desmod Documentation

Pete Grayson

Jan 13, 2021

Contents

Contents:

CHAPTER 1

API Reference

This API reference details desmod's various modules, classes and functions.

1.1 desmod

Full-featured, high-level modeling using SimPy.

The *desmod* package provides a variety of tools for composing, configuring, running, monitoring, and analyzing discrete event simulation (DES) models. It builds on top of the `simpy` simulation kernel, providing features useful for building large-scale models which are out-of-scope for `simpy` itself.

An understanding of SimPy is required to use desmod effectively.

1.1.1 Components

The primary building-block for *desmod* models is the *Component* class. Components provide a means for partitioning the system to be modeled into manageable pieces. Components can play a structural role by parenting other components; or play a behavioral role by having processes and connections to other components; or sometimes play both roles at once.

The *desmod.dot.component_to_dot()* function may be used to create a DOT language representation of the component hierarchy and/or the component connection graph. The resulting DOT representation may be rendered to a variety of graphical formats using GraphViz tools.

1.1.2 Configuration

It is common for models to have configurable paramaters. Desmod provides an opinionated mechanism for simulation configuration. A single, comprehensive configuration dictionary captures all configuration for the simulation. The configuration dictionary is propogated to all Components via the *SimEnvironment*.

The various components (or component hierarchies) may maintain separate configuration namespaces within the configuration dictionary by use of keys conforming to the dot-separated naming convention. For example, "my-model.compA.cfgitem".

The *desmod.config* module provides various functionality useful for managing configuration dictionaries.

1.1.3 Simulation

Desmod takes care of the details of running simulations to allow focus on the act of modeling.

Running a simulation is accomplished with either *simulate()* or *simulate_factors()*, depending whether running a single simulation or a multi-factor set of simulations. In either case, the key ingredients are the configuration dict and the model's top-level *Component*. The *simulate()* function takes responsibility for taking the simulation through its various phases:

- *Initialization*: where the components' *__init__()* methods are called.
- *Elaboration*: where inter-component connections are made and components' processes are started.
- *Simulation*: where discrete event simulation occurs.
- *Post-simulation*: where simulation results are gathered.

TODO: simulation results

1.1.4 Monitoring

TODO: tracers, probes, logging, etc.

1.2 desmod.config

Tools for managing simulation configurations.

Each simulation requires a configuration dictionary that defines various configuration values for both the simulation (*desmod*) and the user model. The configuration dictionary is flat, but the keys use a dotted notation, similar to `Component` scopes, that allows for different namespaces to exist within the [flat] configuration dictionary.

Several configuration key/values are required by desmod itself. These configuration keys are prefixed with 'sim.'; for example: 'sim.duration' and 'sim.seed'.

Models may define their own configuration key/values, but should avoid using the 'sim.' prefix.

The *NamedManager* class provides a mechanism for defining named groupings of configuration values. These named configuration groups allow quick configuration of multiple values. Configuration groups are also composable: a configuration group can be defined to depend on several other configuration groups.

Most functions in this module are provided to support building user interfaces for configuring a model.

class desmod.config.**ConfigError**
Exception raised for a variety of configuration errors.

class desmod.config.**NamedConfig**
Named configuration group details.

Iterating a *NamedManager* instance yields `NamedConfig` instances.

category
Alias for field number 0

name
Alias for field number 1

doc
Alias for field number 2

depend
Alias for field number 3

config
Alias for field number 4

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

class desmod.config.**NamedManager**
Manage named configuration groups.

Any number of named configuration groups can be specified using the *name()* method. The *resolve()* method is used to compose a fully-resolved configuration based on one or more configuration group names.

Iterating a NamedManager instance will yield *NamedConfig* instances for each registered named configuration.

name(*name: str*, *depend: Optional[List[str]] = None*, *config: Optional[Dict[str, Any]] = None*, *category: str = ''*, *doc: str = ''*) → None
Declare a new configuration group.

A configuration group consists of a name, a list of dependencies, and a dictionary of configuration key/values. This function declares a new configuration group that may be later resolved with *resolve()*.

Parameters

- **name** (*str*) – Name of new configuration group.
- **depend** (*list*) – List of configuration group dependencies.
- **config** (*dict*) – Configuration key/values.
- **category** (*str*) – Optional category.
- **doc** (*str*) – Optional documentation for named configuration group.

resolve(**names*) → Dict[str, Any]
Resolve named configs into a new config object.

desmod.config.**apply_user_overrides**(*config: Dict[str, Any]*, *overrides: Iterable[Tuple[str, str]]*, *eval_locals: Optional[Dict[str, Any]] = None*) → None
Apply user-provided overrides to a configuration.

The user-provided *overrides* list are first verified for validity and then applied to the the provided *config* dictionary.

Each user-provided key must already exist in *config*. The *fuzzy_lookup()* function is used to verify that the user-provided key exists unambiguously in *config*.

The user-provided value expressions are evaluated against a safe local environment using eval(). The type of the resulting value must be type-compatible with the existing (default) value in *config*.

Parameters

- **config** (*dict*) – Configuration dictionary to modify.
- **overrides** (*list*) – List of user-provided (key, value expression) tuples.
- **eval_locals** (*dict*) – Optional dictionary of locals to use with `eval()`. A safe and useful set of locals is provided by default.

desmod.config.**parse_user_factors**(*config: Dict[str, Any], user_factors, eval_locals: Optional[Dict[str, Any]] = None*) → List[Tuple[List[str], List[Any]]]

Safely parse user-provided configuration factors.

A configuration factor consists of an n-tuple of configuration keys along with a list of corresponding n-tuples of values. Configuration factors are used by *simulate_factors()* to run multiple simulations to explore a subset of the model's configuration space.

Parameters

- **config** (*dict*) – The configuration dictionary is used to check the keys and values of the user-provided factors. The dictionary is not modified.
- **user_factors** – Sequence of *(user_keys, user_expressions)* tuples. See *parse_user_factor()* for more detail on user keys and expressions.
- **eval_locals** (*dict*) – Optional dictionary of locals used when `eval()`-ing user expressions.

Returns List of keys, values pairs. The returned list of factors is suitable for passing to `simulate_factors()`.

Raises ***ConfigError*** – For invalid user keys or expressions.

desmod.config.**parse_user_factor**(*config: Dict[str, Any], user_keys: str, user_exprs: str, eval_locals: Optional[Dict[str, Any]] = None*) → Tuple[List[str], List[Any]]

Safely parse a user-provided configuration factor.

Example:

```
>>> config = {'a.b.x': 0,
              'a.b.y': True,
              'a.b.z': 'something'}
>>> parse_user_factor(config, 'x,y', '(1,True), (2,False), (3,True)')
[['a.b.x', 'a.b.y'], [[1, True], [2, False], [3, True]]]
```

Parameters

- **config** (*dict*) – The configuration dictionary is used to check the keys and values of the user-provided factors. The dictionary is not modified.
- **user_keys** (*str*) – String of comma-separated configuration keys of the factor. The keys may be fuzzy (i.e. valid for use with *fuzzy_lookup()*), but note that the returned keys will always be fully-qualified (non-fuzzy).
- **user_exprs** (*str*) – User-provided Python expressions string. The expressions string is evaluated using `eval()` with, by default, a safe locals dictionary. The expressions string must evaluate to a sequence of n-tuples where *n* is the number of keys provided in *user_keys*. Further, the elements of each n-tuple must be type-compatible with the existing (default) values in the *config* dict.
- **eval_locals** (*dict*) – Optional dictionary of locals used when `eval()`-ing user expressions.

Returns

A config factor: a pair (2-list) of keys and values lists.

Note: All sequences in the returned factor are expressed as lists, not tuples. This is done to improve YAML serialization.

Raises ***ConfigError*** – For invalid keys or value expressions.

desmod.config.**factorial_config**(*base_config: Dict[str, Any], factors: Iterable[Tuple[List[str], List[Any]]], special_key: Optional[str] = None*) → Iterator[Dict[str, Any]]

Generate configurations from base config and config factors.

Parameters

- **base_config** (*dict*) – Configuration dictionary that the generated configuration dictionaries are based on. This dict is not modified; generated config dicts are created with `copy.deepcopy()`.
- **factors** (*list*) – Sequence of one or more configuration factors. Each configuration factor is a 2-tuple of keys and values lists.
- **special_key** (*str*) – When specified, a key/value will be inserted into the generated configuration dicts that identifies the "special" (unique) key/value combinations of the specified *factors* used in the config dict.

Yields Configuration dictionaries with the cartesian product of the provided *factors* applied. I.e. each yielded config dict will have a unique combination of the *factors*.

desmod.config.**fuzzy_match**(*keys: Iterable[str], fuzzy_key: str*) → str

Match a fuzzy key against sequence of canonical key names.

Parameters

- **keys** – Sequence of canonical key names.
- **fuzzy_key** (*str*) – Fuzzy key to match against canonical keys.

Returns Canonical matching key name.

Raises **KeyError** – If fuzzy key does not match.

desmod.config.**fuzzy_lookup**(*config: Dict[str, Any], fuzzy_key: str*) → Tuple[str, Any]

Lookup a config key/value using a partially specified (fuzzy) key.

The lookup will succeed iff the provided *fuzzy_key* unambiguously matches the tail of a [fully-qualified] key in the *config* dict.

Parameters

- **config** (*dict*) – Configuration dict in which to lookup *fuzzy_key*.
- **fuzzy_key** (*str*) – Partially specified key to lookup in *config*.

