`action.in_state()`, `.not_in_state()`	Evaluate whether entity instances in an entity set are in a particular state or not.
Last Time State Was Entered	`action.get_last_state_entry_time_ticks()`	Determine the last time a particular state was entered by entity instances in an entity set.
State Entry Counter	`action.get_state_entry_counter()`	Determine the number of times a state has been entered by any entity instance.
Call	`action.call()`	Call a function on all the entity instances in an entity set.
Measurements	`action.measurements()`	Evaluate metrics such as count, average, etc., on entity instances in an entity set matching provided criteria.
Lookup in Lookup Table	`action.lookup()`	Look up values in a lookup table, one value per entity instance in an entity set.
Get Current Simulation Time	`action.get_time_ticks()`	Retrieve the current simulation time.
Number Crunching (numpy)	`numpy.*`	Evaluate numerical computations involving entity attribute and entity variables. Computations are performed once for each entity instance in an entity set.

4.9.8.1 Entity Match

The `entity_match` action is described in the section on *entity sets*.

You use this action in a `with .. as ..` construct, as shown below.

```
from dymodetron import action

with action.entity_match(
    entity_type=<ENTITY TYPE>,                      # What entity type?
    entity_set=<INPUT ENTITY SET>,                  # What entity set to start with?
    criteria=lambda E: <EXPRESSION INVOLVING E>     # Which entity instances should be␣
↪included in the resultant entity set?
) as <RESULT ENTITY SET>:                           # The name of the resultant entity set␣
↪follows the 'as' keyword.

    # Use <RESULT ENTITY SET> here.
```

The `action.entity_match()` arguments are described below.

Table 4: `entity_match` arguments

Argument name	Description
`entity_type`	The type of entities in the entity set.
`entity_set`	The name of the entity set containing entity instances to evaluate.
`criteria`	Expression evaluating to True/False that determines which entity instances to include in the resultant entity set.

When using `lambda` expressions for `criteria`, the `lambda` argument name can be whatever you choose (it doesn't have to be `E`).

In the example below, we have an `entity_match` that picks all cats whose height exceeds a given threshold.

```
from dymodetron import State, StateMachine, action

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()

    class sleeping(State):
        @dymaction
        def entry_action(cat: Cat):
            # The entry action takes an entity set as an argument.
            # In this case, we have named the entity set 'cat', and each entity instance
            # has entity type 'Cat'.

            # Select all the cats that are more than 24 inches tall.
            # The 'cat' entity set is provided to the action statement in which
            # this entity_match() is used, and now we are filtering down the cats within␣
↪that entity set
            # based on the value of the entity attribute 'height_inches'.
            with action.entity_match(
                entity_type=Cat(),                              # What entity type?
                entity_set=cat,                                 # What entity set to start␣
↪with?
                criteria=lambda c: c.height_inches > 24.0       # Which entity instances␣
↪should be included in the resultant entity set?
            ) as giant_cat:                                     # The name of the␣
↪resultant entity set follows the 'as' keyword.

                # In this section, you can refer to 'giant_cat' anywhere an entity set is␣
↪called for.
                # It will refer to all entities matching the criteria you provided above.

                # Generate hungry event on giant cats at 5 time ticks in the future.
                action.generate_event_rel(
                    entity_type=Cat(),
                    entity_set=giant_cat,
                    event=hungry_event(),
                    time_ticks_rel=5
                )
```

Note: *What is the argument that gets passed to the criteria function?*

The argument that gets passed to the `criteria` function is an entity set. It is actually the same entity set that you pass to the `entity_set` argument in your `entity_match()` invocation.

However, in the `criteria` expression, you treat this argument as an individual object. For example, as shown above, we access an entity attributes using 'dot' notation: `criteria=lambda c: c.height_inches > 24.0`.

What is happening here is that Dymodetron sets things up under the hood so that you can write the code as if you are examining a single entity instance's attributes. Behind the scenes, Dymodetron will apply the criteria evaluation to all of the entities in the provided `entity_set`.

Similarly, when you use the `numpy` and Dymodetron `action` statements described in the last section, Dymodetron will apply these across the entire set of instances in the input `entity_set`.

When you are writing the criteria, you write it as if you are examining a single entity instance. The Dymodetron action syntax is set up to make this the case, so that the criteria expressions can be easily read and written without worrying about the complexity of managing and iterating through collections of entity instances. All that is done behind the scenes.

The main thing in keep in mind is that it's really an entire set of entities that are being operated on when these `criteria` expressions are evaluated, even though the criteria expressions are written and read as if a single entity is being evaluated.

4.9.8.2 Entity Variables

In action statements, it is often necessary to perform intermediate calculations in the process of generating events or modifying entity attributes. For example, we may need to perform unit conversions or other arithmetic on entity attributes before using them.

Entity variables are variables that store a value for every entity instance in an entity set. Entity variables are always declared and used in the context of some entity set.

When the value of an entity variable is calculated, it is calculated for each entity in the entity set.

Entity variables are similar to entity attributes: they have a name, a type, and a separate value for each entity instance. The main difference between entity variables and entity attributes is that entity variables are transient and only exist in the scope of the action statement where they are declared. Entity attributes, on the other hand, are attached to the entity instances and continue to 'exist' and maintain state irrespective of any state or state machine (although their values are typically read and written by state entry/exit actions).

A more subtle distinction between entity variables and entity attributes is that every entity instance in a simulation has a value for every defined entity attribute. On the other hand, entity variables are defined in terms of an entity set which may or may not include every entity instance in the simulation. The relationship between entity variables and entity sets is described further in the sections below.

There are three actions you can perform with entity variables: declare entity variable, assign to entity variable, and access entity variable values. These are listed in the table below with links to additional information.

Table 5: Entity variable actions

Action	Description
Declare Entity Variable	Declare an entity variable (name and data type) in association with a given entity set.
Assign to Entity Variable	Assign values to an entity variable, separate values associated with each entity instance in the entity set.
Access Entity Variable Value	Retrieve the values of an entity variable to be used in Dymodetron expressions.

In the example below, we modify the example from the last section, and convert the cat height from inches to feet before using it. We implement a crude conversion that rounds down the the nearest integer number of feet. This conversion calculation is calculated separately for each entity instance in `entity_set=cat`.

We then modify the `entity_match` to use the converted value in determining which cats to include in the `giant_cat` entity set. This is done using the expression `.value(...)` tacked onto the entity variable name, within the `criteria` expression.

The expressions `action.declare()`, `action.assign()`, and `.value()` are described further in the following sections.

```
from dymodetron import State, StateMachine, action

class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    entity_type = Cat()

    class sleeping(State):
        @dymaction
        def entry_action(cat: Cat):

            # For each entity instance in entity_set=cat, we'll have a value in variable␣
↪height_feet,
            # initialized to 0.0 for each entity instance.
            height_feet = action.declare(entity_set=cat, var_type=numpy.float, default_
↪value=0.0)

            # For each entity instance in entity_set=cat, we'll calculate a value in␣
↪variable height_feet.
            # We convert to feet, and roud the result down to the next lowest integer.
            action.assign(entity_set=cat, entity_var=height_feet, expr=lambda c: numpy.
↪floor(c.height_feet / 12.0))

            # Select all the cats that are more than 2 feet tall, per the conversion␣
↪above.
            # The 'cat' entity set is provided to the action statement in which
            # this entity_match() is used, and now we are filtering down the cats within␣
↪that entity set
            # based on the value of the entity variable 'height_feet'.
            with action.entity_match(
                entity_type=Cat(),                              # What entity type?
                entity_set=cat,                                 # What entity set to start␣
↪with?
                criteria=lambda c: height_feet.value(c) > 2.0 # Which entity instances␣
↪should be included in the resultant entity set?
            ) as giant_cat:                                     # The name of the␣
↪resultant entity set follows the 'as' keyword.

                ...
```

Declare Entity Variable

An entity variable is declared with an expression of the following form:

```
from dymodetron import action

<ENTITY_VARIABLE_NAME> = action.declare(entity_set, var_type, default_value)
```

The result is an entity variable. The entity variable is a set of values, one for each entity instance in the provided entity set. Each value will be equal to the provided `default_value`, until the entity variable is assigned to using `action.assign()`.

