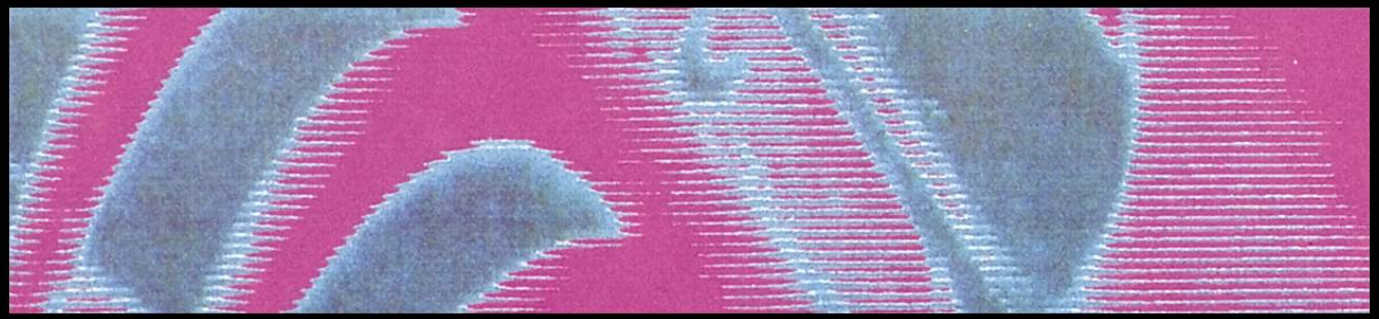
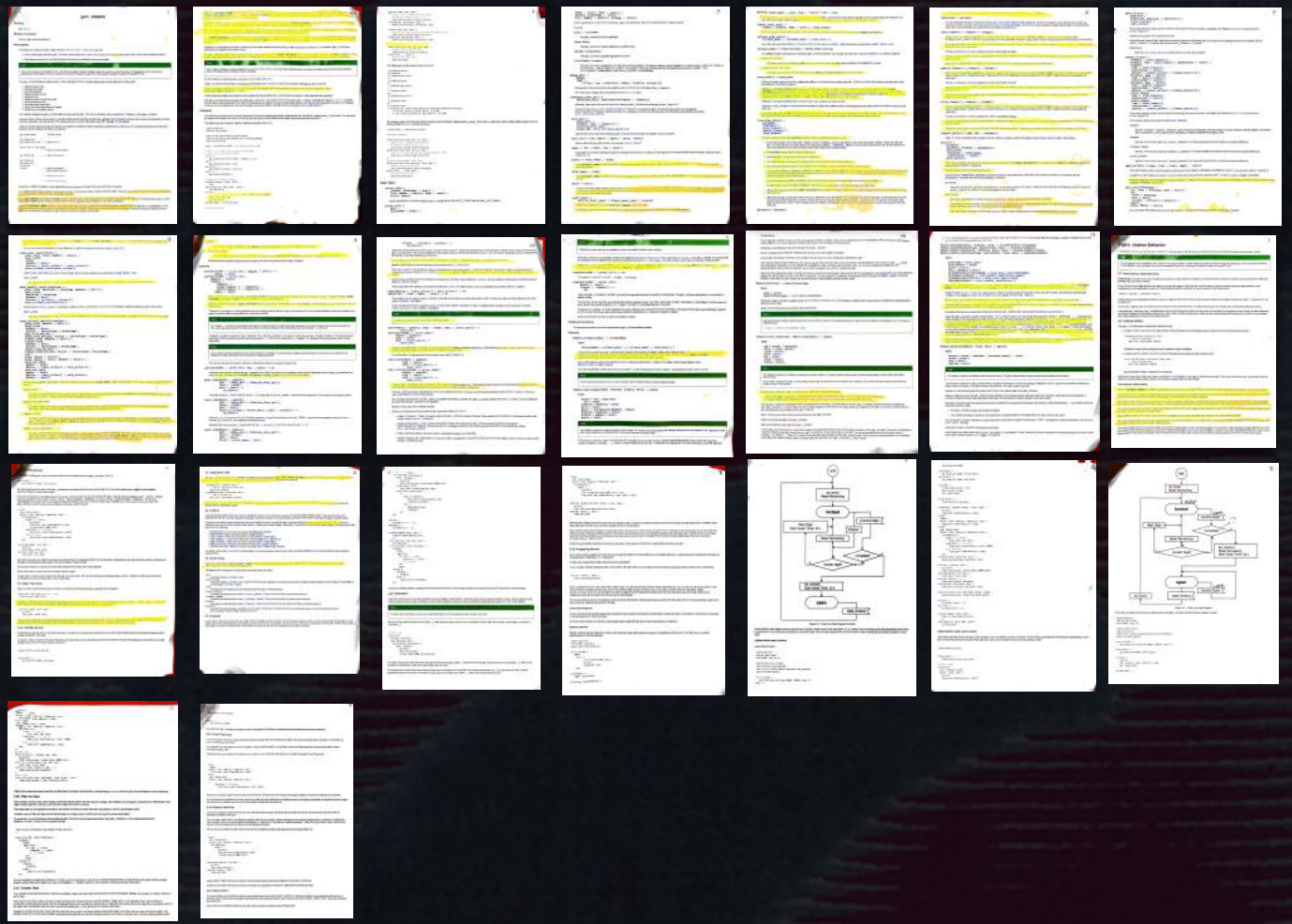
code beam sf 2019