Returns *(key, value)* tuple. The returned key is the regular, fully-qualified key name, not the provided *fuzzy_key*.

Raises ***ConfigError*** – For non-matching *fuzzy_key*.

1.3 desmod.component

Component is the building block for desmod models.

1.3.1 Hierarchy

A desmod model consists of a directed acyclical graph (DAG) of *Component* subclasses. Each Component is composed of zero or more child Components. A single top-level Component class is passed to the *simulate()* function to initiate simulation.

The *Component* hierarchy does not define the behavior of a model, but instead exists as a tool to build large models out of composable and encapsulated pieces.

1.3.2 Connections

Components connect to other components via connection objects. Each component is responsible for declaring the names of external connections as well as make connections for its child components. The final network of inter-component connections is neither directed (a connection object may enable two-way communication), acyclic (groups of components may form cyclical connections), nor constrained to match the component hierarchy.

Ultimately, a connection between two components means that each component instance has a [pythonic] reference to the connection object.

In the spirit of Python, the types connection objects are flexible and dynamic. A connection object may be of any type–it is up to the connected components to cooperatively decide how to use the connection object for communication. That said, some object types are more useful than others for connections. Some useful connection object types include:

- *desmod.queue.Queue*
- simpy.resources.resource.Resource

1.3.3 Processes

A component may have zero or more simulation processes (simpy.events.Process). It is these processes that give a model its simulation-time behavior. The process methods declared by components are started at simulation time. These "standing" processes may dynamically launch addtional processes using *self.env.process()*.

1.3.4 Use Cases

Given the flexibility components to have zero or more children, zero or more processes, and zero or more connections, it can be helpful to give names to the various roles components may play in a model.

- Structural Component – a component with child components, but no processes
- Behavioral Component – a component with processes, but no child components
- Hybrid Component – a component with child components and processes
- State Component – a component with neither children or processes

It is typical for the top-level component in a model to be purely structural, while behavioral components are leaves in the model DAG.

A component with neither children or processes may still be useful. Such a component could, for example, be used as a connection object.

class desmod.component.**Component**(*parent: Optional[Component], env: Optional[desmod.simulation.SimEnvironment] = None, name: Optional[str] = None, index: Optional[int] = None*)

Building block for composing models.

This class is meant to be subclassed. Component subclasses must declare their children, connections, and processes.

Parameters

- **parent** (Component) – Parent component or None for top-level Component.
- **env** (SimEnvironment) – SimPy simulation environment.
- **name** (*str*) – Optional name of Component instance.
- **index** (*int*) – Optional index of Component. This is used when multiple sibling components of the same type are instantiated as an array/list.

env
The simulation environment; a SimEnvironment instance.

name
The component name (str).

index
Index of Component instance within group of sibling instances. Will be None for un-grouped Components.

scope
String indicating the full scope of Component instance in the Component DAG.

children

error(**values*)
Log an error message.

warn(**values*)
Log a warning message.

info(**values*)
Log an informative message.

debug(**values*)
Log a debug message.

add_process(*g: Callable[[...], Generator[simpy.events.Event, Any, None]], *args, **kwargs*) → None
Add a process method to be run at simulation-time.

Subclasses should call this in *__init__()* to declare the process methods to be started at simulation-time.

Parameters

- **process_func** (*function*) – Typically a bound method of the Component subclass.
- **args** – arguments to pass to *process_func*.
- **kwargs** – keyword arguments to pass to *process_func*.

add_processes(**generators*) → None
Declare multiple processes at once.

This is a convenience wrapper for *add_process()* that may be used to quickly declare a list of process methods that do not require any arguments.

Parameters **process_funcs** – argument-less process functions (methods).

add_connections(**connection_names*) → None
Declare names of externally-provided connection objects.

The named connections must be connected (assigned) by an ancestor at elaboration time.

connect(*dst: desmod.component.Component*, *dst_connection: Any*, *src: Optional[Component] = None*, *src_connection: Optional[Any] = None*, *conn_obj: Optional[Any] = None*) → None
Assign connection object from source to destination component.

At elaboration-time, Components must call *connect()* to make the connections declared by descendant (child, grandchild, etc.) components.

Note: *connect()* is nominally called from *connect_children()*.

Parameters

- **dst** (Component) – Destination component being assigned the connection object.
- **dst_connection** (*str*) – Destination's name for the connection object.
- **src** (Component) – Source component providing the connection object. If omitted, the source component is assumed to be *self*.
- **src_connection** (*str*) – Source's name for the connection object. If omitted, *dst_connection* is used.
- **conn_obj** – The connection object to be assigned to the destination component. This parameter may typically be omitted in which case the connection object is resolved using *src* and *src_connection*.

connect_children() → None
Make connections for descendant components.

This method must be overridden in Component subclasses that need to make any connections on behalf of its descendant components. Connections are made using *connect()*.

classmethod pre_init(*env: desmod.simulation.SimEnvironment*) → None
Override-able class method called prior to model initialization.

Component subclasses may override this classmethod to gain access to the simulation environment (*env*) prior to `__init__()` being called.

elaborate() → None
Recursively elaborate the model.

The elaboration phase prepares the model for simulation. Descendant connections are made and components' processes are started at elaboration-time.

elab_hook() → None
Hook called after elaboration and before simulation phase.

Component subclasses may override *elab_hook()* to inject behavior after elaboration, but prior to simulation.

post_simulate() → None
Recursively run post-simulation hooks.

post_sim_hook() → None
Hook called after simulation completes.

Component subclasses may override *post_sim_hook()* to inject behavior after the simulation completes successfully. Note that *post_sim_hook()* will not be called if the simulation terminates with an unhandled exception.

get_result (*result: Dict[str, Any]*) → None
Recursively compose simulation result dict.

Upon successful completion of the simulation phase, each component in the model has the opportunity to add-to or modify the *result* dict via its *get_result_hook()* method.

The fully composed *result* dict is returned by `simulate()`.

Parameters **result** (*dict*) – Result dictionary to be modified.

get_result_hook (*result: Dict[str, Any]*) → None
Hook called after result is composed by descendant components.

1.4 desmod.dot

Generate graphical representation of component hierarchy.

Component hierarchy, connections, and processes can be represented graphically using the Graphviz DOT language.

The *component_to_dot()* function produces a DOT language string that can be rendered into a variety of formats using Graphviz tools. Because the component hierarchy, connections, and processes are determined dynamically, *component_to_dot()* must be called with an instantiated component. A good way to integrate this capabililty into a model is to call *component_to_dot()* from a component's *desmod.component.Component.elab_hook()* method.

The `dot` program from Graphviz may be used to render the generated DOT language description of the component hierarchy:

```
dot -Tpng -o foo.png foo.dot
```

For large component hierarchies, the `osage` program (also part of Graphviz) can produce a more compact layout:

```
osage -Tpng -o foo.png foo.dot
```

desmod.dot.**component_to_dot** (*top: desmod.component.Component*, *show_hierarchy: bool = True*, *show_connections: bool = True*, *show_processes: bool = True*, *colorscheme: str = ''*) → str
Produce a dot stream from a component hierarchy.

The DOT language representation of the component instance hierarchy can show the component hierarchy, the inter-component connections, components' processes, or any combination thereof.

Note: The *top* component hierarchy must be initialized and all connections must be made in order for *component_to_dot()* to inspect these graphs. The *desmod.component.Component.elab_hook()* method is a good place to call *component_to_dot()* since the model is fully elaborated at that point and simulation has not yet started.

Parameters

- **top** (Component) – Top-level component (instance).
- **show_hierarchy** (*bool*) – Should the component hierarchy be shown in the graph.

- **show_connections** (*bool*) – Should the inter-component connections be shown in the graph.
- **show_processes** (*bool*) – Should each component's processes be shown in the graph.
- **colorscheme** (*str*) – One of the Brewer color schemes supported by graphviz, e.g. "blues8" or "set27". Each level of the component hierarchy will use a different color from the color scheme. N.B. Brewer color schemes have between 3 and 12 colors; one should be chosen that has at least as many colors as the depth of the component hierarchy.

Returns str DOT language representation of the component/connection graph(s).

1.5 desmod.pool

Pool class for modeling a container of resources.

A pool models a container of homogeneous resources, similar to `simpy.resources.Container`, but with additional events when the container is empty or full. Resources are *Pool.put()* or *Pool.get()* to/from the pool in specified amounts. The pool's resources may be modeled as either discrete or continuous depending on whether the put/get amounts are *int* or *float*.

class desmod.pool.**Pool**(*env: simpy.core.Environment*, *capacity: Union[int, float] = inf*, *init: Union[int, float] = 0*, *hard_cap: bool = False*, *name: Optional[str] = None*)

Simulation pool of discrete or continuous resources.

Pool is similar to `simpy.resources.Container`. It provides a simulation-aware container for managing a shared pool of resources. The resources can be either discrete objects (like apples) or continuous (like water).

Resources are added and removed using *put()* and *get()*.