The `action.declare()` arguments are described below.

Table 6: `declare` arguments

Argument name	Description
`entity_set`	The entity set for which we are creating an entity variable that will store a separate value for each entity instance.
`var_type`	The type of the variable. The supported types are the numpy types.
`default_value`	The default value to be used for each entity instance, if an assignment doesn't set it.

Assign to Entity Variable

Typically your model will assign values to entity variables. When assigning values to an entity variable you specify the subset (one, many, or all) of entity instances that go along with the values to assign. Values can be assigned by providing a constant, an expression, or a random distribution. These options are described further below.

Table 7: Entity variable assignment options

Assignment type	Description
Value	Provide a literal constant value to be used for all the entity instances.
Expression	Provide an expression that will be calculated independently for each entity instance.
Random Distribution	Provide a random distribution that will be sampled once for each entity instance.

Assign to Entity Variable - Value

You can assign a fixed scalar value to an entity variable. This results in each entity instance in the entity variable's entity set having the same value for the variable.

An entity variable is assigned in 'value' mode with an expression of the following form:

```
from dymodetron import action

action.assign(entity_set, entity_variable, value=<VALUE>)
```

Table 8: Arguments for entity variable assignment from value

Argument name	Description
`entity_set`	The name of the entity set for which entity instances should be assigned variable values. This must be a subset of the `entity_set` originally provided to `action.declare()` when creating the entity variable.
`entity_variable`	The entity variable that was returned from the call to `action.declare()`.
`<VALUE>`	A scalar value. This value become the entity variable associated with each entity instance in `entity_set`.

In the example below, we declare an entity variable `heart_rate_bpm`. This entity variable has a separate value for each entity instance in the entity set `cat`. The values are floating point numbers initialized to `80`.

Then, we use *action.entity_match()* to select the subset of all cats in state `sleeping`, and assign a different value for `heart_rate_bpm` for those cats.

```
from dymodetron import action

# Declare an entity variable.
#
# The name of the entity variable is on the left-hand side of the equal sign.
#
# The entity set 'cat' comes from the surrounding context.
#
# The variable type is floating point number.
#
# The entity variable will contain a separate value for each entity instance in entity␣
↪set 'cat'.
#
# To start with, the value of 'heart_rate_bpm' for all the entity instances will be 80.
heart_rate_bpm = action.declare(entity_set=cat, var_type=numpy.float, default_value=80)

# Select all cats that are in state 'sleeping'.
with action.entity_match(
        entity_type=Cat(),
        entity_set=cat,
        criteria=lambda c: action.in_state(c, minimal_cat_sleeping_eating_drinking.
↪sleeping())
) as sleeping_cat:

    # Assign value 40 to 'heart_rate_bpm' for all sleeping cats.
    action.assign(entity_set=sleeping_cat, entity_var=heart_rate_bpm, value=40)
```

Assign to Entity Variable - Expression

You can assign an entity variable based on an expression. The expression is evaluated separately for each entity instance in the entity variable's entity set. Usually you'll use entity attributes in the expression. Each individual entity instance's value for the entity attribute is used, resulting in potentially different resulting entity variable values for each entity instance.

An entity variable is assigned in 'expression' mode as follows:

```
from dymodetron import action

action.assign(entity_set, entity_variable, expr=lambda E: <EXPRESSION INVOLVING E>)
```

Table 9: Arguments for entity variable assignment from expression

Argument name	Description
`entity_set`	The name of the entity set for which entity instances should be assigned variable values. This must be a subset of the `entity_set` originally provided to `action.declare()` when creating the entity variable.
`entity_variable`	The entity variable that was returned from the call to `action.declare()`.
`<EXPRESSION>`	An expression involving entity attributes.

Note: What is this `lambda` thing again?

As discussed in the section on *entity match criteria*, the python `lambda` syntax is a way of defining functions inline in the code. For `action.assign()`, this makes it so that we can specify the expression for calculating entity variable values right there in the assignment expression. You can learn more about lambda expressions in the python documentation. The short version is lambdas are "syntactic sugar for a normal function definition".

In your `lambda` expression, the `lambda` argument name can be whatever you choose (it doesn't have to be `E`).

Similar to the `criteria` argument for `action.entity_match`, the `<EXPRESSION INVOLVING E>` can access entity attributes.

In the example below, assume that cats have an entity attribute `weight_lbm`. We assign cat heart rate using a calculation based on cat weight. For each entity instance in entity set `cat`, the associated value for `heart_rate_bpm` will be calculated using that entity instance's individual entity attribute value for `weight_lbm`.

```
from dymodetron import action

# Declare an entity variable.
#
# The name of the entity variable is on the left-hand side of the equal sign.
#
# The entity set 'cat' comes from the surrounding context.
#
# The variable type is floating point number.
#
# The entity variable will contain a separate value for each entity instance in entity␣
↪set 'cat'.
#
# To start with, the value of 'heart_rate_bpm' for all the entity instances will be 80.
heart_rate_bpm = action.declare(entity_set=cat, var_type=numpy.float, default_value=80)

# Assign heart rate based on weight: 40 bpm plus 1/2 bpm per pound.
action.assign(
    entity_set=cat,
    entity_var=heart_rate_bpm,
    expr=lambda c: 40.0 + (c.weight_lbm * 0.5)
)
```

Assign to Entity Variable - Distribution

You can assign to an entity variable from a random distribution. The random distribution is sampled separately for each entity instance in the entity variable's entity set.

Recall that random distributions are defined in your model parameters block.

An entity variable is assigned in 'random distribution' mode with an expression of the following form:

```
from dymodetron import action

action.assign(entity_set, entity_variable, dist=ModelParameters.<DISTRIBUTION_NAME>)
```

Table 10: Arguments for entity variable assignment from random distribution

Argument name	Description
`entity_set`	The name of the entity set for which entity instances should be assigned variable values. This must be a subset of the `entity_set` originally provided to `action.declare()` when creating the entity variable.
`entity_variable`	The entity variable that was returned from the call to `action.declare()`.
`ModelParameters`	The name of the class you've used to define your *model parameters*. If you used a different name for your model parameters class, use that name instead.
`<DISTRIBUTION_NAME>`	The name of a random distribution that you have defined in your model parameters block.