pretty state machine

how it started



think like a telecom switch

jeff smith

@electricshaman

enbala
YOUR GRID. IN BALANCE.

talk objective

encourage adoption of gen_statem in OTP

gen_statem

performance on par with gen_server

(in practice)

gen_state_machine

elixir wrapper

concepts

state

data

event

action

concepts

`:locked`

data

event

action

concepts

```
:locked
%{code: 12345}
```

event

action

concepts

```
:locked
%{code: 12345}
{:button, 5}
```

action

concepts

event type

event content

```
:locked
%{code: 12345}
:cast, {:button, 5}
```

action

concepts

```
:locked
%{code: 12345}
:cast, {:button, 5}
{:reply, from, :ok}
```

multiple actions

```
[
  {:reply, from, :ok},
  {:state_timeout, 10_000},
  {:next_event, :cast, {:button, 5},
  # ...
]
```

getting started

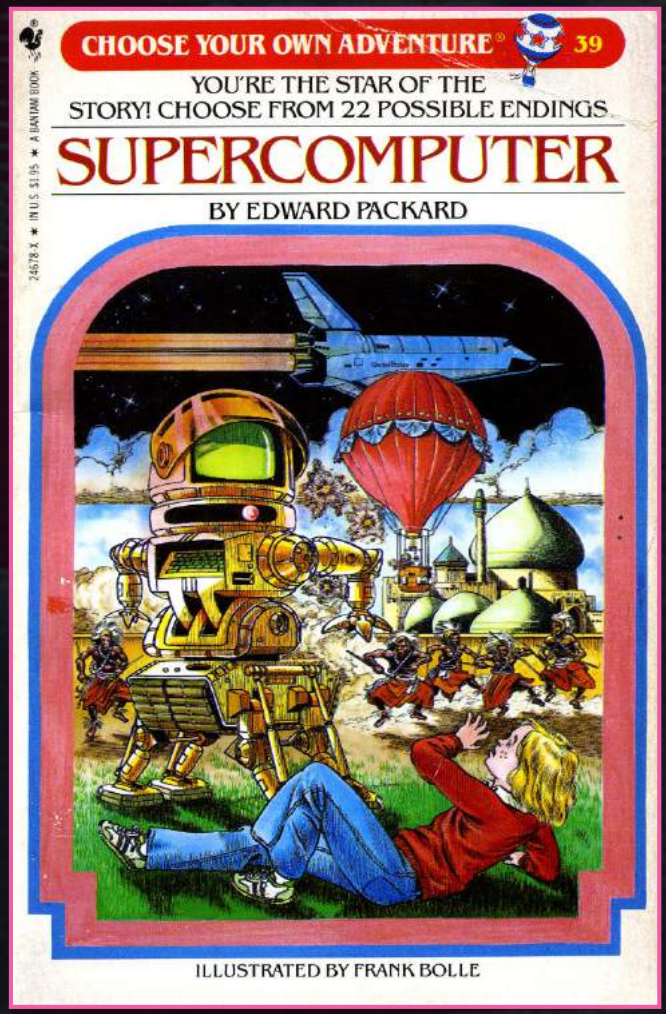
```elixir
def start_link(_opts) do
  GenStateMachine.start_link(__MODULE__, [])
end
```

```elixir
def init(_) do
  data = %{}
  {:ok, :off, data}
end
```

