Workshop: The State of Things

Christopher R. Genovese

Department of Statistics & Data Science

Tue 16 Sep 2025
Session #7

Plan

Workshop on OOP: State Machines and the Builder Pattern

Announcements

- My Office Hours today at 4pm and by appointment.
- **Homework**:
  - **migit** assignment Tasks #1–#6 due Tue 16 Sep. Available on github problem bank.
  - **migit** assignment Tasks #1–#10 due Tue 23 Sep. Available on github problem bank.

Goals for Today

Last time, we looked at "Propositions as Types" and used it to analyze and generalize Binary Search.

Today's goal is continue our exploration of object-oriented programming with a practical task that uses these ideas. Along the way, we will see the "builder" pattern.

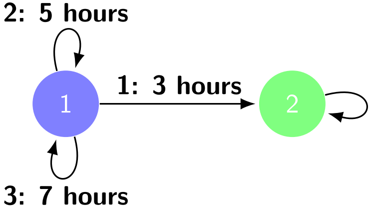
Plan

Workshop on OOP: State Machines and the Builder Pattern

(Finite) State Machines

A **finite state machine** (FSM) is a system that can at any time be in one of a finite number of possible *states*.

The system evolves in steps making *transitions* from one state to another.

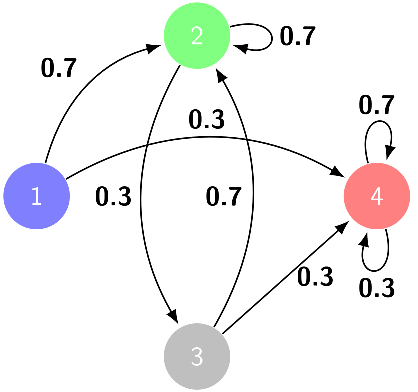


(Finite) State Machines

A **finite state machine** (FSM) is a system that can at any time be in one of a finite number of possible *states*.

The system evolves in steps making *transitions* from one state to another.

Here is a FSM in which these transitions are random. (Look familiar?)



State Machines

A **state machine** is an abstraction that represents a collection of possible states, allowed transitions between particular states, actions performed at various points during transitions, and events that can be dispatched to initiate changes/actions.

Our goal is to build an object-oriented description of state machines, with two key classes:

- `StateMachine` – class that represents a state machine, it can dispatch events, execute actions, and track the corresponding state.
- `StateMachineBuilder` – class provides methods for building a state machine while ensuring validity.

See the `state-machines` assignment in the problem bank.

State Machines (cont'd)

State machines require the specification of several elements:

- **States**. A finite set of arbitrary "symbols" representing distinct states, or internal configurations, of the system.

State Machines (cont'd)

State machines require the specification of several elements:

- **States**. A finite set of arbitrary "symbols" representing distinct states, or internal configurations, of the system.
- **Transitions**. Transitions are the allowed moves from one state to another, where both source and target state may be the same. There may be more than one transition from source state to target state.

State Machines (cont'd)

State machines require the specification of several elements:

- **States**. A finite set of arbitrary "symbols" representing distinct states, or internal configurations, of the system.
- **Transitions**. Transitions are the allowed moves from one state to another, where both source and target state may be the same. There may be more than one transition from source state to target state.
- **Events**. Transitions indicate which moves between states are allowed, but our state machine also needs to know when to take them. Events are named sets of transitions and the conditions or data that trigger them. The name is called the event **type**; it determines which transitions are taken when an event of that type is dispatched to the machine. For convenience, we use the same namespace for transitions and events, so a named transition corresponds to an event that is a singleton set.

  When events are dispatched to a machine, they come with a type and with an optional *payload*, data of a structure determined by the type, to be used by *actions*.

State Machines (cont'd)

State machines require the specification of several elements:

- **Guards and Selectors**. If we are in state s and the machine handles an event of type t where t contains a transition with source state s, then in principle the machine would make that transition. But there are two nuances.
  - A *guard* is a predicate *associated with a transition* that takes the event (type and payload) and returns true when the transition should be taken.
  - A *selector* is a function *associated with an event* that selects one among the possible transitions to take.

  Selectors run before guards.

State Machines (cont'd)

State machines require the specification of several elements:

- **Guards and Selectors**. If we are in state s and the machine handles an event of type t where t contains a transition with source state s, then in principle the machine would make that transition. But there are two nuances.
  - A *guard* is a predicate *associated with a transition* that takes the event (type and payload) and returns true when the transition should be taken.
  - A *selector* is a function *associated with an event* that selects one among the possible transitions to take.

  Selectors run before guards.

- **Actions**. Actions are functions that run at particular parts of the transition. They can produce output or other effects or track/manipulate internal data. We consider five types of actions, based on when the actions are executed:

  1. *before* a transition,
  2. on *exit* from a state,
  3. *during* a transition,
  4. on *enter*ing a state, and
  5. *after* a transition.

Specifying State Machines

We specify a state machine by defining states, transitions, events, actions, along with guards and selectors if appropriate.

We will do this using the `StateMachineBuilder` class. So for instance, this might look like

```
machine = (
    StateMachineBuilder()
    .add_states('a', 'b', 'c', 'd')
    .add_transition('a', 'b')       # Name defaults to 'a -> b'
    .add_transition('c', 'd', name='foo')
    .add_action('enter', 'c', on_c_func)
        #...
    .build()
)
```

or in R

```
machine = StateMachineBuilder() |>
            add_states('a', 'b', 'c', 'd') |>
            add_transition('a', 'b') |>       # Name defaults to 'a -> b'
            add_transition('c', 'd', name='foo') |>
            add_action('enter', 'c', on_c_func)  |>
                #... |>
            build()
```

The `build` method validates the specification and creates the `StateMachine` object. You should design the Builder interface to be clear and easy to use but flexible enough for reasonable

Specifying State Machines

We specify a state machine by defining states, transitions, events, actions, along with guards and selectors if appropriate.

More generally, we can create a *Domain Specific Language* (DSL) for specifying a state machine:

```
state a
state b

transition one from c to a
transition two from a, b to c
transition three from b to c, d
transition six from b to d or from d to e
delegate input to one

action on enter a
  | ... arbitrary lines
  | ...
end
action during two with event, source, target
  | ...
  | ...
end
```

We can then parse the DSL, using the `StateMachineBuilder` internally. We won't deal with the parsing today.

The Task

Individually or in teams, implement the `StateMachine` and `StateMachineBuilder` classes, and implement an example or two of them in action.

The `StateMachine` constructor should be “private,” with the `StateMachineBuilder` the typical way to build a machine.

The End