In the example below, we define a random distribution for heart rate in our model parameters block. Then, we use the distribution in the assignment to the entity variable. Each entity instance in the entity set `cat` gets a separately sampled value from the distribution `heart_rate_bpm_dist` defined in `ModelParameters`.

```
from dymodetron import Params, random as dyrandom, action

###############################################################################################
↪#
# Define model parameters.
###############################################################################################
↪#
class ModelParameters(Params):

    # A random distribution to be used for cat heart rates:
    # normal distribution with mean = 60, standard deviation = 10.
    heart_rate_bpm_dist = dyrandom.NormalDistribution(
        mean=60.0,
        std=10.0
    )

# ...
# ... later in model, in an action statement ...
# ...

# Declare an entity variable.
#
# The name of the entity variable is on the left-hand side of the equal sign.
#
```

(continues on next page)

(continued from previous page)

```
# The entity set 'cat' comes from the surrounding context.
#
# The variable type is floating point number.
#
# The entity variable will contain a separate value for each entity instance in entity␣
↪set 'cat'.
#
# To start with, the value of 'heart_rate_bpm' for all the entity instances will be 80.
heart_rate_bpm = action.declare(entity_set=cat, var_type=numpy.float, default_value=80)

# Assign heart rate based on random distribution from model parameters.
# Each entity instance in entity set 'cat' gets a sampled value from the referenced
# distribution.
action.assign(
    entity_set=cat,
    entity_var=heart_rate_bpm,
    dist=ModelParameters.heart_rate_bpm_dist
)
```

Access Entity Variable Value

After going to all the work of *declaring entity variables* and *assigning to entity variables*, you usually want to use the values for something.

Recall that entity variables are associated with an entity set, and that they store a separate value for each entity instance in the entity set. When you use an entity variable, you use it in the context of an expression involving the entity variable's entity set, or a subset of it.

Entity variable values are accessed as follows:

```
from dymodetron import action

# Access entity variable values in the context of some expression.
... <ENTITY_VARIABLE_NAME>.value(<ENTITY_SET>) ...
```

Table 11: Arguments for accessing entity variable values

Argument name	Description
<ENTITY_VARIABLE_NAME>	The name you gave the entity variable *when you declared it*.
<ENTITY_SET>	An entity set containing the entity instances for which you wish to access the variable values. Must be a subset of the entity set you provided *when you declared the entity variable*.

In the example below, we update the last example by picking out cats where the value of entity variable `heart_rate_bpm` is less than some threshold, and then generate an event just on those cats. In this example, we are looking at entity variable values for all the entities in the entity set that we used when we declared the entity variable.

```
from dymodetron import Params, random as dyrandom, action

################################################################################################
↪#
# Define model parameters.
```

(continues on next page)

(continued from previous page)

```
##########################################################################################
↪#
class ModelParameters(Params):

    # A random distribution to be used for cat heart rates:
    # normal distribution with mean = 60, standard deviation = 10.
    heart_rate_bpm_dist = dyrandom.NormalDistribution(
        mean=60.0,
        std=10.0
    )

# ...
# ... later in model, in an action statement ...
# ...

# Declare an entity variable.
#
# The name of the entity variable is on the left-hand side of the equal sign.
#
# The entity set 'cat' comes from the surrounding context.
#
# The variable type is floating point number.
#
# The entity variable will contain a separate value for each entity instance in entity␣
↪set 'cat'.
#
# To start with, the value of 'heart_rate_bpm' for all the entity instances will be 80.
heart_rate_bpm = action.declare(entity_set=cat, var_type=numpy.float, default_value=80)

# Assign heart rate based on random distribution from model parameters.
# Each entity instance in entity set 'cat' gets a sampled value from the referenced
# distribution.
action.assign(
    entity_set=cat,
    entity_var=heart_rate_bpm,
    dist=ModelParameters.heart_rate_bpm_dist
)

# Pick out the cats with a low heart rate.
with action.entity_match(
        entity_type=Cat(),
        entity_set=cat,
        criteria=lambda c: heart_rate_bpm_dist.value(c) < 30
) as cat_with_low_heart_rate:

    # Generate event on cats with low heart rate.
    cat_with_low_heart_rate.generate_event_rel(
        entity_type=Cat(),
        entity_set=cat_with_low_heart_rate,
        event=tired_event(),
        time_ticks_rel=0,
    )
```

Sometimes you will access entity variable values on just a subset of entities. In the example below, we assign heart_rate_bpm on all the cats, but we access the value only for the sleeping subset of cats.

```
from dymodetron import Params, random as dyrandom, action

##############################################################################################
#
# Define model parameters.
##############################################################################################
#
class ModelParameters(Params):

    # A random distribution to be used for cat heart rates:
    # normal distribution with mean = 60, standard deviation = 10.
    heart_rate_bpm_dist = dyrandom.NormalDistribution(
        mean=60.0,
        std=10.0
    )

# ...
# ... later in model, in an action statement ...
# ...

# Declare an entity variable.
#
# The name of the entity variable is on the left-hand side of the equal sign.
#
# The entity set 'cat' comes from the surrounding context.
#
# The variable type is floating point number.
#
# The entity variable will contain a separate value for each entity instance in entity set 'cat'.
#
# To start with, the value of 'heart_rate_bpm' for all the entity instances will be 80.
heart_rate_bpm = action.declare(entity_set=cat, var_type=numpy.float, default_value=80)

# Assign heart rate based on random distribution from model parameters.
# Each entity instance in entity set 'cat' gets a sampled value from the referenced
# distribution.
action.assign(
    entity_set=cat,
    entity_var=heart_rate_bpm,
    dist=ModelParameters.heart_rate_bpm_dist
)

# Select all cats that are in state 'sleeping'.
with action.entity_match(
        entity_type=Cat(),
        entity_set=cat,
        criteria=lambda c: action.in_state(c, minimal_cat_sleeping_eating_drinking.sleeping())
) as sleeping_cat:
```

(continues on next page)

(continued from previous page)

```
    # Pick out the sleeping cats with a high heart rate.
    with action.entity_match(
            entity_type=Cat(),
            entity_set=sleeping_cat,
            criteria=lambda c: sleeping_cat.value(c) >= 30
    ) as sleeping_cat_with_high_heart_rate:

        # Generate event on sleeping cats with high heart rate.
        sleeping_cat_with_high_heart_rate.generate_event_rel(
            entity_type=Cat(),
            entity_set=sleeping_cat_with_high_heart_rate,
            event=not_tired_event(),
            time_ticks_rel=0,
        )
```

4.9.8.3 Generate Events

Events are *described here*. The action statements for generating events are listed below.

Table 12: Event generating action statements.

Action	Action statement command	Description
Generate event	`generate_event_rel` `generate_event_abs`	Generate events on entities at a given simulation time.
Generate event w/ binomial probability	`generate_event_binomial_rel`, `generate_event_binomial_abs`	Generate events, or not, on entities at a given simulation time, according to a binomial probability distribution with given parameters.
Generate event w/ given probability	`generate_event_trial_rel`, `generate_event_trial_abs`	Generate events, or not, on entities with a given probability for each entity.

Generate event

Events are generated at an absolute simulation time with expressions of the following form:

```
from dymodetron import action

# ... within entry or exit action ...

# Generate events at absolute sim time.
action.generate_event_abs(
    entity_type=<ENTITY_TYPE>(),
    entity_set=<ENTITY_SET>,
    event=<EVENT_TYPE>(),
    time_ticks_abs=<SIMULATION_TIME_TICKS>
)

# Generate events at sim time, relative to sim time at which this action is executed.
action.generate_event_rel(
```

(continues on next page)

(continued from previous page)

```
    entity_type=<ENTITY_TYPE>(),
    entity_set=<ENTITY_SET>,
    event=<EVENT_TYPE>(),
    time_ticks_rel=<SIMULATION_TIME_TICKS>
)
```

Note the parentheses `()` following the entity type and the event type.

Table 13: Arguments for `generate_event_...`

Argument name	Description
`entity_type`	The type of entity you are generating events on.
`entity_set`	The set of entity instances (one, many, or all) that should receive the event.
`event`	The type of event to generate. This should reference an event type *that you defined elsewhere in the model definition*.
`time_ticks_abs`	Applicable only to `generate_event_abs()`. The absolute simulation time at which to generate the event, measured in *ticks*. Can also be an array of times, one for each entity instance in `entity_set`.
`time_ticks_rel`	Applicable only to `generate_event_rel()`. The relative simulation time at which to generate the event. Relative to the simulation time at which the action was executed. Can also be an array of times, one for each entity instance in `entity_set`.