initial state

callback modes

choose your own adventure



the mission

handle all possible combinations of event & state

adventure 1: handle_event_function

"event centered approach"
branch on event then state

all events handled in function
`handle_event/4`

default callback mode

adventure 1: handle_event_function

current state

```
def handle_event(:cast, :flip, :off, data) do
  {:next_state, :on, data + 1}
end
```

adventure 2: state_functions

"state centered approach"

callbacks organized around state

branch on state then event

adventure 2: state_functions

one callback function* per state

`StateName/3`

adventure 2: state_functions

current state

```
def off(:cast, :flip, data) do
  {:next_state, :on, data + 1}
end
```

choose your adventure

```
callback_mode() → state_functions.
```

```
use GenStateMachine,
  callback_mode: :state_functions
```

```
callback_mode() → handle_event_function.
```

```
use GenStateMachine,
  callback_mode: :handle_event_function
```

same behavior
different callbacks

state enter callbacks

```
use GenStateMachine, callback_mode:
[:state_functions, :state_enter]
```

event content: old state

```
def off(:enter, :on, data) do
  # ...
  :keep_state_and_data
end
```

inserted events

Generate events from your own state machine!

```
{:next_event,
 event_type,
 event_content}
```

postpone

I just can’t deal with you right now

emulate selective receive

The `gen_statem` event queue model is sufficient to emulate the normal process message queue with selective receive.

Postponing an event corresponds to not matching it in a receive statement, and changing states corresponds to entering a new receive statement.

emulate selective receive

```
def wait_for_it do
  receive do
    {:ok, {:step, 1}} →
      # ... Do some stuff
      wait_for_it()
  after 0 →
    wait_for_it()
  end
end
```

timeouts

3 types of timeouts

event timeout

cancelled on any message

```
{:timeout, 1_000}
```

state timeout

cancelled when state changes

```
{:state_timeout, 10_000}
```

state timeout in practice

```
{
  :state_timeout,
  30_000,
  {:response_timeout, stream, from}
}
```

state timeout in practice

```
def handle_event(
  :state_timeout,
  {:response_timeout, stream, from},
  :awaiting_response,
  data
) do

  # Do some cleanup stuff ...
  actions = {:reply, from,
{:error, :response_timeout}}
  {:next_state, :ready, data, actions}
end
```

generic timeout

not cancelled for you

```
{
  {:timeout, :name},
  10_000
}
```

welcome to the machine

```
def flip_it() do
 GenStateMachine.cast(__MODULE__, :flip)
end
```

```
def call_it() do
 GenStateMachine.call(__MODULE__, :hello)
end
```

additional resources

http://erlang.org/doc/man/gen_statem.html

http://erlang.org/doc/design_principles/statem.html

https://potatosalad.io/2017/10/13/time-out-elixir-state-machines-versus-servers

https://github.com/erlang/otp/blob/master/lib/stdlib/src/gen_statem.erl