Purely Event-Driven Programming Python Simulation Documentation

Release 1.0

Bas van den Heuvel

June 07, 2016

Simulates a purely event-driven programming language.

This simulator is part of a computer science bachelor's thesis for the University of Amsterdam, by Bas van den Heuvel. It follows all concepts introduced in this thesis. The purpose is to be able to test and refine those concepts.

The scheduling method is deterministically sequential. No concurrent execution is implemented.

Classes:

- MachineControl: manages and schedules state machines and events
- Event: event for communication between state machines
- StateMachine: superclass for all possible state machines

class `simulator.simulator.`**`Event`**(*typ*, *emitter*, *value=None*, *destination=None*, *ack=False*)
An event for interaction between state machines.

`__init__`(*typ*, *emitter*, *value=None*, *destination=None*, *ack=False*)
Initialize the event.

Parameters

- **`typ`** – the event's type string
- **`emitter`** – the StateMachine emitting the event

Keyword Arguments

- **value** – value to transmit (default None)
- **destination** – the StateMachine the event should end up with (default None)
- **ack** – whether the receiving machine should emit an acknowledgement (default False)

class `simulator.simulator.`**`MachineControl`**(*debug=False*, *step=False*)
Manage and schedule state machines and events.

Every program created with this simulator should have an instance of this class. Starting a program goes through this instance, as well as instantiating state machines and sending events.

Execution of its machine's states happens through cycles. The scheduling for this is sequential and very minimalistic: the first machine in its queue gets cycled after which it gets replaced at the end of the queue.

It can be said that there is no event scheduling. Before each state cycle, all events in the event buss are distributed to their respective state machines.

`__init__`(*debug=False*, *step=False*)
Initialize a machine control.

It setups up a machine list, which in this implementation is a queue. Reaction maps are created as dictionaries, and the event buss is another queue.

The *ctx* variable is not yet set. This happens when the simulator is started.

Keyword Arguments

- **debug** – opens a window for each state machine showing state and event information if True (default True)
- **step** – allows one to cycle stepwise (default False)

`add_event_reaction`(*typ*, *reactor*, *state*)
Add a reaction to an event.

Parameters

- **`typ`** – the event's type string

- **reactor** – the StateMachine that should react
- **state** – the state the machine should transition to, a method

add_machine_reaction(*typ*, *emitter*, *reactor*, *state*)
Add a reaction to a state machine's event.

Parameters

- **typ** – the event's type string
- **emitter** – the event's emitting StateMachine
- **reactor** – the StateMachine that should react
- **state** – the state the machine should transition to, a method

cycle()
Distribute events, cycle a machine and return whether any are left.

When the machine queue is empty, False is returned. Otherwise, True is returned.

If debuggin is on, the cycles machine's state before and after the cycle is shown in the machine's debuggin window, accompanied by any variables indicated in the machine. If these variables have changed after the cycle, the changed values are shown as well.

debug_aftercycle(*machine*, *p_state*, *n_state*, *p_var_str*)
Send info to a machine's debug window after a cycle.

First the machine's variables are compared to its variables before the cycle. If they are changed, they are displayed. If the machine was in its listen state and reacted to an event, this event is displayed. Finally, if the cycle resulted in a state transition, the new state is displayed.

If the machine cycled its halt state, nothing is to be done.

Parameters

- **machine** – the StateMachine to show debug information for
- **p_state** – the machine's state before the cycle
- **n_state** – the machine's state after the cycle
- **p_var_str** – the machine's variable string before the cycle

debug_precycle(*machine*)
Send info to a machine's debug window before a cycle.

First the machine's current state is show. After this, if the machine has indicitated any variables as information, these are shown as well.

Parameters **machine** – the StateMachine to show debug information for

distribute_events()
Distribute an event to machines and return whether any are left.

If an event has a destination and that destination is still alive (i.e. not halted), the event is put into that machine's inbox. Otherwise, the event is put into the inbox of all live machines, except the event's emitter.

If an event has been distributed, True is returned. Otherwise, False is returned.

emit(*event*)
Add an event to the event buss.

If debugging is on, the event is displayed in the emitter's debug window.

Parameters **event** – the to be emitted Event

filter_event(*machine*, *event*)
Returns a state if a machine should react to an event.

If a reaction exists, the machine's reaction state is returned. Otherwise, None is returned.

First machine reactions is checked, because such reactions are more specific and thus have priority.

Parameters

- **machine** – the reacting state machine, a StateMachine
- **event** – the event to be checked, an Event

halt(*machine*)
Halt a machine.

If debuggin is on, the machine's debuggin window's title is altered to include 'HALTED' and the window's stdin pipe is closed.

remove_event_reaction(*typ*, *reactor*)
Remove a reaction to an event.

Parameters

- **typ** – the event's type string
- **reactor** – the StateMachine that should ignore the event

remove_machine_reaction(*typ*, *emitter*, *reactor*)
Remove a reaction to a state machine's event.

Parameters

- **typ** – the event's type string
- **emitter** – the event's emitting StateMachine
- **reactor** – the StateMachine that should ignore the event

reset()
Reset machine control.

This prepares it for a next run. Python garbage collects itself, but in an actual implementation, all these fields need to be emptied carefully.

run(*machine_cls*, **args*, ***kwargs*)
Start a state machine and cycle until all machines have halted.

A context is created for this first machine, by instantiating the StateMachine superclass without a context.

Parameters

- **machine_cls** – a StateMachine subclass
- ***args/**kwargs** – any arguments the state machine takes

start_machine(*machine_cls*, *ctx*, **args*, ***kwargs*)
Start a state machine.