Generate event w/ binomial probability

Sometimes you may want to generate an event probabilistically on a set of entities, where a random process determines whether or not each entity gets an event or not. With this action statement, for each entity instance in `entity_set`, a random draw from a binomial distribution will be taken. For a given entity instance, if the "success" outcome is achieved, then `event_a` will be triggered for that entity instance. If the "failure" outcome is achieved, then `event_b` will be triggered instead. You can also set `event_b=None`, in which case "failure" outcomes result in no event being scheduled for the associated entity instance.

```
from dymodetron import action

# ... within entry or exit action ...

# Generate events probabilistically at absolute simulation time.
action.generate_event_binomial_abs(
    entity_type=<ENTITY_TYPE>(),
    entity_set=<ENTITY_SET>,
    event_a=<EVENT_TYPE>(),
    event_b=<EVENT_TYPE>(),
    p_a=<PROBABILITY>,
    time_ticks_abs=<SIMULATION_TIME_TICKS>
)


# Generate events probabilistically at sim time, relative to sim time at which this␣
↪action is executed.
action.generate_event_binomial_rel(
    entity_type=<ENTITY_TYPE>(),
    entity_set=<ENTITY_SET>,
```

(continues on next page)

(continued from previous page)

```
    event_a=<EVENT_TYPE>(),
    event_b=<EVENT_TYPE>(),
    p_a=<PROBABILITY>,
    time_ticks_rel=<SIMULATION_TIME_TICKS>
)
```

Table 14: Arguments for `generate_event_binomial_...`

Argument name	Description
`entity_type`	The type of entity you are generating events on.
`entity_set`	The set of entity instances (one, many, or all) that should receive events.
`event_a`	The type of event to generate for "success" outcome of `n` trials with probability `p_a`. This should reference an event type *that you defined elsewhere in the model definition*.
`event_b`	The type of event to generate for "failure" outcome of `n` trials with probability `p_a`. Can also be `None`, in which case no event is generated in the "failure" outcome case.
`p_a`	The binomial distribution 'probability' parameter.
`n`	The binomial distribution 'number of trials' parameter. (NOT CURRENTLY IMPLEMENTED, hard-coded to n=1, sorry!).
`time_ticks_abs`	Applicable only to `generate_event_binomial_abs`. The simulation time at which to generate the events, measured in *ticks*. Can also be an array of times, one for each entity instance in `entity_set`.
`time_ticks_rel`	Applicable only to `generate_event_binomial_rel`. The simulation time at which to generate the events, measured in *ticks*. Can also be an array of times, one for each entity instance in `entity_set`.

Generate event w/ given probability

This action is similar to the last section, but instead of drawing from the same distribution for each entity instance, you explicitly provide the probabilities yourself for each entity instance as an array argument.

```
from dymodetron import action

# ... within entry or exit action ...

# Generate events probabilistically at absolute simulation time.
action.generate_event_trials_abs(
    entity_type=<ENTITY_TYPE>(),
    entity_set=<ENTITY_SET>,
    event=<EVENT_TYPE>(),
    p=<PROBABILITY>,
    time_ticks_abs=<SIMULATION_TIME_TICKS>
)


# Generate events probabilistically at sim time, relative to sim time at which this␣
↪action is executed.
action.generate_event_trials_rel(
    entity_type=<ENTITY_TYPE>(),
    entity_set=<ENTITY_SET>,
    event=<EVENT_TYPE>(),
    p=<PROBABILITY>,
```

(continues on next page)

(continued from previous page)

```
    time_ticks_rel=<SIMULATION_TIME_TICKS>
)
```

Table 15: Arguments for `generate_event_trials_...`

Argument name	Description
`entity_type`	The type of entity you are generating events on.
`entity_set`	The set of entity instances (one, many, or all) that should receive events.
`event`	The type of event to generate with probability `p`. This should reference an event type *that you defined elsewhere in the model definition*.
`p`	Array of probabilities of event being generated, one entry for each individual entity instance.
`time_ticks_abs`	Applicable only to `generate_event_trials_abs`. The simulation time at which to generate the events, measured in *ticks*. Can also be an array of times, one for each entity instance in `entity_set`.
`time_ticks_rel`	Applicable only to `generate_event_trials_rel`. The simulation time at which to generate the events, measured in *ticks*. Can also be an array of times, one for each entity instance in `entity_set`.

4.9.8.4 In State or Not?

These actions are used in the `criteria` clause of the `entity_match` expression. They let you you determine whether entity instances are in a specified state.

```
from dymodetron import action

# ... within entry or exit action ...

with action.entity_match(
        entity_type=<ENTITY_TYPE>,
        entity_set=<ENTITY_SET>,
        criteria=lambda e: action.in_state(e, <STATE_NAME>)
) as entities_in_state:

    # Do something with entities_in_state ...
```

You can also check if an entity instance is not in a specified state:

```
from dymodetron import action

# ... within entry or exit action ...

with action.entity_match(
        entity_type=<ENTITY_TYPE>,
        entity_set=<ENTITY_SET>,
        criteria=lambda e: action.not_in_state(e, <STATE_NAME>())
) as entities_not_in_state:

    # Do something with entities_not_in_state ...
```

4.9.8.5 State Entry Counter

This action provides the number of times that that a specified state has been entered, for each entity instance in a specified entity set.

```
from dymodetron import action

with action.entity_match(
    entity_type=<ENTITY_TYPE>(),
    entity_set=<ENTITY_SET>,
    criteria=lambda e: action.get_state_entry_counter(e, <STATE_NAME>()) > 0
) as entities_that_have_entered_state:

    # Do something with entities_that_have_entered_state ...
```

4.9.8.6 Last Time State Was Entered

This action provides the last time that that a specified state was entered, for each entity instance in a specified entity set.

```
from dymodetron import action

with action.entity_match(
    entity_type=<ENTITY_TYPE>(),
    entity_set=<ENTITY_SET>,
    criteria=lambda e: action.get_state_entry_counter(e, <STATE_NAME>()) > 0
) as entities_that_have_entered_state:

    # Get the current time (a scalar).
    t_now = action.get_time_ticks()

    # Get the last time STATE_NAME was entered by the entities in question.
    _, t_last_state_entry_time = action.get_last_state_entry_time_ticks(entities_that_
↪have_entered_state, <STATE_NAME>())

    # Declare entity variable to store result of calculation for each entity instance.
    time_since_last_state_entry = action.declare(entity_set=entities_that_have_entered_
↪state, var_type=numpy.float)

    # Calculate the simulation time that has passed since these entities were last in␣
↪the state.
    time_since_last_state_entry.assign(
        entity_set=entities_that_have_entered_state,
        entity_var=time_since_last_state_entry,
        # This expression relies on the numpy behavior of 'broadcasting' the scalar 't_now'
        # so that we take the different between t_now and the value of each entry in t_
↪last_state_entry_time.
        expr=lambda e: numpy.maximum(1, t_now - t_last_state_entry_time)
    )
```

4.9.8.7 Call

It may be useful sometimes to carve a set of action statements out into its own function. Once you've done that, you can call the function from an action statement using the `call` action. This is done with expressions of the following form:

```
from dymodetron import action

# ... within entry or exit action ...

action.call(<ENTITY_SET>, <FUNCTION>)
```

Table 16: Arguments for `call()`

Argument name	Description
`ENTITY_SET`	The entity set to pass to the function.
`FUNCTION`	The function to call.

```
from dymodetron import action, dymaction

# ... within a state machine definition ...

# In this example model, we have a state that a cat gets into when it jumps.
class jump(State):
    @dymaction
    def entry_action(cat: Cat):
        action.call(cat, update_cat_energy)

# A function consisting of action statements on an entity set.
@dymaction
def update_cat_energy(cat: Cat):
    # Calculate the energy of each entity instance in entity set 'cat',
    # and save the value to entity attributes.
    cat.energy = 0.5 * cat.mass_lbm * numpy.power(cat.velocity, 2)
```

A function that will be called by the `call` statement must take a single argument: an entity set that it will operate on. The function also needs the `@dymaction` decorator applied to it, similar to state entry and exit actions.