Parameters

- **env** – Simulation environment.
- **capacity** – Capacity of the pool; infinite by default.
- **hard_cap** – If specified, the pool overflows when the *capacity* is reached.
- **init_level** – Initial level of the pool.
- **name** – Optional name to associate with the queue.

capacity = None
Capacity of the pool (maximum level).

level = None
Current fill level of the pool.

remaining
Remaining pool capacity.

is_empty
Indicates whether the pool is empty.

is_full
Indicates whether the pool is full.

put
alias of `PoolPutEvent`

get
alias of `PoolGetEvent`

when_at_least
alias of PoolWhenAtLeastEvent

when_at_most
alias of PoolWhenAtMostEvent

when_any
alias of PoolWhenAnyEvent

when_full
alias of PoolWhenFullEvent

when_not_full
alias of PoolWhenNotFullEvent

when_empty
alias of PoolWhenEmptyEvent

class desmod.pool.**PriorityPool**(*env: simpy.core.Environment*, *capacity: Union[int, float] = inf*, *init: Union[int, float] = 0*, *hard_cap: bool = False*, *name: Optional[str] = None*)

Pool with prioritizied put() and get() requests.

A priority is provided with *put()* and *get()* requests. This priority determines the strict order in which requests are fulfilled. Requests of the same priority are serviced in strict FIFO order.

is_empty
Indicates whether the pool is empty.

is_full
Indicates whether the pool is full.

remaining
Remaining pool capacity.

when_any
alias of PoolWhenAnyEvent

when_at_least
alias of PoolWhenAtLeastEvent

when_at_most
alias of PoolWhenAtMostEvent

when_empty
alias of PoolWhenEmptyEvent

when_full
alias of PoolWhenFullEvent

when_not_full
alias of PoolWhenNotFullEvent

put
alias of PriorityPoolPutEvent

get
alias of PriorityPoolGetEvent

1.6 desmod.queue

Queue classes useful for modeling.

A queue may be used for inter-process message passing, resource pools, event sequences, and many other modeling applications. The *Queue* class implements a simulation-aware, general-purpose queue useful for these modeling applications.

The *PriorityQueue* class is an alternative to *Queue* that dequeues items in priority-order instead of *Queue*'s FIFO discipline.

class desmod.queue.**Queue**(*env: simpy.core.Environment*, *capacity: Union[int, float] = inf*, *hard_cap: bool = False*, *items: Iterable[ItemType] = ()*, *name: Optional[str] = None*)

Simulation queue of arbitrary items.

Queue is similar to `simpy.Store`. It provides a simulation-aware first-in first-out (FIFO) queue useful for passing messages between simulation processes or managing a pool of objects needed by multiple processes.

Items are enqueued and dequeued using *put()* and *get()*.

Parameters

- **env** – Simulation environment.
- **capacity** – Capacity of the queue; infinite by default.
- **hard_cap** – If specified, the queue overflows when the *capacity* is reached.
- **items** – Optional sequence of items to pre-populate the queue.
- **name** – Optional name to associate with the queue.

capacity = None
Capacity of the queue (maximum number of items).

size
Number of items in queue.

remaining
Remaining queue capacity.

is_empty
Indicates whether the queue is empty.

is_full
Indicates whether the queue is full.

peek() → ItemType
Peek at the next item in the queue.

put
alias of `QueuePutEvent`

get
alias of `QueueGetEvent`

when_at_least
alias of `QueueWhenAtLeastEvent`

when_at_most
alias of `QueueWhenAtMostEvent`

when_any
alias of QueueWhenAnyEvent

when_full
alias of QueueWhenFullEvent

when_not_full
alias of QueueWhenNotFullEvent

when_empty
alias of QueueWhenEmptyEvent

class desmod.queue.**PriorityQueue**(*env: simpy.core.Environment*, *capacity: Union[int, float] = inf*, *hard_cap: bool = False*, *items: Iterable[ItemType] = ()*, *name: Optional[str] = None*)
Specialized queue where items are dequeued in priority order.

Items in *PriorityQueue* must be orderable (implement __lt__()). Unorderable items may be used with *PriorityQueue* by wrapping with *PriorityItem*.

Items that evaluate less-than other items will be dequeued first.

get
alias of QueueGetEvent

is_empty
Indicates whether the queue is empty.

is_full
Indicates whether the queue is full.

peek() → ItemType
Peek at the next item in the queue.

put
alias of QueuePutEvent

remaining
Remaining queue capacity.

size
Number of items in queue.

when_any
alias of QueueWhenAnyEvent

when_at_least
alias of QueueWhenAtLeastEvent

when_at_most
alias of QueueWhenAtMostEvent

when_empty
alias of QueueWhenEmptyEvent

when_full
alias of QueueWhenFullEvent

when_not_full
alias of QueueWhenNotFullEvent

class desmod.queue.**PriorityItem**
Wrap items with explicit priority for use with *PriorityQueue*.

Parameters

- **priority** – Orderable priority value. Smaller values are dequeued first.
- **item** – Arbitrary item. Only the *priority* is determines dequeue order, so the *item* itself does not have to be orderable.

priority
Alias for field number 0

item
Alias for field number 1

1.7 desmod.simulation

Simulation model with batteries included.

class desmod.simulation.**SimEnvironment**(*config: Dict[str, Any]*)
Simulation Environment.

The *SimEnvironment* class is a simpy.Environment subclass that adds some useful features:

- Access to the configuration dictionary (*config*).
- Access to a seeded pseudo-random number generator (*rand*).
- Access to the simulation timescale (*timescale*).
- Access to the simulation duration (*duration*).

Some models may need to share additional state with all its *desmod.component.Component* instances. SimEnvironment may be subclassed to add additional members to achieve this sharing.

Parameters **config** (*dict*) – A fully-initialized configuration dictionary.

config
The configuration dictionary.

rand
The pseudo-random number generator; an instance of random.Random.

timescale
Simulation timescale (magnitude, units) tuple. The current simulation time is now * timescale.

duration
The intended simulation duration, in units of *timescale*.

tracemgr
TraceManager instance.

now
The current simulation time.

time(*t=None*, *unit='s'*)
The current simulation time scaled to specified unit.

Parameters

- **t** (*float*) – Time in simulation units. Default is *now*.
- **unit** (*str*) – Unit of time to scale to. Default is 's' (seconds).

Returns Simulation time scaled to to *unit*.

active_process
The currently active process of the environment.

process(*generator*)
Process an event yielding generator.

A generator (also known as a coroutine) can suspend its execution by yielding an event. `Process` will take care of resuming the generator with the value of that event once it has happened. The exception of failed events is thrown into the generator.

`Process` itself is an event, too. It is triggered, once the generator returns or raises an exception. The value of the process is the return value of the generator or the exception, respectively.

Processes can be interrupted during their execution by `interrupt()`.

timeout(*delay*, *value*)
A `Event` that gets triggered after a *delay* has passed.

This event is automatically triggered when it is created.

event()
An event that may happen at some point in time.

An event

- may happen (`triggered` is `False`),
- is going to happen (`triggered` is `True`) or
- has happened (`processed` is `True`).

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

An event has a list of `callbacks`. A callback can be any callable. Once an event gets processed, all callbacks will be invoked with the event as the single argument. Callbacks can check if the event was successful by examining *ok* and do further processing with the *value* it has produced.

Failed events are never silently ignored and will raise an exception upon being processed. If a callback handles an exception, it must set `defused` to `True` to prevent this.

This class also implements `__and__()` (`&`) and `__or__()` (`|`). If you concatenate two events using one of these operators, a `Condition` event is generated that lets you wait for both or one of them.

all_of(*events*)
A `Condition` event that is triggered if all of a list of *events* have been successfully triggered. Fails immediately if any of *events* failed.

any_of(*events*)
A `Condition` event that is triggered if any of a list of *events* has been successfully triggered. Fails immediately if any of *events* failed.

schedule(*event: simpy.events.Event*, *priority: NewType.<locals>.new_type = 1*, *delay: Union[int, float] = 0*) → None
Schedule an *event* with a given *priority* and a *delay*.

peek() → Union[int, float]
Get the time of the next scheduled event. Return `Infinity` if there is no further event.

step() → None
Process the next event.

Raise an `EmptySchedule` if no further events are available.

desmod.simulation.**simulate**(*config: Dict[str, Any], top_type: Type[Component], env_type: Type[desmod.simulation.SimEnvironment] = <class 'desmod.simulation.SimEnvironment'>, reraise: bool = True, progress_manager=<function standalone_progress_manager>*) → Dict[str, Any]

Initialize, elaborate, and run a simulation.

All exceptions are caught by *simulate()* so they can be logged and captured in the result file. By default, any unhandled exception caught by *simulate()* will be re-raised. Setting *reraise* to False prevents exceptions from propagating to the caller. Instead, the returned result dict will indicate if an exception occurred via the 'sim.exception' item.

Parameters

- **config** (*dict*) – Configuration dictionary for the simulation.
- **top_type** – The model's top-level Component subclass.
- **env_type** – *SimEnvironment* subclass.
- **reraise** (*bool*) – Should unhandled exceptions propogate to the caller.

Returns Dictionary containing the model-specific results of the simulation.

desmod.simulation.**simulate_factors**(*base_config: Dict[str, Any], factors: List[Tuple[List[str], List[Any]]], top_type: Type[Component], env_type: Type[desmod.simulation.SimEnvironment] = <class 'desmod.simulation.SimEnvironment'>, jobs: Optional[int] = None, config_filter: Optional[Callable[[Dict[str, Any]], bool]] = None*) → List[Dict[str, Any]]

Run multi-factor simulations in separate processes.

The *factors* are used to compose specialized config dictionaries for the simulations.

The multiprocessing module is used run each simulation with a separate Python process. This allows multi-factor simulations to run in parallel on all available CPU cores.

