# libdw.sm module

Classes for representing and combining state machines.

*class* `libdw.sm.`**`Cascade`**(*m1*, *m2*, *name=None*)

Bases: **`libdw.sm.SM`**

Cascade composition of two state machines. The output of `sm1`is the input to `sm2`

**`done`**(*state*)

**`getNextValues`**(*state*, *inp*)

**`printDebugInfo`**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**`startState`**()

*class* `libdw.sm.`**`Constant`**(*c*)

Bases: **`libdw.sm.SM`**

Machine whose output is a constant, independent of the input

**`getNextState`**(*state*, *inp*)

*class* `libdw.sm.`**`DebugParams`**(*traceTasks*, *verbose*, *compact*, *printInput*)

Housekeeping stuff

`libdw.sm.`**`Delay`**

Delay is another name for the class R, for backward compatibility

alias of **`R`**

*class* `libdw.sm.`**`Feedback`**(*m*, *name=None*)

Bases: **`libdw.sm.SM`**

Take the output of `m` and feed it back to its input. Resulting machine has no input. The output of `m` **must not** depend on its input without a delay.

**`done`**(*state*)

**`getNextValues`**(*state*, *inp*)

Ignores input.

**`printDebugInfo`**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**`startState`**()

*class* `libdw.sm.`**`Feedback2`**(*m*, *name=None*)

Bases: **`libdw.sm.Feedback`**

Like previous `Feedback`, but takes a machine with two inps and one output at initialization time. Feeds the output back to the second inp. Result is a machine with a single inp and single output.

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

*class* `libdw.sm.`**FeedbackAdd**(*m1*, *m2*, *name=None*)

Bases: **libdw.sm.SM**

Takes two machines, m1 and m2. Output of the composite machine is the output to m1. Output of m1 is fed back through m2; that result is added to the input and used as the 'error' signal, which is the input to m1.

**done**(*state*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState**()

*class* `libdw.sm.`**FeedbackSubtract**(*m1*, *m2*, *name=None*)

Bases: **libdw.sm.SM**

Takes two machines, m1 and m2. Output of the composite machine is the output to m1. Output of m1 is fed back through m2; that result is subtracted from the input and used as the 'error' signal, which is the input to m1. Transformation is the one described by Black's formula.

**done**(*state*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState**()

*class* `libdw.sm.`**Gain**(*k*)

Bases: **libdw.sm.SM**

Machine whose output is the input, but multiplied by k. Specify k in initializer.

**getNextValues**(*state*, *inp*)

*class* `libdw.sm.`**If**(*condition*, *sm1*, *sm2*, *name=None*)

Bases: **libdw.sm.SM**

Given a condition (function from inps to boolean) and two state machines, make a new machine. The condition is evaluated at start time, and one machine is selected, permanently, for execution.

Rarely useful.

**done**(*state*)

**getFirstRealState**(*inp*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState** = *('start', None)*

*class* `libdw.sm.`**Mux**(*condition*, *sm1*, *sm2*, *name=None*)

Bases: **`libdw.sm.Switch`**

Like `Switch`, but updates both machines no matter whether the condition is true or false. Condition is only used to decide which output to generate. If the condition is true, it generates the output from the first machine, otherwise, from the second.

**getNextValues**(*state*, *inp*)

*class* `libdw.sm.`**Parallel**(*m1*, *m2*, *name=None*)

Bases: **`libdw.sm.SM`**

Takes a single inp and feeds it to two machines in parallel. Output of the composite machine is the pair of outputs of the two individual machines.

**done**(*state*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState**()

*class* `libdw.sm.`**Parallel2**(*m1*, *m2*)

Bases: **`libdw.sm.Parallel`**

Like `Parallel`, but takes two inps. Output of the composite machine is the pair of outputs of the two individual machines.

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

*class* `libdw.sm.`**ParallelAdd**(*m1*, *m2*, *name=None*)

Bases: **`libdw.sm.Parallel`**

Like `Parallel`, but output is the sum of the outputs of the two machines.