4.9.8.8 Measurements

These actions are for quantifying aggregate metrics across entity instances at the simulated time that the action statements are executed.

You create measurements with action statements as illustrated in the example below.

```
from dymodetron import action

# ... within entry or exit action ...

with action.measurements(
    entity_type=Cat(),
    entity_set=cat
) as measuring_cats:
```

(continues on next page)

(continued from previous page)

```
    # Count the number of cats under a given weight.
    # The measurement will be attached to the 'measuring_cats' object.
    measuring_cats.count(
        # The name of the measurement.
        'Number of light cats',
        # The criteria that must be satisfied to be counted.
        criteria=lambda c: c.weight_lbm < 40.0
    )

    # Measure the mean cat height.
    measuring_cats.mean(
        # The name of the measurement.
        'Mean cat height (inches)',
        # Take the mean of the weight.
        value=lambda c: c.height_inches
        # No criteria provided, means use all the cats in the entity set provided
        # in the original expression above (entity_set=cat).
    )

    # Measure the mean cat height, but only the light cats.
    measuring_cats.mean(
        # The name of the measurement.
        'Mean height (inches) of light cats',
        # Take the mean of the weight.
        value=lambda c: c.height_inches
        # Only the entity instances that match this criteria will be included in taking␣
↪the mean.
        criteria=lambda c: c.weight_lbm < 40.0
    )

    # Apply a numpy function. In this case we're getting the median height of light cats.
    measuring_cats.apply(
        # The name of the measurement.
        'Median height (inches) of light cats',
        # Take the mean of the weight.
        value=lambda c: c.height_inches,
        f=numpy.median
        # The criteria that must be satisfied to be included in taking the median.
        criteria=lambda c: c.weight_lbm < 40.0
    )

    # Count and store the number of heavy cats.
    heavy_cat_count = measuring_cats.count(
        # The name of the measurement.
        'Number of heavy cats',
        # The criteria that must be satisfied to be counted.
        criteria=lambda c: c.weight_lbm >= 40.0
    )

    # Count and store the total number of cats.
    total_cat_count = measuring_cats.count(
```

(continues on next page)

(continued from previous page)

```
        # The name of the measurement.
        'Number of cats',
        # The criteria that must be satisfied to be counted.
        # criteria=None means count them all.
        criteria=None
    )

    # This is the same as the last one, but a different way to do it.
    # Count and store the total number of cats.
    total_cat_count = measuring_cats.count(
        # The name of the measurement.
        'Number of cats',
        # Not providing any criteria means count them all.
    )

    # Store the proportion of heavy cats as a measurement.
    measuring_cats.attach(
        'Proportion of heavy cats',
        heavy_cat_count / total_cat_count
    )

    # Now log the measurements.
    action.log(measuring_cats)
```

The last statement above writes the measurements to the log. For this example, the output might be as follows:

```
{'Number of light cats': 2946,
 'Mean cat height (inches)': 13.2,
 'Mean height (inches) of light cats': 11.9,
 'Median height (inches) of light cats': 8.4,
 'Number of heavy cats': 2054,
 'Number of cats': 5000,
 'Proportion of heavy cats': 0.4108,
}
```

Table 17: Measurement functions

Measurement function name	Description
count	Count the number of entity instances matching some provided criteria.
mean	Take the mean of an expression involving entity attributes, optionally subject to some criteria.
nanmean	Take the mean of an expression involving entity attributes, optionally subject to some criteria, ignoring nans.
apply	Use a provided function to calculate an aggregate measurement.
attach	Attach your own computed value to the measurements object.

Creating measurement object

You create the measurement object with expressions of the following form.

```
from dymodetron import action

# ... within entry or exit action ...

with action.measurements(
    entity_type=<ENTITY_TYPE>>,
    entity_set=<ENTITY_SET>
) as <MEASUREMENT_OBJECT>:

    # ... use <MEASUREMENT_OBJECT> ...
```

Table 18: Arguments for `measurements()`

Argument name	Description
`entity_type`	The type of entity to measure.
`entity_set`	The entity set containing the entity instances to measure.

See example.

Count

```
from dymodetron import action

# ... within entry or exit action ...

with action.measurements(
    entity_type=<ENTITY_TYPE>>,
    entity_set=<ENTITY_SET>
) as <MEASUREMENT_OBJECT>:

    # Count the number of entity instances in <ENTITY_SET> that match
    # the criteria.
    count_value = <MEASUREMENT_OBJECT>.count(
        label=<STRING>,
        criteria=lambda E: <EXPRESSION_INVOLVING_E>
    )
```

Table 19: Arguments for `count()`

Argument name	Description
`label`	The string to use to label the measurement.
`criteria`	A boolean expression on an entity indicating whether or not the entity should be counted.

See example.

Capturing the return value `count_value` is optional.

Mean

```
from dymodetron import action

# ... within entry or exit action ...

with action.measurements(
    entity_type=<ENTITY_TYPE>>,
    entity_set=<ENTITY_SET>
) as <MEASUREMENT_OBJECT>:

    # Calculate the mean of 'value' across all entity instances in <ENTITY_SET> that match
    # the criteria.
    mean_value = <MEASUREMENT_OBJECT>.mean(
        label=<STRING>,
        value=lambda E: <EXPRESSION_INVOLVING_E>,
        criteria=lambda E: <EXPRESSION_INVOLVING_E>
    )
```

Table 20: Arguments for `mean()`

Argument name	Description
`label`	The string to use to label the measurement.
`value`	An expression returning the value to use for each entity instance. The measurement calculates the mean of this value over all entity instances that satisfy `criteria`.
`criteria`	A boolean expression on an entity indicating whether or not the entity should be included in the calculation.

See example.

Capturing the return value `mean_value` is optional.

Nanmean

Same as *mean*, but ignores `nan` values in the calculation.

```
from dymodetron import action

# ... within entry or exit action ...

with action.measurements(
    entity_type=<ENTITY_TYPE>>,
    entity_set=<ENTITY_SET>
) as <MEASUREMENT_OBJECT>:

    # Calculate the mean of 'value' across all entity instances in <ENTITY_SET> that match
    # the criteria.  Ignore an entity instance when 'value' is nan.
    mean_value = <MEASUREMENT_OBJECT>.nanmean(
        label=<STRING>,
        value=lambda E: <EXPRESSION_INVOLVING_E>,
        criteria=lambda E: <EXPRESSION_INVOLVING_E>
    )
```

See example.