Parameters

- **base_config** (*dict*) – Base configuration dictionary to be specialized.
- **factors** (*list*) – List of factors.
- **top_type** – The model's top-level Component subclass.
- **env_type** – *SimEnvironment* subclass.
- **jobs** (*int*) – User specified number of concurent processes.
- **config_filter** (*function*) – A function which will be passed a config and returns a bool to filter.

Returns Sequence of result dictionaries for each simulation.

CHAPTER 2

Examples

2.1 Gas Station

This example expands upon SimPy's Gas Station Refueling example, demonstrating various desmod features.

Note: Desmod's goal is to support large-scale modeling. Thus this example is somewhat larger-scale than the SimPy model it expands upon.

```
"""Model refueling at several gas stations.

Each gas station has several fuel pumps and a single, shared reservoir. Each
arrving car pumps gas from the reservoir via a fuel pump.

As the gas station's reservoir empties, a request is made to a tanker truck
company to send a truck to refill the reservoir. The tanker company maintains a
fleet of tanker trucks.

This example demonstrates core desmod concepts including:
 - Modeling using Component subclasses
 - The "batteries-included" simulation environment
 - Centralized configuration
 - Logging

"""
from itertools import count, cycle

from simpy import Resource

from desmod.component import Component
from desmod.dot import generate_dot
from desmod.pool import Pool
from desmod.queue import Queue
```

(continues on next page)

(continued from previous page)

```
from desmod.simulation import simulate


class Top(Component):
    """Every model has a single top-level Component.

    For this gas station model, the top level components are gas stations and a
    tanker truck company.

    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # The simulation configuration is available everywhere via the
        # simulation environment.
        num_gas_stations = self.env.config.get('gas_station.count', 1)

        # Instantiate GasStation components. An index is passed so that each
        # child gas station gets a unique name.
        self.gas_stations = [GasStation(self, index=i) for i in range(num_gas_
↪stations)]

        # There is just one tanker company.
        self.tanker_company = TankerCompany(self)

    def connect_children(self):
        # This function is called during the elaboration phase, i.e. after all
        # of the components have been instantiated, but before the simulation
        # phase.
        for gas_station in self.gas_stations:
            # Each GasStation instance gets a reference to (is connected to)
            # the tanker_company instance. This demonstrates the most
            # abbreviated way to call connect().
            self.connect(gas_station, 'tanker_company')

    def elab_hook(self):
        generate_dot(self)


class TankerCompany(Component):
    """The tanker company owns and dispatches its fleet of tanker trunks."""

    # This base_name is used to build names and scopes of component instances.
    base_name = 'tankerco'

    def __init__(self, *args, **kwargs):
        # Many Component subclasses can simply forward *args and **kwargs to
        # the superclass initializer; although Component subclasses may also
        # have custom positional and keyword arguments.
        super().__init__(*args, **kwargs)
        num_tankers = self.env.config.get('tanker.count', 1)

        # Instantiate the fleet of tanker trucks.
        trucks = [TankerTruck(self, index=i) for i in range(num_tankers)]

        # Trucks are dispatched in a simple round-robin fashion.
```

(continues on next page)

(continued from previous page)

```
        self.trucks_round_robin = cycle(trucks)

    def request_truck(self, gas_station, done_event):
        """Called by gas stations to request a truck to refill its reservior.

        Returns an event that the gas station must yield for.

        """
        truck = next(self.trucks_round_robin)

        # Each component has debug(), info(), warn(), and error() log methods.
        # Log lines are automatically annotated with the simulation time and
        # the scope of the component doing the logging.
        self.info(f'dispatching {truck.name} to {gas_station.name}')
        return truck.dispatch(gas_station, done_event)


class TankerTruck(Component):
    """Tanker trucks carry fuel to gas stations.

    Each tanker truck has a queue of gas stations it must visit. When the
    truck's tank becomes empty, it must go refill itself.

    """

    base_name = 'truck'

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.pump_rate = self.env.config.get('tanker.pump_rate', 10)
        self.avg_travel = self.env.config.get('tanker.travel_time', 600)
        tank_capacity = self.env.config.get('tanker.capacity', 200)
        self.tank = Pool(self.env, tank_capacity)

        # This auto_probe() call uses the self.tank Pool get/put hooks so that
        # whenever it's level changes, the new level is noted in the log.
        self.auto_probe('tank', log={})

        # The parent TankerCompany enqueues instructions to this queue.
        self._instructions = Queue(self.env)

        # Declare a persistant process to be started at simulation-time.
        self.add_process(self._dispatch_loop)

    def dispatch(self, gas_station, done_event):
        """Append dispatch instructions to the truck's queue."""
        return self._instructions.put((gas_station, done_event))

    def _dispatch_loop(self):
        """This is the tanker truck's main behavior. Travel, pump, refill..."""
        while True:
            if not self.tank.level:
                self.info('going for refill')

                # Desmod simulation environments come equipped with a
                # random.Random() instance seeded based on the 'sim.seed'
                # configuration key.
```

(continues on next page)

(continued from previous page)

```
                travel_time = self.env.rand.expovariate(1 / self.avg_travel)
                yield self.env.timeout(travel_time)

                self.info('refilling')
                pump_time = self.tank.capacity / self.pump_rate
                yield self.env.timeout(pump_time)

                yield self.tank.put(self.tank.capacity)
                self.info(f'refilled {self.tank.capacity}L in {pump_time:.0f}s')

            gas_station, done_event = yield self._instructions.get()
            self.info(f'traveling to {gas_station.name}')
            travel_time = self.env.rand.expovariate(1 / self.avg_travel)
            yield self.env.timeout(travel_time)
            self.info(f'arrived at {gas_station.name}')
            while self.tank.level and (
                gas_station.reservoir.level < gas_station.reservoir.capacity
            ):
                yield self.env.timeout(1 / self.pump_rate)
                yield gas_station.reservoir.put(1)
                yield self.tank.get(1)
            self.info('done pumping')
            done_event.succeed()


class GasStation(Component):
    """A gas station has a fuel reservoir shared among several fuel pumps.

    The gas station has a traffic generator process that causes cars to arrive
    to fill up their tanks.

    As the cars fill up, the reservoir's level goes down. When the level goes
    below a critical threshold, the gas station makes a request to the tanker
    company for a tanker truck to refill the reservoir.

    """

    base_name = 'station'

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        config = self.env.config
        self.add_connections('tanker_company')
        self.arrival_interval = config.get('gas_station.arrival_interval', 60)

        station_capacity = config.get('gas_station.capacity', 200)
        self.reservoir = Pool(
            self.env, capacity=station_capacity, init=station_capacity
        )
        self.auto_probe('reservoir', log={})

        threshold_pct = config.get('gas_station.threshold_pct', 10)
        self.reservoir_low_water = threshold_pct * station_capacity / 100

        self.pump_rate = config.get('gas_station.pump_rate', 2)
        num_pumps = config.get('gas_station.pumps', 2)
        self.fuel_pumps = Resource(self.env, capacity=num_pumps)
```

(continues on next page)

(continued from previous page)

```
        self.auto_probe('fuel_pumps', log={})

        self.car_capacity = config.get('car.capacity', 50)
        self.car_level_range = config.get('car.level', [5, 25])

        # A gas station has two persistent processes. One to monitor the
        # reservoir level and one that models the arrival of cars at the
        # station. Desmod starts these processes before simulation phase.
        self.add_processes(self._monitor_reservoir, self._traffic_generator)

    @property
    def reservoir_pct(self):
        return self.reservoir.level / self.reservoir.capacity * 100

    def _monitor_reservoir(self):
        """Periodically monitor reservoir level.

        The a request is made to the tanker company when the reservoir falls
        below a critical threshold.

        """
        while True:
            yield self.reservoir.when_at_most(self.reservoir_low_water)
            done_event = self.env.event()
            yield self.tanker_company.request_truck(self, done_event)
            yield done_event

    def _traffic_generator(self):
        """Model the sporadic arrival of cars to the gas station."""
        for i in count():
            interval = self.env.rand.expovariate(1 / self.arrival_interval)
            yield self.env.timeout(interval)
            self.env.process(self._car(i))

    def _car(self, i):
        """Model a car transacting fuel."""
        with self.fuel_pumps.request() as pump_req:
            self.info(f'car{i} awaiting pump')
            yield pump_req
            self.info(f'car{i} at pump')
            car_level = self.env.rand.randint(*self.car_level_range)
            amount = self.car_capacity - car_level
            t0 = self.env.now
            for _ in range(amount):
                yield self.reservoir.get(1)
                yield self.env.timeout(1 / self.pump_rate)
            pump_time = self.env.now - t0
            self.info(f'car{i} pumped {amount}L in {pump_time:.0f}s')


# Desmod uses a plain dictionary to represent the simulation configuration.
# The various 'sim.xxx' keys are reserved for desmod while the remainder are
# application-specific.
config = {
    'car.capacity': 50,
    'car.level': [5, 25],
    'gas_station.capacity': 200,
```

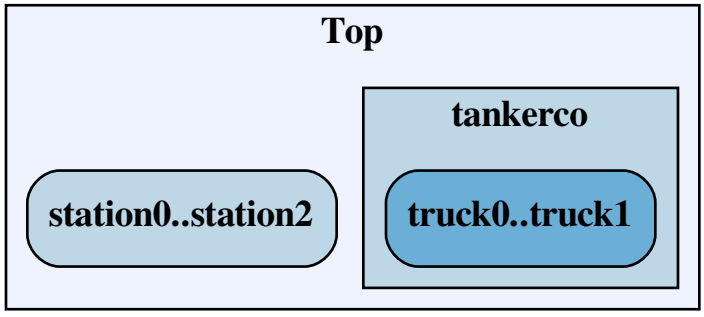
(continues on next page)