**getNextValues**(*state*, *inp*)

*class* `libdw.sm.`**PureFunction**(*f*)

Bases: **`libdw.sm.SM`**

Machine whose output is produced by applying a specified Python function to its input.

**getNextValues**(*state*, *inp*)

*class* libdw.sm.**R**(*v0=0*)

Bases: **libdw.sm.SM**

Machine whose output is the input, but delayed by one time step. Specify initial output in initializer.

**getNextValues**(*state*, *inp*)

**startState** = *None*

State is the previous input

*class* libdw.sm.**Repeat**(*sm*, *n=None*, *name=None*)

Bases: **libdw.sm.SM**

Given a terminating state machine, generate a new one that will execute it n times. If n is unspecified, it will repeat forever.

**advanceIfDone**(*counter*, *smState*)

**done**(*state*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState**()

*class* libdw.sm.**RepeatUntil**(*condition*, *sm*, *name=None*)

Bases: **libdw.sm.SM**

Given a terminating state machine and a condition on the input, generate a new one that will run the machine until the condition becomes true. However, the condition is **only**evaluated when the sub-machine terminates.

**done**(*state*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState**()

*class* libdw.sm.**SM**

Generic superclass representing state machines. Don't instantiate this: make a subclass with definitions for the following methods:

- `getNextValues:(state_t,inp_t) ->(state_t+1,output_t)` or`getNextState:(state_t,inpt_t) ->state_t+1`

- `startState: state` or `startState()-> state`

optional:

- `done:(state) ->boolean`(defaults to always false)
- `legalInputs:list(inp)`

See State Machines chapter in 6.01 Readings for detailed explanation.

**check**(*thesm*, *inps=None*)

Run a rudimentary check on a state machine, using the list of inputs provided. Makes sure that getNextValues is defined, and that it takes the proper number of input arguments (three: self, start, inp). Also print out the start state, and check that getNextValues provides a legal return value (list of 2 elements: (state,output)). And tries to check if getNextValues is changing either self.state or some other attribute of the state machine instance (it shouldn't: getNextValues should be a pure function).

Raises exception 'InvalidSM' if a problem is found.

| | |
|---|---|
| **Parameters:** | • **thesm** – the state machine instance to check<br>• **inps** – list of inputs to test the state machine on (default None) |
| **Returns:** | none |

**doTraceTasks**(*inp*, *state*, *out*, *debugParams*)

Actually execute the trace tasks. A trace task is a list consisting of three components:

- `name`: is the name of the machine to be traced
- `mode`: is one of `'input'`, `'output'`, or `'state'`
- `fun`: is a function

To **do** a trace task, we call the function `fun` on the specified attribute of the specified mahine. In particular, we execute it right now if its machine name equals the name of this machine.

**done**(*state*)

By default, machines don't terminate

**getNextValues**(*state*, *inp*)

Default version of this method. If a subclass only defines`getNextState`, then we assume that the output of the machine is the same as its next state.

**getStartState**()

Handles the case that self.startState is a function. Necessary for stochastic state machines. Ignore otherwise.

**guaranteeName**()

Makes sure that this instance has a unique name that can be used for tracing.

**isDone**()

Should only be used by transduce. Don't call this.

**legalInputs** = *[]*

By default, the space of legal inputs is not defined.

**name** = *None*

Name used for tracing

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

Default method for printing out all of the debugging information for a primitive machine.

**run**(*n=10*, *verbose=False*, *traceTasks=[]*, *compact=True*, *printInput=True*, *check=False*)

For a machine that doesn't consume input (e.g., one made with`feedback`, for `n` steps or until it terminates.