Apply

Apply a function of your choice to calculate the aggregate. The function should take an array-like of numbers and return a scalar value.

```
from dymodetron import action

# ... within entry or exit action ...

with action.measurements(
    entity_type=<ENTITY_TYPE>>,
    entity_set=<ENTITY_SET>
) as <MEASUREMENT_OBJECT>:

    # Apply 'f' to the array of 'value' across all entity instances matching 'criteria'.
    measurement_value = <MEASUREMENT_OBJECT>.apply(
        label=<STRING>,
        value=lambda E: <EXPRESSION_INVOLVING_E>,
        f=<FUNCTION>,
        criteria=lambda E: <EXPRESSION_INVOLVING_E>
    )
```

Table 21: Arguments for `apply()`

Argument name	Description
`label`	The string to use to label the measurement.
`value`	An expression returning the value to use for each entity instance. The measurement calculates the mean of this value over all entity instances that satisfy `criteria`.
`f`	A function taking an array-like of numbers, and returning a scalar. The resulting measurement is the calculation of this function being passed an array-like containing `value` for all entity instances that satisfy `criteria`.
`criteria`	A boolean expression on an entity indicating whether or not the entity should be included in the calculation.

See example.

Capturing the return value `measurement_value` is optional.

Attach

Sometimes you may want to perform some intermediate calculations and attach them to the measurement object. For example, you may calculate a ratio of *counts*, and add that as a labeled measurement.

```
from dymodetron import action

# ... within entry or exit action ...

with action.measurements(
    entity_type=<ENTITY_TYPE>>,
    entity_set=<ENTITY_SET>
```

(continues on next page)

(continued from previous page)

```
) as <MEASUREMENT_OBJECT>:

    # ... calculate a scalar value, usually out of other measurements ...
    v = 1 + 2

    # Attach 'v'
    <MEASUREMENT_OBJECT>.attach(
        label=<STRING>,
        value=v
    )
```

Table 22: Arguments for `apply()`

Argument name	Description
`label`	The string to use to label the measurement.
`value`	A scalar value to store as a measurement.

See example.

4.9.8.9 Lookup in Lookup Table

If you've *defined a lookup table* in your model, you can look up values in it using action statements of the following form. This assigns a looked-up 'output' entity attribute value for each entity instance in the given entity set, given the input `lookup_values`.

```
from dymodetron import action

# ... within entry or exit action ...

# Assign entity attribute from lookup table.
<ENTITY_SET>.<OUTPUT_ENTITY_ATTRIBUTE> = action.lookup(
    lookup_values=<ENTITY_SET>.<LOOKUP_ENTITY_ATTRIBUTE>,
    lookup_table=<LOOKUP_TABLE>
)
```

Here's an example.

Alternatively, you can assign the lookup output to an entity variable, with the following.

```
from dymodetron import action

# Assign entity variable from lookup table.
my_variable = action.declare(entity_set=<ENTITY_SET>, var_type=<VAR_TYPE>)

action.assign(
    entity_set=<ENTITY_SET>,
    entity_var=my_variable,
    values=action.lookup(<ENTITY_SET>.<LOOKUP_ENTITY_ATTRIBUTE>, <LOOKUP_TABLE>)
)
```

You can also use an entity variable as the input to the lookup, as follows.

```
from dymodetron import action

# Assign entity variable from lookup table.
my_variable = action.declare(entity_set=<ENTITY_SET>, var_type=<VAR_TYPE>)

# Suppose model parameters has a random distribution defined named 'my_random_distribution
↪'.
# We'll assign to my_variable by sampling from the random distribution.
action.assign(
    entity_set=<ENTITY_SET>,
    entity_var=my_variable,
    dist=ModelParameters.my_random_distribution
)

# Now assign to an entity attribute using looked-up values with 'my_variable' as the␣
↪input.
<ENTITY_SET>.<OUTPUT_ENTITY_ATTRIBUTE> = action.lookup(my_variable.value(), <LOOKUP_
↪TABLE>)
```

4.9.8.10 Get Current Simulation Time

You can retrieve the current simulation time with the following expression.

```
from dymodetron import action

# Retrieve current simulation time.
t_now_ticks = action.get_time_ticks()

# ... use 't_now_ticks' in expression ...
```

Here's an example.

4.9.8.11 Number Crunching (numpy)

Often your model will need to calculate numerical expressions involving entity attributes and entity variables. You can do this using numpy mathematical functions.

Some examples are shown below.

```
from dymodetron import action, State, dymaction

# In this example model, we have a state that a cat gets into when it jumps.
class jump(State):
    @dymaction
    # This is the entry action for the state 'jump'.
    def entry_action(cat: Cat):

        # Put a floor on cat altitude.
        capped_altitude_feet = numpy.max(1, cat.altitude_feet)

        # Calculate the air pressure for all the cats.
        air_pressure_pascals = 101325 * numpy.power(1 - 2.25577e-5 * cat.altitude_feet,␣
↪5.25588)
```

(continues on next page)

(continued from previous page)

```
        # Calculate the cats' kinetic energy.
        cat_energy = 0.5 * cat.mass_kg * numpy.power(cat.velocity_m_s, 2)
```

4.9.9 Full minimal state machine example

Below is a full minimal example of a model with a state machine. This example illustrates the use of states, events, and transitions. This example does not include state entry/exit actions, action statements, model parameters, or sub-state-machines.

```
from dymodetron import \
    EntityType, \
    StateMachine, \
    ModelDescription, \
    Params, \
    State, \
    Event, \
    Transition, \
    Transitions, \
    initial_state, \
    initial_event

class minimal_cat(ModelDescription):
    # A model description just has a docstring explaining what you are up to with this␣
↪model.
    """We model the behaviors of a certain kind of cat."""


class Cat(EntityType):
    pass

class ModelParameters(Params):
    pass


# An event type is defined by creating a class that sub-classes Event.
class tired_event(Event):
    pass


class not_tired_event(Event):
    pass


class minimal_cat_sleeping_eating_drinking(StateMachine):
    """Defines the sleeping patterns of cats, as far as this model is concerned."""

    # Every state machine must declare the type of entity it is associated with.
    # The entity type is declared elsewhere in the model, and we are referencing it here.
    entity_type = Cat()
```

(continues on next page)

(continued from previous page)

```
    class sleeping(State):
        pass

    class awake(State):
        pass

    # Here, we define the transition table for state-machine 'minimal_cat_sleeping_eating_
↪drinking'.
    #
    # The state machine initializes into the 'sleeping' state. Then, events toggle it␣
↪back and forth
    # between 'sleeping' and 'awake'.
    transitions = Transitions([
        Transition(event_type=initial_event(),   source_state=initial_state(), target_
↪state=sleeping()),
        Transition(           not_tired_event(),              sleeping(),              ␣
↪   awake()),
        Transition(           tired_event(),                  awake(),                 ␣
↪   sleeping())
    ])
```

Below is the generated diagram for the minimal state machine example.

The diagram is generated by running the following command from the root of the Dymodetron folder:

```
# python -m dymodetron.generators.state_machine_diagrams --model_definition_file=examples/
minimal_cat.py --overwrite_existing=1
```

CHAPTER

FIVE

GENERATORS

5.1 Generate Code

The Dymodetron code generator creates code to simulate a model. The code generator takes a model definition file as input, and creates an output file containing the simulation code.

The code generator is a python module that can be invoked from the command line, as follows. Use `--help` to list the command line arguments.

```
# python generated/python_arrayified/cat_model.py --help
usage: cat_model.py [-h] [--num_entities NUM_ENTITIES] [--enable_logging ENABLE_LOGGING]␣
↪[--rand_seed RAND_SEED] [--end_time_ticks END_TIME_TICKS] [--progress_interval_ticks␣
↪PROGRESS_INTERVAL_TICKS]

optional arguments:
  -h, --help            show this help message and exit
  --num_entities NUM_ENTITIES
                        Number of entity instances.
  --enable_logging ENABLE_LOGGING
                        Enable logging, or not.
  --rand_seed RAND_SEED
                        Random number seed.
  --end_time_ticks END_TIME_TICKS
                        Simulation end time (ticks).
  --progress_interval_ticks PROGRESS_INTERVAL_TICKS
                        Interval between reporting sim progress (ticks).
```

For example, you can generate the code for the example cat model by running the following from the root Dymodetron directory.

```
# python -m dymodetron.generators.python_arrayified --model_definition_file=examples/cat.
↪py --overwrite_existing=1
python_arrayified
Generating code for entity type [Cat]
Outputting to [generated/python_arrayified/cat_model.py]
```

5.1.1 Run Code

After you've generated the simulation code, you can run it as follows.

```
# python generated/python_arrayified/cat_model.py --enable_logging=1 --end_time_
↪ticks=100 --progress_interval_ticks=10
```

Which will result in output something like the following. First it reads back the arguments, some of which have default values because they weren't specified on the command line.