(continued from previous page)

```
    'gas_station.count': 3,
    'gas_station.pump_rate': 2,
    'gas_station.pumps': 2,
    'gas_station.arrival_interval': 60,
    'sim.dot.enable': True,
    'sim.dot.colorscheme': 'blues5',
    'sim.duration': '500 s',
    'sim.log.enable': True,
    'sim.log.file': 'sim.log',
    'sim.log.format': '{level:7} {ts:.3f} {ts_unit}: {scope:<16}:',
    'sim.log.level': 'INFO',
    'sim.result.file': 'results.yaml',
    'sim.seed': 42,
    'sim.timescale': 's',
    'sim.workspace': 'workspace',
    'tanker.capacity': 200,
    'tanker.count': 2,
    'tanker.pump_rate': 10,
    'tanker.travel_time': 100,
}

if __name__ == '__main__':
    # Desmod takes responsibility for instantiating and elaborating the model,
    # thus we only need to pass the configuration dict and the top-level
    # Component class (Top) to simulate().
    simulate(config, Top)
```

The model hierarchy is captured during elaboration as a DOT graph. See the *desmod.dot* documentation for more detail on DOT output.



The simulation log, `sim.log`, shows what happened during the simulation:

```
INFO      0.000 s: tankerco.truck0 : going for refill
INFO      0.000 s: tankerco.truck1 : going for refill
INFO      1.520 s: station1        : car0 awaiting pump
INFO      1.520 s: station1        : car0 at pump
INFO     15.520 s: station1         : car0 pumped 28L in 14s
```

(continues on next page)

(continued from previous page)

```
INFO     19.297 s: station2        : car0 awaiting pump
INFO     19.297 s: station2        : car0 at pump
INFO     24.755 s: station2        : car1 awaiting pump
INFO     24.755 s: station2        : car1 at pump
INFO     25.259 s: tankerco.truck0 : refilling
INFO     26.693 s: station2        : car2 awaiting pump
INFO     35.297 s: station2        : car0 pumped 32L in 16s
INFO     35.297 s: station2        : car2 at pump
INFO     41.496 s: station2        : car3 awaiting pump
INFO     45.259 s: tankerco.truck0 : refilled 200L in 20s
INFO     46.255 s: station2        : car1 pumped 43L in 22s
INFO     46.255 s: station2        : car3 at pump
INFO     49.797 s: station2        : car2 pumped 29L in 14s
INFO     60.255 s: station2        : car3 pumped 28L in 14s
INFO     61.204 s: station0        : car0 awaiting pump
INFO     61.204 s: station0        : car0 at pump
INFO     69.270 s: station1        : car1 awaiting pump
INFO     69.270 s: station1        : car1 at pump
INFO     73.704 s: station0        : car0 pumped 25L in 13s
INFO     74.505 s: station0        : car1 awaiting pump
INFO     74.505 s: station0        : car1 at pump
INFO     85.270 s: station1        : car1 pumped 32L in 16s
INFO     88.005 s: station0        : car1 pumped 27L in 14s
INFO     89.447 s: station0        : car2 awaiting pump
INFO     89.447 s: station0        : car2 at pump
INFO     96.777 s: station2        : car4 awaiting pump
INFO     96.777 s: station2        : car4 at pump
INFO    109.006 s: station0        : car3 awaiting pump
INFO    109.006 s: station0        : car3 at pump
INFO    111.947 s: station0        : car2 pumped 45L in 22s
INFO    116.777 s: station2        : car4 pumped 40L in 20s
INFO    126.506 s: station0        : car3 pumped 35L in 18s
INFO    133.359 s: tankerco.truck1 : refilling
INFO    141.774 s: station1        : car2 awaiting pump
INFO    141.774 s: station1        : car2 at pump
INFO    153.359 s: tankerco.truck1 : refilled 200L in 20s
INFO    161.274 s: station1        : car2 pumped 39L in 20s
INFO    161.307 s: station1        : car3 awaiting pump
INFO    161.307 s: station1        : car3 at pump
INFO    178.807 s: station1        : car3 pumped 35L in 18s
INFO    180.874 s: station0        : car4 awaiting pump
INFO    180.874 s: station0        : car4 at pump
INFO    182.106 s: station2        : car5 awaiting pump
INFO    182.106 s: station2        : car5 at pump
INFO    185.606 s: tankerco        : dispatching truck0 to station2
INFO    185.606 s: tankerco.truck0 : traveling to station2
INFO    187.343 s: station0        : car5 awaiting pump
INFO    187.343 s: station0        : car5 at pump
INFO    188.209 s: station2        : car6 awaiting pump
INFO    188.209 s: station2        : car6 at pump
INFO    195.843 s: tankerco        : dispatching truck1 to station0
INFO    195.843 s: tankerco.truck1 : traveling to station0
INFO    197.374 s: station0        : car4 pumped 33L in 16s
INFO    202.843 s: station0        : car5 pumped 31L in 16s
INFO    204.051 s: tankerco.truck1 : arrived at station0
INFO    223.651 s: tankerco.truck1 : done pumping
INFO    234.311 s: station2        : car7 awaiting pump
```

(continues on next page)

(continued from previous page)

```
INFO     255.130 s: station2        : car8 awaiting pump
INFO     278.171 s: tankerco.truck0 : arrived at station2
INFO     285.271 s: station2        : car5 pumped 34L in 103s
INFO     285.271 s: station2        : car7 at pump
INFO     286.087 s: station0        : car6 awaiting pump
INFO     286.087 s: station0        : car6 at pump
INFO     290.871 s: station2        : car6 pumped 33L in 103s
INFO     290.871 s: station2        : car8 at pump
INFO     298.171 s: tankerco.truck0 : done pumping
INFO     298.171 s: tankerco.truck0 : going for refill
INFO     302.271 s: station2        : car7 pumped 34L in 17s
INFO     307.587 s: station0        : car6 pumped 43L in 22s
INFO     312.871 s: station2        : car8 pumped 44L in 22s
INFO     314.565 s: station2        : car9 awaiting pump
INFO     314.565 s: station2        : car9 at pump
INFO     336.065 s: station2        : car9 pumped 43L in 22s
INFO     337.760 s: station0        : car7 awaiting pump
INFO     337.760 s: station0        : car7 at pump
INFO     350.399 s: station1        : car4 awaiting pump
INFO     350.399 s: station1        : car4 at pump
INFO     358.760 s: station0        : car7 pumped 42L in 21s
INFO     365.899 s: station1        : car4 pumped 31L in 16s
INFO     379.093 s: station1        : car5 awaiting pump
INFO     379.093 s: station1        : car5 at pump
INFO     386.093 s: tankerco        : dispatching truck0 to station1
INFO     396.093 s: station1        : car5 pumped 34L in 17s
INFO     403.551 s: station2        : car10 awaiting pump
INFO     403.551 s: station2        : car10 at pump
INFO     406.424 s: tankerco.truck0 : refilling
INFO     413.051 s: tankerco        : dispatching truck1 to station2
INFO     413.051 s: tankerco.truck1 : traveling to station2
INFO     414.202 s: station2        : car11 awaiting pump
INFO     414.202 s: station2        : car11 at pump
INFO     426.424 s: tankerco.truck0 : refilled 200L in 20s
INFO     426.424 s: tankerco.truck0 : traveling to station1
INFO     432.837 s: station2        : car12 awaiting pump
INFO     433.832 s: tankerco.truck0 : arrived at station1
INFO     436.561 s: tankerco.truck1 : arrived at station2
INFO     436.961 s: tankerco.truck1 : done pumping
INFO     436.961 s: tankerco.truck1 : going for refill
INFO     436.961 s: tankerco        : dispatching truck0 to station2
INFO     439.677 s: station1        : car6 awaiting pump
INFO     439.677 s: station1        : car6 at pump
INFO     453.753 s: station0        : car8 awaiting pump
INFO     453.753 s: station0        : car8 at pump
INFO     453.832 s: tankerco.truck0 : done pumping
INFO     453.832 s: tankerco.truck0 : going for refill
INFO     455.177 s: station1        : car6 pumped 31L in 16s
INFO     456.524 s: station1        : car7 awaiting pump
INFO     456.524 s: station1        : car7 at pump
INFO     466.253 s: station0        : car8 pumped 25L in 12s
INFO     471.402 s: station1        : car8 awaiting pump
INFO     471.402 s: station1        : car8 at pump
INFO     474.024 s: station1        : car7 pumped 35L in 18s
INFO     482.382 s: station0        : car9 awaiting pump
INFO     482.382 s: station0        : car9 at pump
INFO     493.305 s: station2        : car13 awaiting pump
```

(continues on next page)

(continued from previous page)