Initializes a machine, given arbitrary arguments, and adds it to the machine queue. After this, event reaction to 'halt' is added.

If debugging is turned on, this also starts a debug window, able to show state and event information.

Parameters

- **machine_cls** – a StateMachine subclass

- **ctx** – the state machine that starts this new machine
- ***args/**kwargs** – any arguments the state machine takes

class simulator.simulator.**StateMachine**(*ctl*, *ctx*)
Represent a state machine.

To create purely event-driven programs, one can subclass this class. The *__init__* method should be extended with local variables and an initial state, but first call *super().__init__()* to prepare the machine.

States can be implemented by adding methods to the class. The initial state can be indicated by setting *self.init_state* to the preferred state method in *__init__*. Loops are not impossible, but should not be used as they do not exist in the language proposed by this thesis.

Do not override *listen* and *halt*, this will break the simulator. However, referring to both states is no problem (and usually necessary).

The current event can be referred to through *self.event*. Do not mutate this variable.

__init__(*ctl*, *ctx*)
Initialize the state machine.

Prepares the machine for execution and prepares event processing necessities.

Parameters

- **ctl** – a MachineControl instance
- **ctx** – the machine's context, a StateMachine

cycle()
Run the current state and determine the next.

If no next state is obtained, the new state will be 'listen'.

emit(*typ*, *value=None*)
Emit an event.

Parameters **typ** – the event's type string

Keyword Arguments **value** – value to transmit with the event (default None)

emit_to(*destination*, *typ*, *value=None*, *ack_state=None*)
Emit an event to a machine.

Parameters

- **destination** – the StateMachine to send the event to
- **typ** – the event's type string

Keyword Arguments

- **value** – value to transmit with the event (default None)
- **ack_state** – a state for acknowledgement, a method

If *ack_state* is given, the receiving machine will send an acknowledgement event. When the emitting machine recieves this event, it will transition to the given state.

filter_event(*event*)
Return a state if a reaction to the event exists.

Parameters **event** – the Event

halt()
Halt state for all machines.

Do not extend or override this method.

First emits 'halt', which halts all child machines. Then MachineControl is told to halt the machine.

ignore_when(*typ*)
Remove an event reaction.

Besides ignoring further such events, all events from the given machine and of the given type in the machine's inbox are removed.

Parameters **type** – the event's type string

ignore_when_machine_emits(*typ*, *machine*)
Remove a machine event reaction.

Besides ignoring further such events, all events from the given machine and of the given type in the machine's inbox are removed.

Parameters

- **typ** – the event's type string
- **machine** – the event's emitting StateMachine

listen()
Listen state for all machines.

Do not extend or override this method.

Checks the event inbox for any events and possible reactions. If an acknowledgement is required, this is sent.

start_machine(*machine_cls*, **args*, ***kwargs*)
Instantiate and start a machine.

Parameters

- **machine_cls** – a StateMachine subclass
- ***args/**kwargs** – any arguments the state machine takes

var_str()
Create a string of formatted variables.

self.info should contain a list of tuples with a format string and the name of a variable. This method aggregates them into a comma-separated string containing these formatted values.

when(*typ*, *state*)
Add an event reaction.

Parameters

- **typ** – the event's type string
- **state** – the state to transition to, a method

when_machine_emits(*typ*, *machine*, *state*)
Add a machine event reaction.

Parameters

- **typ** – the event's type string
- **machine** – the emitting StateMachine

- **state** – the state to transition to, a method

class simulator.debug_window.**DebugWindow**(*title='State Machine'*)
Create a Tk window for displaying debug messages.

The *Window* class in this script is the actual window. By invoking this file directly, such a window is created. This window reads from stdin. This class does exactly that. It runs this script as a subprocess, linking its stdin to a writable buffer.

__init__(*title='State Machine'*)
Initialize a debug window.

Opens a Tk window in a subprocess, in text-mode which allows the stdin pipe to be used for text directly.

Keyword Arguments **title** – the window's initial title (default 'State Machine')

close()
Close the window's stdin pipe.

This does not actually close the window, only the stream. The window is kept open so the user can analyse states even after a program is finished.

set_title(*title*)
Set the window's title.

Parameters **title** – the title

write(*text*)
Write a line to the window.

The window might have been closed by the user or some different event. This is ignored.

Parameters **text** – text excluding newline

class simulator.debug_window.**Window**
Show a Tk window with scrollable text from stdin.

Checks stdin for a new line every one millisecond. If the line starts with a '#', the rest of the line is used as a new title for the window. Otherwise, the line is appended to the textfield, including the newline character.

__init__()
Initialize the window.

Creates a frame, holding a srollable textfield. Finally reading from stdin is initiated.

do_read()
Try to read a line from stdin.

process_line(*line*)
Process a line for debug display.

If a line starts with '#', change the window's title. Otherwise, write the line to the textbox.

Parameters **line** – the line to be processed, including newline character

write_text(*text*)
Write text to the end of the textfield.

Parameters **text** – the text to be added to the textfield.

simulator.debug_window.**main**()
Make stdin nonblocking and open a window.

simulator.debug_window.**make_nonblocking**(*fh*)
Make a file nonblocking.

fcntl is a C system call, used to modify file descriptors. The operation used (F_SETFL) sets the file descriptor's flags.

The argument to this function call uses F_GETFL, which gets the currently set flags. These are combined with a new flag: O_NONBLOCK. This flag makes sure no calls to the file cause the process to wait, i.e. nonblocking.

S

Symbols

A

C

D

E

F

H

I

L

M

P

R

S

V

W