Then, time progress log messages are displayed. The formatting has to do with the logger configuration.

Finally, the `State Tracking` report is displayed. In this case, our model only has state tracking enabled for the 'eating' state, so that is the only one that shows any results: `track_cumulative=400`. This says that the state 'eating' was entered 400 times by a cat. Some of these are repeat occurrences of the same cat entering the 'eating' state multiple times. (The state 'eating' exists in sub-state-machine 'awake', in state machine 'cat_sleeping_eating_drinking'.)

The `track_cumulative=None` indicators tell you about states where state tracking is not enabled. *Read about enabling state tracking here*.

```
# python generated/python_arrayified/cat_model.py --enable_logging=1 --end_time_
↪ticks=100 --progress_interval_ticks=10
Sim Arguments:
{
  'enable_logging': 1,
  'end_time_ticks': 100.0,
  'num_entities': 100,
  'progress_interval_ticks': 10.0,
  'rand_seed': 1,
}
INFO:__main__:t = 0.0
INFO:__main__:t = 10.0
INFO:__main__:t = 20.0
INFO:__main__:t = 35.0
INFO:__main__:t = 45.0
INFO:__main__:t = 60.0
INFO:__main__:t = 70.0
INFO:__main__:t = 85.0
INFO:__main__:t = 95.0
-----------------------------------
State Tracking
-----------------------------------
instance(Entities_Cat_state_tracking):
  cat_sleeping_eating_drinking__awake: StateTracking(track_cumulative=None),
  cat_sleeping_eating_drinking__awake__drinking: StateTracking(track_cumulative=None),
  cat_sleeping_eating_drinking__awake__eating: StateTracking(track_cumulative=400),
  cat_sleeping_eating_drinking__awake__staring_out_window: StateTracking(track_
↪cumulative=None),
  cat_sleeping_eating_drinking__sleeping: StateTracking(track_cumulative=None)
```

Note: The `progress_interval_ticks` argument throttles the printing of the message `t = xyz` indicating simulation progress.

You'll notice that the printed event times don't come on even multiples of 10. This is because for this simulation, there aren't necessarily events occurring at every `t = n * 10`. The `progress_interval_ticks` sets an upper limit on

how many messages you'll see every n ticks, but doesn't make them show up on regular intervals.

You can get display additional information about event scheduling and processing by setting environment variable LOGLEVEL=DEBUG:

```
# LOGLEVEL=DEBUG python generated/python_arrayified/cat_model.py --enable_logging=1 --
↪end_time_ticks=30 --progress_interval_ticks=10
```

Which provides the additional output shown below:

```
# LOGLEVEL=DEBUG python generated/python_arrayified/cat_model.py --enable_logging=1 --
↪end_time_ticks=30 --progress_interval_ticks=10
Sim Arguments:
{
  'enable_logging': 1,
  'end_time_ticks': 30.0,
  'num_entities': 100,
  'progress_interval_ticks': 10.0,
  'rand_seed': 1,
}
DEBUG:__main__:Scheduling event: [ScheduledEvent(time_ticks=0.0, event_id=0, event_obj=<_
↪_main__.initial_event_types.initial_event_cat_sleeping_eating_drinking object at␣
↪0x7fb4370b4e20>, entity_ids=None)]
INFO:__main__:t = 0.0
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=0.0, event_id=0, event_obj=<_
↪_main__.initial_event_types.initial_event_cat_sleeping_eating_drinking object at␣
↪0x7fb4370b4e20>, entity_ids=None)]
DEBUG:__main__:Scheduling event: [ScheduledEvent(time_ticks=10.0, event_id=1, event_obj=
↪<examples.cat.not_tired_event object at 0x7fb4370c5a30>, entity_ids=array([ 0,  1,  2,␣
↪ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
INFO:__main__:t = 10.0
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=10.0, event_id=1, event_obj=
↪<examples.cat.not_tired_event object at 0x7fb4370c5a30>, entity_ids=array([ 0,  1,  2,␣
↪ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
DEBUG:__main__:Scheduling event: [ScheduledEvent(time_ticks=10.0, event_id=2, event_obj=
↪<__main__.initial_event_types.initial_event_cat_sleeping_eating_drinking__awake object␣
↪at 0x7fb4370b4970>, entity_ids=array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,␣
↪12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
```

(continues on next page)

(continued from previous page)

```
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=10.0, event_id=2, event_obj=
↪<__main__.initial_event_types.initial_event_cat_sleeping_eating_drinking__awake object␣
↪at 0x7fb4370b4970>, entity_ids=array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,␣
↪12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
DEBUG:__main__:Event was scheduled with no target entity_ids: [<examples.cat.hungry_
↪event object at 0x7fb4370c5b20> @ 15.0]
DEBUG:__main__:Scheduling event: [ScheduledEvent(time_ticks=17.0, event_id=3, event_obj=
↪<examples.cat.thirsty_event object at 0x7fb4370c5be0>, entity_ids=array([ 0,  1,  2, ␣
↪3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=17.0, event_id=3, event_obj=
↪<examples.cat.thirsty_event object at 0x7fb4370c5be0>, entity_ids=array([ 0,  1,  2, ␣
↪3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
DEBUG:__main__:Scheduling event: [ScheduledEvent(time_ticks=20.0, event_id=4, event_obj=
↪<examples.cat.not_thirsty_event object at 0x7fb4370b4e50>, entity_ids=array([ 0,  1, ␣
↪2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
INFO:__main__:t = 20.0
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=20.0, event_id=4, event_obj=
↪<examples.cat.not_thirsty_event object at 0x7fb4370b4e50>, entity_ids=array([ 0,  1, ␣
↪2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
DEBUG:__main__:Scheduling event: [ScheduledEvent(time_ticks=25.0, event_id=5, event_obj=
↪<examples.cat.hungry_event object at 0x7fb4370c5a90>, entity_ids=array([ 0,  1,  2,  3,
↪  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
```

(continues on next page)

(continued from previous page)

```
DEBUG:__main__:Event was scheduled with no target entity_ids: [<examples.cat.thirsty_
↪event object at 0x7fb4370c5d30> @ 27.0]
DEBUG:__main__:Processing event: [ScheduledEvent(time_ticks=25.0, event_id=5, event_obj=
↪<examples.cat.hungry_event object at 0x7fb4370c5a90>, entity_ids=array([ 0,  1,  2,  3,
↪  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
DEBUG:__main__:Scheduling event: [ScheduledEvent(time_ticks=35.0, event_id=6, event_obj=
↪<examples.cat.not_hungry_event object at 0x7fb4370b4e50>, entity_ids=array([ 0,  1,  2,
↪  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]))]
-----------------------------------
State Tracking
-----------------------------------
instance(Entities_Cat_state_tracking):
  cat_sleeping_eating_drinking__awake: StateTracking(track_cumulative=None),
  cat_sleeping_eating_drinking__awake__drinking: StateTracking(track_cumulative=None),
  cat_sleeping_eating_drinking__awake__eating: StateTracking(track_cumulative=100),
  cat_sleeping_eating_drinking__awake__staring_out_window: StateTracking(track_
↪cumulative=None),
  cat_sleeping_eating_drinking__sleeping: StateTracking(track_cumulative=None)
```

5.2 Generate Diagrams

The Dymodetron diagram generator creates state machine diagrams for each state machine in your model.