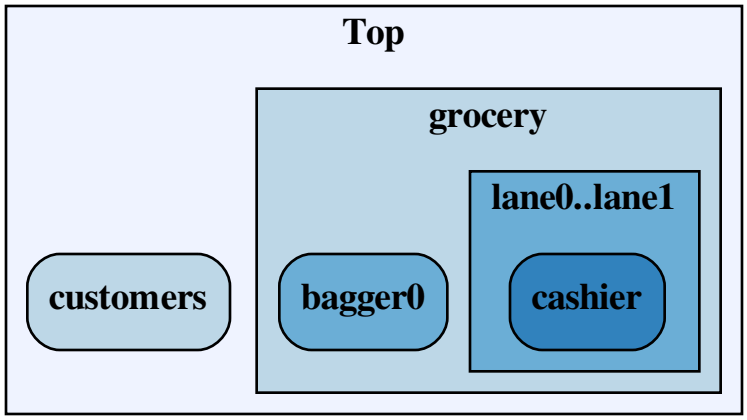
```
INFO     493.402 s: station1        : car8 pumped 44L in 22s
INFO     497.990 s: station0        : car10 awaiting pump
INFO     497.990 s: station0        : car10 at pump
```

This example does not make heavy use of desmod's result-gathering capability, but we can nonetheless see the minimal `results.yaml` file generated from the simulation:

```
config:
  car.capacity: 50
  car.level: [5, 25]
  gas_station.arrival_interval: 60
  gas_station.capacity: 200
  gas_station.count: 3
  gas_station.pump_rate: 2
  gas_station.pumps: 2
  meta.sim.workspace: workspace
  sim.config.file: null
  sim.db.enable: false
  sim.db.persist: true
  sim.dot.all.file: all.dot
  sim.dot.colorscheme: blues5
  sim.dot.conn.file: conn.dot
  sim.dot.enable: true
  sim.dot.hier.file: hier.dot
  sim.duration: 500 s
  sim.log.buffering: -1
  sim.log.enable: true
  sim.log.exclude_pat: []
  sim.log.file: sim.log
  sim.log.format: '{level:7} {ts:.3f} {ts_unit}: {scope:<16}:'
  sim.log.include_pat: [.*]
  sim.log.level: INFO
  sim.log.persist: true
  sim.progress.enable: false
  sim.progress.max_width: null
  sim.progress.update_period: 1 s
  sim.result.file: results.yaml
  sim.seed: 42
  sim.timescale: s
  sim.vcd.enable: false
  sim.vcd.persist: true
  sim.workspace: workspace
  sim.workspace.overwrite: false
  tanker.capacity: 200
  tanker.count: 2
  tanker.pump_rate: 10
  tanker.travel_time: 100
sim.exception: null
sim.now: 500.0
sim.runtime: 0.039155590000000004
sim.time: 500
```

2.2 Grocery Store

This example aims to demonstrate a wide breadth of desmod features.




```
"""Model grocery store checkout lanes.

A grocery store checkout system is modeled. Each grocery store has one or more
checkout lanes. Each lane has a cashier that scans customers' items. Zero or
more baggers bag items after the cashier scans them. Cashiers will also bag
items if there is no bagger helping at their lane.

Several bagger assignment policies are implemented. This model helps determine
the optimal policy under various conditions. The model is also useful for
estimating bagger, checkout lane, and cashier resources needed for various
customer profiles.

"""
from argparse import ArgumentParser
from datetime import timedelta
from functools import partial
from itertools import count

from simpy import Container, Resource
from vcd.gtkw import GTKWSave

from desmod.component import Component
from desmod.config import apply_user_overrides, parse_user_factors
from desmod.dot import generate_dot
from desmod.queue import Queue
from desmod.simulation import simulate, simulate_factors


class Top(Component):
    """The top-level component of the model."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.customers = Customers(self)
```

(continues on next page)

(continued from previous page)

```
        self.grocery = GroceryStore(self)

    def connect_children(self):
        self.connect(self.customers, 'grocery')

    @classmethod
    def pre_init(cls, env):
        # Compose a GTKWave save file that lays-out the various VCD signals in
        # a meaningful manner. This must be done at pre-init time to allow
        # sim.gtkw.live to work.
        analog_kwargs = {
            'datafmt': 'dec',
            'color': 'cycle',
            'extraflags': ['analog_step'],
        }
        with open(env.config['sim.gtkw.file'], 'w') as gtkw_file:
            gtkw = GTKWSave(gtkw_file)
            gtkw.dumpfile(env.config['sim.vcd.dump_file'], abspath=False)
            gtkw.treeopen('grocery')
            gtkw.signals_width(300)
            gtkw.trace('customers.active', **analog_kwargs)
            for i in range(env.config['grocery.num_lanes']):
                with gtkw.group(f'Lane{i}'):
                    scope = f'grocery.lane{i}'
                    gtkw.trace(f'{scope}.customer_queue', **analog_kwargs)
                    gtkw.trace(f'{scope}.feed_belt', **analog_kwargs)
                    gtkw.trace(f'{scope}.bag_area', **analog_kwargs)
                    gtkw.trace(f'{scope}.baggers', **analog_kwargs)

    def elab_hook(self):
        # We generate DOT representations of the component hierarchy. It is
        # only after elaboration that the component tree is fully populated and
        # connected, thus generate_dot() is called here in elab_hook().
        generate_dot(self)


class GroceryStore(Component):
    """Model a grocery store with checkout lanes, cashiers, and baggers."""

    base_name = 'grocery'

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        num_lanes = self.env.config['grocery.num_lanes']
        self.checkout_lanes = [CheckoutLane(self, index=i) for i in range(num_lanes)]

        num_baggers = self.env.config['grocery.num_baggers']
        self.baggers = [Bagger(self, index=i) for i in range(num_baggers)]

    def connect_children(self):
        # The baggers move between checkout lanes depending on bagger.policy,
        # so each bagger must be connected to all of the checkout lanes.
        for bagger in self.baggers:
            self.connect(bagger, 'checkout_lanes')


class CheckoutLane(Component):
```

(continues on next page)

(continued from previous page)

```
    """Model a grocery store checkout lane.

    Each lane has a customer queue which is modeled with a
    :class:`simpy.Resource`. Customers are addressed in a first-come,
    first-serve manner.

    Once a customer reaches the front of the checkout lane's line, they place
    their items on the lane's feed belt. The feed belt is modeled as a
    :class:`desmod.queue.Queue` and has limited capacity for items. That
    capacity is configurable with `checkout.feed_capacity`.

    The checkout lane's cashier takes items from the feed belt, scans them, and
    places them in the lane's bagging area. The bagging area is also modeled as
    a queue with limited capacity.

    Finally, a checkout lane may have zero or more baggers assigned to it. A
    :class:`simpy.Container` is used to keep track of how many baggers are
    present at each lane.

    """

    base_name = 'lane'

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.cashier = Cashier(self)

        self.customer_queue = Resource(self.env)
        self.auto_probe('customer_queue', trace_queue=True, vcd={'init': 0})

        feed_capacity = self.env.config['checkout.feed_capacity']
        self.feed_belt = Queue(self.env, capacity=feed_capacity)
        self.auto_probe('feed_belt', vcd={})

        bag_area_capacity = self.env.config['checkout.bag_area_capacity']
        self.bag_area = Queue(self.env, capacity=bag_area_capacity)
        self.auto_probe('bag_area', vcd={})

        self.baggers = Container(self.env)
        self.auto_probe('baggers', vcd={})

    def connect_children(self):
        self.connect(self.cashier, 'lane', conn_obj=self)


class Cashier(Component):
    """Model checkout lane cashier.

    A cashier occupies a single checkout lane. They take items from the lane's
    feeder belt, scan the item, and place the item in the lane's bagging area.

    If no bagger personel is present, the cashier will also perform bagging,
    but a cashier cannot scan and bag at the same time.

    """
```

(continues on next page)

(continued from previous page)

```
    base_name = 'cashier'

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.add_connections('lane')
        self.add_process(self.checkout)

        # Use exponential distribution to model item scan and bag times.
        self.scan_dist = partial(
            self.env.rand.expovariate, 1 / self.env.config['cashier.scan_time']
        )

        self.bag_dist = partial(
            self.env.rand.expovariate, 1 / self.env.config['cashier.bag_time']
        )

    def checkout(self):
        """Cashier checkout behavior."""
        while True:
            if not self.lane.baggers.level and self.lane.bag_area.is_full:
                yield self.env.process(self.bag_items())
            item, done_event = yield self.lane.feed_belt.get()
            yield self.env.timeout(self.scan_dist())
            yield self.lane.bag_area.put((item, done_event))
            if done_event is not None:
                # A customer's final item comes with a done event. The bag area
                # must be emptied between customers.
                yield self.env.process(self.bag_items())

    def bag_items(self):
        while not self.lane.bag_area.is_empty and not self.lane.baggers.level:
            item, done_event = yield self.lane.bag_area.get()
            yield self.env.timeout(self.bag_dist())
            if done_event is not None:
                # Notify the customer that their checkout is complete.
                done_event.succeed()


class Bagger(Component):
    """Model a grocery store bagger employee.

    Baggers take customer items from checkout lane bagging areas and bag them.

    Several policies for how baggers are assigned to checkout lanes can be
    configured with `bagger.policy`.

    """

    base_name = 'bagger'

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.add_connections('checkout_lanes')
        self.bag_dist = partial(
            self.env.rand.expovariate, 1 / self.env.config['bagger.bag_time']
        )
```

(continues on next page)