See documentation for the `start`method for description of the rest of the parameters.

| | |
|---|---|
| **Parameters:** | **n** – number of steps to run |
| **Returns:** | list of outputs |

**start**(*traceTasks=[]*, *verbose=False*, *compact=True*, *printInput=True*)

Call before providing inp to a machine, or to reset it. Sets self.state and arranges things for tracing and debugging.

| | |
|---|---|
| **Parameters:** | • **traceTasks** – list of trace tasks. See documentation for`doTraceTasks`for details<br>• **verbose** – If `True`, print a description of each step of the machine<br>• **compact** – If `True`, then if `verbose =True`, print a one-line description of the step; if `False`, print out the recursive substructure of the state update at each step<br>• **printInput** – If `True`, then if `verbose =True`, print the whole input in each step, otherwise don't. Useful to set to `False`when the input is large and you don't want to see it all. |

**startState** = *None*

By default, startState is none

**step**(*inp*)

Execute one 'step' of the machine, by propagating `inp` through to get a result, then updating `self.state`. Error to call `step` if `done` is true. :param inp: next input to the machine

**transduce**(*inps*, *verbose=False*, *traceTasks=[]*, *compact=True*, *printInput=True*, *check=False*)

Start the machine fresh, and feed a sequence of values into the machine, collecting the sequence of outputs

For debugging, set the optional parameter check = True to (partially) check the representation invariance of the state machine before running it. See the documentation for the `check`method for more information about what is tested.

See documentation for the `start`method for description of the rest of the parameters.

| | |
|---|---|
| **Parameters:** | **inps** – list of inputs appropriate for this state machine |
| **Returns:** | list of outputs |

**transduceF**(*inpFn*, *n=10*, *verbose=False*, *traceTasks=[]*, *compact=True*, *printInput=True*)

Like `transduce`, but rather than getting inputs from a list of values, get them by calling a function with the input index as the argument.

*class* **libdw.sm.Select**(*k*)

Bases: **libdw.sm.SM**

Machine whose input is a structure list and whose output is the `k` th element of that list.

**getNextState**(*state*, *inp*)

*class* **libdw.sm.Sequence**(*smList*, *name=None*)

Bases: **libdw.sm.SM**

Given a list of state machines, make a new machine that will execute the first until it is done, then execute the second, etc. Assume they all have the same input space.

**advanceIfDone**(*counter*, *smState*)

Internal use only. If that machine is done, start new machines until we get to one that isn't done

**done**(*state*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState**()

*class* **libdw.sm.Switch**(*condition*, *sm1*, *sm2*, *name=None*)

Bases: **libdw.sm.SM**

Given a condition (function from inps to boolean) and two state machines, make a new machine. The condition is evaluated on every step, and the selected machine is used to generate output and has its state updated. If the condition is true, `sm1` is used, and if it is false, `sm2` is used.

**done**(*state*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState**()

*class* **libdw.sm.Until**(*condition*, *sm*, *name=None*)

Bases: **libdw.sm.SM**

Execute SM until it terminates or the condition becomes true. Condition is evaluated on the inp

**done**(*state*)

**getNextValues**(*state*, *inp*)

**printDebugInfo**(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

**startState**()

*class* `libdw.sm.`**`Wire`**

Bases: **`libdw.sm.SM`**

Machine whose output is the input

**getNextValues**(*state*, *inp*)

`libdw.sm.`**`allDefined`**(*struct*)

`libdw.sm.`**`coupledMachine`**(*m1*, *m2*)

Couple two machines together. :param m1: `SM`:param m2: `SM`:returns: New machine with no input, in which the output of `m1` is the input to `m2` and vice versa.

`libdw.sm.`**`isDefined`**(*v*)

`libdw.sm.`**`safe`**(*f*)

`libdw.sm.`**`safeAdd`**(*a1*, *a2*)

`libdw.sm.`**`safeMul`**(*a1*, *a2*)

`libdw.sm.`**`safeSub`**(*a1*, *a2*)

`libdw.sm.`**`splitValue`**(*v*, *n=2*)

If `v` is a list of `n`elements, return it; if it is ‘undefined’, return a list of `n`‘undefined’ values; else generate an error