The diagram generator is a python module that can be invoked from the command line, as follows. Use `--help` to list the command line arguments.

```
# python -m dymodetron.generators.state_machine_diagrams --help
usage: state_machine_diagrams.py [-h] --model_definition_file MODEL_DEFINITION_FILE [--
↪output_folder OUTPUT_FOLDER] [--overwrite_existing OVERWRITE_EXISTING]

optional arguments:
  -h, --help            show this help message and exit
  --model_definition_file MODEL_DEFINITION_FILE
                        Location of model definition file.
  --output_folder OUTPUT_FOLDER
                        Output folder in which to place the generated diagrams.
  --overwrite_existing OVERWRITE_EXISTING
                        Whether or not to overwrite existing diagrams in the output␣
↪folder, if they already exist.
```

For example, you can generate the diagrams for the example cat model by running the following from the root Dymod-

etron directory.

```
# python -m dymodetron.generators.state_machine_diagrams --model_definition_
↪file=examples/cat.py --overwrite_existing=1
state_machine_diagrams
Generating state machine diagram for state machine [cat_sleeping_eating_drinking]
Outputting to [generated/state_machine_diagrams/cat_sleeping_eating_drinking.html]
```

Each state machine in your model will output to its own diagram file. This model has one state machine model, so there's one diagram file.

The state chart diagram is an html file. Load it into your web browser to view it. It should look like the following.

CHAPTER

SIX

EXAMPLE

CHAPTER

SEVEN

TUTORIAL

TODO

CHAPTER

EIGHT

DYMODETRON HOW-TO GUIDE

This section how to do various things that are done with Dymodetron.

8.1 Generate simulation code for a model

8.2 Generate diagrams for a model

8.3 Build models

This topic has its own *separate guide*.

8.4 Run a simulation of a model

8.5 Examine simulation results

8.6 Change model parameters

8.7 Change model random seed

8.8 Debug a model simulation

CHAPTER

NINE

DYMODETRON HOW-TO GUIDE: MODEL BUILDING

CHAPTER

TEN

WHY DYMODETRON?

10.1 What are the problems that Dymodetron is trying to solve?

- Reduce the time, effort, and programming expertise required to rapidly build, analyze, and iterate on simulations of useful dynamic models.
  - The goal is to make it easy and fast to ask questions of a model, get answers, and then ask new questions.
- Separate model definition/construction from the code required to simulate execution of the model, so that the model definition can be studied, reviewed, and manipulated independently of the complexity of the code that simulates the model.
  - This is accomplished with tools that generate code automatically from the model definition.
- Provide transparency to every step of the computations used in simulating dynamic models.
  - The generated code is visible, boring, and not mysterious. So simple, a robot could write it!
- Enable clear visualizations of the model that can be brought up to date with a single command execution or button press.
  - This is done by providing tools to automatically generate diagrams from the model definition.
- Through the use of a set of inter-connectable modeling constructs, enable the sharing of models and model component libraries.
- Allow anyone to change Dymodetron as they see fit: modify and add modeling constructs, change the code and diagram generators, etc.
  - Dymodetron is open source with a flexible license that lets you do whatever you want with it.

CHAPTER

ELEVEN

WHAT'S THE NAME ALL ABOUT?

11.1 Die-moohda-what ?

pronounced: dy-MOE-duh-tron, or dy-MAHWD-uh-tron

11.2 What does "tron" mean?

From Ancient Greek - (-tron), a suffix denoting an instrument…

(Per https://en.wiktionary.org/wiki/-tron on 4/27/2021)

11.3 What does "dymodetron" mean?

DY NAMIC **MODE** L **TRON** - an instrument for building dynamic models.

CHAPTER

TWELVE

VERIFICATION STATUS

12.1

Dymodetron has no tests. Dymodetron has had very little in the way of code review.

This is all very “alpha”. Use at your own risk!

CHAPTER

THIRTEEN

ASPIRATIONS

There are a number of things Dymodetron wants to do, but doesn't. In no particular order:

13.1 Easy ensembles of simulation runs

Constructs and features for configuring, executing, and analyzing ensembles of many runs of a simulation, sampling the random distributions on each run, in order to easily analyze the distributions of the simulation results. For now, this is left as an exercise for the user. This would be used to support the following use cases:

1. Random distribution sampling (running multiple "seeds" of a simulation with the same set of fixed parameters).
2. Sensitivity analysis using parameter sweeps and monte-carlo.
3. Model parameter calibration.
4. History matching.
5. Generate simulation data for follow-on model reduction processes.

13.2 Model Probes

TODO

13.3 Multiple Entity Types

Support multiple entity types in a single model, each having their own distinct sets of state machines.

13.4 State Machine Orthogonal Regions

Support Harel statechart orthogonal regions.

13.5 State Machine History States

Support Harel statechart history states.

13.6 Support Additional Models of Computation

Models are built to answer questions. Different kinds of questions require different kinds of models. Different kinds of models require different underlying *"models of computation"*. There are a number of other models of computation that Dymodetron wants to support directly. These would need to be done in such a way that any given model could utilize multiple of them simultaneously. This requires the different notions of time and causality to be integrated together. In some cases you can kinda-sorta approximate these additional models of computation by implementing them inside a state machine, but that's not the same thing and there will be limitations, negative performance implications, or both, on top of the extra effort required on the part of the modeler, and the fact that any such workaround will be a point solution, probably only useful for a single model, and hard for others to leverage. It would be better for the additional models of computation to be supported directly. Some key missing models of computation are listed below.

13.6.1 Ordinary, Stochastic, and Partial Differential Equations (ODE/SDE/PDE)

System states modeled using differential equations. Rate equation parameters can be modified by e.g. state machine events. Threshold crossings by e.g. ODE-calculated state values can trigger state machine events.

13.6.2 Stochastic Simulation Algorithm (aka Gillespie, SSA)

Stochastic simulation of sets of finite populations that vary in time. Reaction rates (propensity functions) are used to describe how the populations change in time. Similar to ODE model, but having quantized state values and rate equations capable of capturing e.g. local depletion/exhaustion; more valid when a continuum approximation is not appropriate.

13.6.3 Flow Chart

Describe a sequence of activities, often with conditional logic. Different from a state machine, in that a state machine is "reactive" while a flow chart is more like "imperative". In a state machine, the "boxes" are states that an entity can be in, and the arrows are transitions triggered by events that move entities between states. In a flow chart, the "boxes" are steps to take, and the arrows are sequencing of those steps and conditional logic that can take the flow down a different path. Flow charts are good for modeling e.g. a decision making process that involves a set of rules to be evaluated.

13.6.4 Data flow / process model

Similar to that supported by xarray-simlab. Imperative graphs of data processing.

13.6.5 Components and Interconnects.

Enable "model blocks" built out of the other Dymodetron constructs to be linked together and built up hierarchically into larger model blocks.

13.6.6 Additional model description elements.

Add more model description elements, such as author, references, links, etc.

13.6.7 Additional options for lookup table behavior.

Add options for out-of-range lookup values to either rail to a constant value or raise errors, instead of extrapolating.

Add 'Exact' lookup method that raises error if the input lookup value isn't exactly a breakpoint.