(continued from previous page)

```
        policy = self.env.config['bagger.policy']
        if policy == 'float-aggressive':
            self.add_process(self.policy_float_aggressive)
        elif policy == 'float-lazy':
            self.add_process(self.policy_float_lazy)
        elif policy == 'fixed-lane':
            self.add_process(self.policy_fixed_lane)
        else:
            raise ValueError(f'invalid bagger.policy {policy}')

    def policy_float_aggressive(self):
        """Assign bagger to the first lane with any baggable items.

        The bagger floats between checkout lanes. As soon as a lane is
        identified with any baggable items, the bagger assigns to that lane and
        bags until the lane's bag area is empty.

        """
        while True:
            yield self.env.any_of(
                lane.bag_area.when_any() for lane in self.checkout_lanes
            )

            lanes = reversed(
                sorted(
                    filter(lambda lane: not lane.baggers.level, self.checkout_lanes),
                    key=lambda lane: lane.bag_area.size,
                )
            )
            for lane in lanes:
                yield lane.baggers.put(1)
                self.debug('assigned to lane', lane.index)
                yield self.env.process(self.bag_items(lane.bag_area))
                yield lane.baggers.get(1)
                self.debug('leave lane', lane.index)
                break

    def policy_float_lazy(self):
        """Assign bagger to lane with full bagging area.

        The bagger remains idle until he identifies a lane with a full bagging
        area. The bagger bags at that lane until the bagging area is emptied.

        """
        while True:
            yield self.env.any_of(
                lane.bag_area.when_full() for lane in self.checkout_lanes
            )

            for lane in filter(lambda lane: lane.bag_area.is_full, self.checkout_
↪lanes):
                yield lane.baggers.put(1)
                self.debug('assigned to lane', lane.index)
                yield self.env.process(self.bag_items(lane.bag_area))
                yield lane.baggers.get(1)
                self.debug('leave lane', lane.index)
                break
```

(continues on next page)

(continued from previous page)

```
    def policy_fixed_lane(self):
        """Static assignment of bagger to a lane.

        The bagger finds the first lane with no other baggers and stays there.

        """
        _, lane = min((lane.baggers.level, lane) for lane in self.checkout_lanes)
        yield lane.baggers.put(1)
        self.debug('assigned to lane', lane.index)
        while True:
            yield lane.bag_area.when_any()
            yield self.env.process(self.bag_items(lane.bag_area))

    def bag_items(self, bag_area):
        while not bag_area.is_empty:
            item, done_event = yield bag_area.get()
            yield self.env.timeout(self.bag_dist())
            if done_event is not None:
                done_event.succeed()


class Customers(Component):
    """Model customer arrival rate and in-store behavior.

    Each customer's arrival time, number of items, and shopping time is
    determined by configuration.

    A new process is spawned for each customer.

    A "customers" database table captures per-customer checkout times. A
    primary goal for this model is optimizing customer checkout time (latency)
    and throughput.

    """

    base_name = 'customers'

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.add_connections('grocery')
        self.add_process(self.generate_customers)
        self.active = Container(self.env)
        self.auto_probe('active', vcd={})
        if self.env.tracemgr.sqlite_tracer.enabled:
            self.db = self.env.tracemgr.sqlite_tracer.db
            self.db.execute(
                'CREATE TABLE customers '
                '(cust_id INTEGER PRIMARY KEY,'
                ' num_items INTEGER,'
                ' shop_time REAL,'
                ' checkout_time REAL)'
            )
        else:
            self.db = None

    def generate_customers(self):
```

(continues on next page)

(continued from previous page)

```
        """Generate grocery store customers.

        Various configuration parameters determine the distribution of customer
        arrival times as well as the number of items each customer will shop
        for.

        """
        cust_id = count()
        arrival_interval_dist = partial(
            self.env.rand.expovariate, 1 / self.env.config['customer.arrival_interval
↪']
        )
        time_per_item_dist = partial(
            self.env.rand.expovariate, 1 / self.env.config['customer.time_per_item']
        )
        num_items_mu = self.env.config['customer.num_items.mu']
        num_items_sigma = self.env.config['customer.num_items.sigma']
        num_items_dist = partial(
            self.env.rand.normalvariate, num_items_mu, num_items_sigma
        )

        while True:
            num_items = max(1, round(num_items_dist()))
            self.env.process(
                self.customer(
                    next(cust_id), num_items, shop_time=num_items * time_per_item_
↪dist()
                )
            )
            yield self.env.timeout(arrival_interval_dist())

    def customer(self, cust_id, num_items, shop_time):
        """Grocery store customer behavior."""
        yield self.active.put(1)
        self.debug(cust_id, 'start shopping for', num_items, 'items')
        yield self.env.timeout(shop_time)
        self.debug(cust_id, 'ready to checkout after', timedelta(seconds=shop_time))

        t0 = self.env.now

        lane = sorted(
            self.grocery.checkout_lanes, key=lambda lane: len(lane.customer_queue.
↪queue)
        )[0]

        with lane.customer_queue.request() as req:
            self.debug('enter queue', lane.index)
            yield req
            for i in range(num_items - 1):
                yield lane.feed_belt.put((i, None))
            checkout_done = self.env.event()
            yield lane.feed_belt.put((num_items - 1, checkout_done))

        yield checkout_done
        checkout_time = self.env.now - t0
        self.debug(cust_id, 'done checking out after', timedelta(seconds=checkout_
↪time))
```

(continues on next page)

(continued from previous page)

```
        yield self.active.get(1)
        if self.db:
            self.db.execute(
                'INSERT INTO customers '
                '(cust_id, num_items, shop_time, checkout_time) '
                'VALUES (?,?,?,?)',
                (cust_id, num_items, shop_time, checkout_time),
            )

    def get_result_hook(self, result):
        if not self.db:
            return
        result['checkout_time_avg'] = self.db.execute(
            'SELECT AVG(checkout_time) FROM customers'
        ).fetchone()[0]
        result['checkout_time_min'] = self.db.execute(
            'SELECT MIN(checkout_time) FROM customers'
        ).fetchone()[0]
        result['checkout_time_max'] = self.db.execute(
            'SELECT MAX(checkout_time) FROM customers'
        ).fetchone()[0]
        result['customers_total'] = self.db.execute(
            'SELECT COUNT() FROM customers'
        ).fetchone()[0]
        result['customers_per_hour'] = result['customers_total'] / (
            self.env.time() / 3600
        )


if __name__ == '__main__':
    config = {
        'bagger.bag_time': 1.5,
        'bagger.policy': 'float-aggressive',
        'cashier.bag_time': 2.0,
        'cashier.scan_time': 2.0,
        'checkout.bag_area_capacity': 15,
        'checkout.feed_capacity': 20,
        'customer.arrival_interval': 60,
        'customer.num_items.mu': 50,
        'customer.num_items.sigma': 10,
        'customer.time_per_item': 30.0,
        'grocery.num_baggers': 1,
        'grocery.num_lanes': 2,
        'sim.db.enable': True,
        'sim.db.persist': False,
        'sim.dot.colorscheme': 'blues5',
        'sim.dot.enable': True,
        'sim.duration': '7200 s',
        'sim.gtkw.file': 'sim.gtkw',
        'sim.gtkw.live': False,
        'sim.log.enable': True,
        'sim.progress.enable': False,
        'sim.result.file': 'result.json',
        'sim.seed': 1234,
        'sim.timescale': 's',
        'sim.vcd.dump_file': 'sim.vcd',
        'sim.vcd.enable': True,
```

(continues on next page)

(continued from previous page)

```
        'sim.vcd.persist': False,
        'sim.workspace': 'workspace',
    }

    parser = ArgumentParser()
    parser.add_argument(
        '--set',
        '-s',
        nargs=2,
        metavar=('KEY', 'VALUE'),
        action='append',
        default=[],
        dest='config_overrides',
        help='Override config KEY with VALUE expression',
    )
    parser.add_argument(
        '--factor',
        '-f',
        nargs=2,
        metavar=('KEYS', 'VALUES'),
        action='append',
        default=[],
        dest='factors',
        help='Add multi-factor VALUES for KEY(S)',
    )
    args = parser.parse_args()
    apply_user_overrides(config, args.config_overrides)
    factors = parse_user_factors(config, args.factors)
    if factors:
        simulate_factors(config, factors, Top)
    else:
        simulate(config, Top)
```

Running the simulation with the default configuration produces the following `result.json`:

```
{
  "checkout_time_avg": 354.3239645800326,
  "checkout_time_max": 895.4542368592038,
  "checkout_time_min": 90.43561995182836,
  "config": {
    "bagger.bag_time": 1.5,
    "bagger.policy": "float-aggressive",
    "cashier.bag_time": 2.0,
    "cashier.scan_time": 2.0,
    "checkout.bag_area_capacity": 15,
    "checkout.feed_capacity": 20,
    "customer.arrival_interval": 60,
    "customer.num_items.mu": 50,
    "customer.num_items.sigma": 10,
    "customer.time_per_item": 30.0,
    "grocery.num_baggers": 1,
    "grocery.num_lanes": 2,
    "meta.sim.workspace": "workspace",
    "sim.config.file": null,
    "sim.db.enable": true,
    "sim.db.exclude_pat": [],
    "sim.db.file": "sim.sqlite",
```

(continues on next page)

(continued from previous page)


```
    "sim.db.include_pat": [
      ".*"
    ],
    "sim.db.persist": false,
    "sim.db.trace_table": "trace",
    "sim.dot.all.file": "all.dot",
    "sim.dot.colorscheme": "blues5",
    "sim.dot.conn.file": "conn.dot",
    "sim.dot.enable": true,
    "sim.dot.hier.file": "hier.dot",
    "sim.duration": "7200 s",
    "sim.gtkw.file": "sim.gtkw",
    "sim.gtkw.live": false,
    "sim.log.buffering": -1,
    "sim.log.enable": true,
    "sim.log.exclude_pat": [],
    "sim.log.file": "sim.log",
    "sim.log.format": "{level:7} {ts:.3f} {ts_unit}: {scope}:",
    "sim.log.include_pat": [
      ".*"
    ],
    "sim.log.level": "INFO",
    "sim.log.persist": true,
    "sim.progress.enable": false,
    "sim.progress.max_width": null,
    "sim.progress.update_period": "1 s",
    "sim.result.file": "result.json",
    "sim.seed": 1234,
    "sim.timescale": "s",
    "sim.vcd.check_values": true,
    "sim.vcd.dump_file": "sim.vcd",
    "sim.vcd.enable": true,
    "sim.vcd.exclude_pat": [],
    "sim.vcd.include_pat": [
      ".*"
    ],
    "sim.vcd.persist": false,
    "sim.vcd.start_time": "",
    "sim.vcd.stop_time": "",
    "sim.workspace": "workspace",
    "sim.workspace.overwrite": false
  },
  "customers_per_hour": 43.5,
  "customers_total": 87,
  "sim.exception": null,
  "sim.now": 7200.0,
  "sim.runtime": 0.503598068957217,
  "sim.time": 7200
}
```

CHAPTER 3

History

Desmod development began in early 2016 as an internal project at SanDisk Corporation (now Western Digital) out of a desire to improve the pace of model development for architectural exploration and performance estimation of solid state storage systems. Using Python and SimPy for rapid model iteration proved to be a great improvement over SystemC based strategies.

However, although SimPy is a solid foundation for discrete event simulation, building a complete model demanded solutions for other problems such as configuration, command line interface, model organization, monitoring, logging, capturing results, and more. Desmod was written to fill those gaps.

Desmod was released as Free Software under the terms of the MIT License in July, 2016.

CHAPTER 4

Changelog

4.1 desmod-0.6.1 (2020-04-16)

- [FIX] Pool when_not_full and when_not_empty broken epsilon
- [FIX] Typing for SimEnvironment.time()
- [FIX] Typing for __exit__() methods

4.2 desmod-0.6.0 (2020-04-07)

- [BREAK] Drop support for Python < 3.6
- [NEW] Inline type annotations
- [FIX] Use yaml.safe_load() in tests

4.3 desmod-0.5.6 (2019-02-12)

- [NEW] PriorityPool for prioritized get/put requests
- [NEW] Queue.when_at_most() and when_at_least() events (#18)
- [NEW] Pool.when_at_most() and when_at_least() events (#18)
- [CHANGE] Remove Queue.when_new() event
- [CHANGE] Gas station example uses Pool/Pool.when_at_most() (#18)
- [FIX] Add API docs for desmod.pool

4.4 desmod-0.5.5 (2018-12-19)

- [NEW] Add Queue.when_not_full() and Pool.when_not_full()
- [NEW] Context manager protocol for Queue and Pool
- [CHANGE] Pool checks validity of get/put amounts
- [CHANGE] Pool getters/putters are not strictly FIFO
- [CHANGE] __repr__() for Queue and Pool
- [FIX] Pool no longer allows capacity to be exceeded
- [FIX] Pool and Queue trigger all getters and putters
- [FIX] Pool and Queue trigger from callbacks
- [FIX] Repair deprecated import from collections
- [FIX] Various Pool docstrings
- [FIX] Complete unit test coverage for Queue and Pool

4.5 desmod-0.5.4 (2018-08-20)

- [NEW] Add desmod.pool.Pool for modeling pool of resources

4.6 desmod-0.5.3 (2018-05-25)

- [FIX] Repair silent truncation of config override
- [CHANGE] Update dev requirements
- [CHANGE] Do not use bare except
- [CHANGE] Modernize travis-ci config

4.7 desmod-0.5.2 (2017-09-08)

- [FIX] Join worker processes in simulate_many()
- [FIX] Ensure PriorityQueue's items are heapified

4.8 desmod-0.5.1 (2017-04-27)

- [NEW] Add config_filter param to simulate_factors() (#14)
- [FIX] Use pyenv for travis builds

4.9 desmod-0.5.0 (2017-04-27)

- [NEW] Add desmod.dot.generate_dot()
- [NEW] Add "persist" option for tracers
- [NEW] Add SQLiteTracer
- [NEW] Add grocery store example
- [NEW] Support probing a Resource's queue
- [FIX] Stable sort order in DOT generation
- [CHANGE] Rearrange doc index page
- [CHANGE] Change examples hierarchy
- [CHANGE] Add DOT to Gas Station example
- [CHANGE] Tests and cleanup for desmod.probe

4.10 desmod-0.4.0 (2017-03-20)

- [CHANGE] meta.sim.index and meta.sim.special
- [CHANGE] Add meta.sim.workspace
- [FIX] Check simulate_many() jobs
- [CHANGE] Add named configuration categories and doc strings

4.11 desmod-0.3.3 (2017-02-28)

- [CHANGE] Make NamedManager.name() deps argument optional
- [FIX] Add test for desmod.config.parse_user_factors()
- [FIX] More testing for tracer.py

4.12 desmod-0.3.2 (2017-02-24)

- [FIX] Documentation repairs for desmod.config
- [FIX] Add tests for sim.config.file
- [FIX] Annotate no coverage line in test_dot.py
- [NEW] Add desmod.config.apply_user_config()
- [NEW] Support dumping JSON or Python config and result

4.13 desmod-0.3.1 (2017-02-10)

- [NEW] Add sim.vcd.start_time and sim.vcd.stop_time
- [NEW] Add unit tests for desmod.tracer
- [NEW] Dump configuration to file in workspace
- [NEW] Add unit tests for desmod.dot
- [FIX] Use component scope instead of id() for DOT nodes
- [NEW] Colored component hierarchy in DOT
- [FIX] Repair typo in fuzzy_match() exception

4.14 desmod-0.3.0 (2017-01-23)

- [CHANGE] Overhaul progress display
- [NEW] Flexible control of simulation stop criteria
- [FIX] Support progress notification on spawned processes
- [FIX] Remove dead path in test_simulation.py
- [FIX] Various doc repairs to SimEnvironment
- [CHANGE] Add t parameter to SimEnvironment.time()
- [CHANGE Parse unit in SimEnvironment.time()
- [NEW] Add desmod.config.fuzzy_match()
- [REMOVE] Remove desmod.config.short_special()
- [NEW] Add coveralls to travis test suite
- [NEW] Add flush() to tracing subsystem
- [CHANGE] Do not use tox with travis
- [NEW] Add Python 3.6 support in travis
- [FIX] Repair gas_station.py for Python 2

4.15 desmod-0.2.0 (2016-10-25)

- [CHANGE] simulate_factors() now has factors parameter
- [NEW] simulate() can suppress exceptions
- [FIX] simulate_factors() respects sim.workspace.overwrite
- [CHANGE] Update config with missing defaults at runtime

4.16 desmod-0.1.6 (2016-10-25)

- [NEW] Add env.time() and 'sim.now' result
- [FIX] Enter workspace directory before instantiating env
- [CHANGE] Use yaml.safe_dump()
- [FIX] Add dist to .gitignore
- [FIX] Squash warning in setup.cfg

4.17 desmod-0.1.5 (2016-10-17)

- [NEW] Add Queue.size and Queue.remaining properties (#9)
- [NEW] Trace Queue's remaining capacity (#10)
- [NEW] Add Queue.when_new() event (#11)

4.18 desmod-0.1.4 (2016-09-21)

- [NEW] Add desmod.simulation.simulate_many()
- [FIX] Repair various docstring typos
- [FIX] Disable progress bar for simulate_factors() on Windows
- [NEW] Add CHANGELOG.txt to long description in setup.py

4.19 desmod-0.1.3 (2016-07-28)

- [NEW] Cancelable Queue events
- [CHANGE] Connection errors now raise ConnectError
- [FIX] Update pytest-flake8 and flake8 dependencies (yet again)

4.20 desmod-0.1.2 (2016-07-26)

- [NEW] Add "sim.log.buffering" configuration
- [FIX] Repair unit tests (pytest-flake8 dependency)
- [NEW] New optional *Queue.name* attribute
- [FIX] Use *repr()* for exception string in result dict

4.21 desmod-0.1.1 (2016-07-14)

- [FIX] Using ‘True’ and ‘False’ in expressions from the command line
- [CHANGE] Improve simulation workspace handling (sim.workspace.overwrite)
- [CHANGE] Make some ‘sim.xxx’ configuration keys optional
- [NEW] Gas Station example in docs
- [NEW] Add this CHANGELOG.rst and History page in docs

4.22 desmod-0.1.0 (2016-07-06)

- Initial public release

CHAPTER 5

desmod

The desmod package provides a pythonic environment for composing Discrete Event Simulation MODels. The excellent SimPy package provides the discrete event simulation kernel. Desmod provides additional capabilities useful for composing, monitoring, configuring, and simulating arbitrarily complex models.

5.1 Installation

Desmod is available on PyPI and can be installed with *pip*:

```
pip install desmod
```

5.2 Resources

- Documentation on ReadTheDocs
- Questions and development discussion Google Group
- Source code, issue tracker, and CI on GitHub
- Package on PyPI

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

L

N

P

Q

R

S

T

W