UI Modeling with Statecharts

How to make sense of complex UI logic?

Often UI logic is spread all over component code and difficult to maintain.

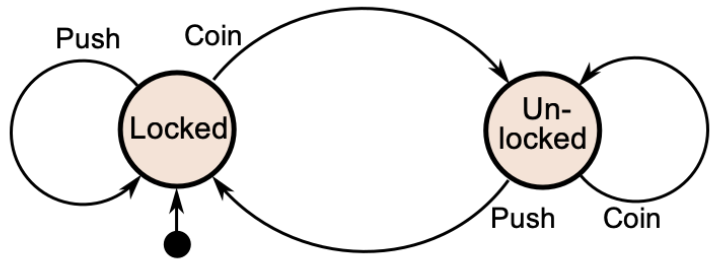
```
// ...
const [loading, setLoading] = useState(false);
const [uploadError, setUploadError] = useState(false);

const onDrop = async files => {
  setLoading(true);
  setUploadError(false);
  try {
    const values = await upload(files);
    setLoading(false);
    onChange(values);
  } catch (e) {
    setLoading(false);
    setUploadError(true);
    setTimeout(() => {
      setUploadError(false);
    }, 2000);
  }
};
// ...
```

We can do better...

State machine is an abstract machine that can be in exactly one state at a time

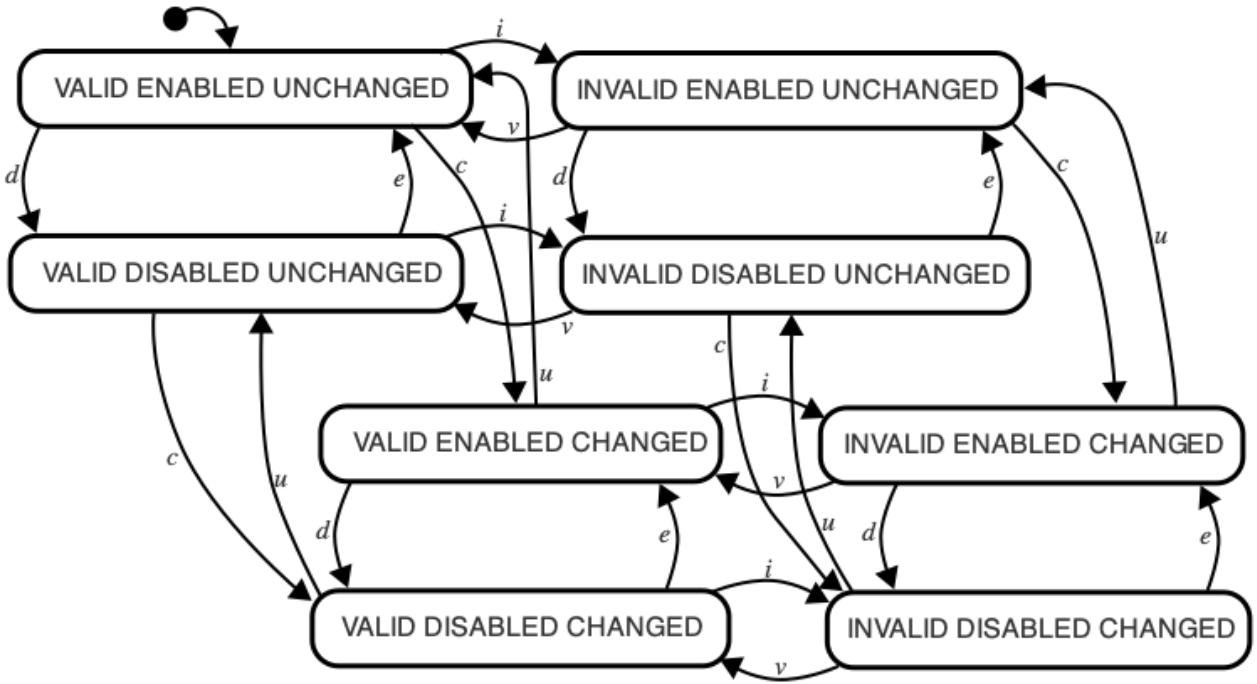
- States
- Events
- Transitions



A state machine diagram for a turnstile

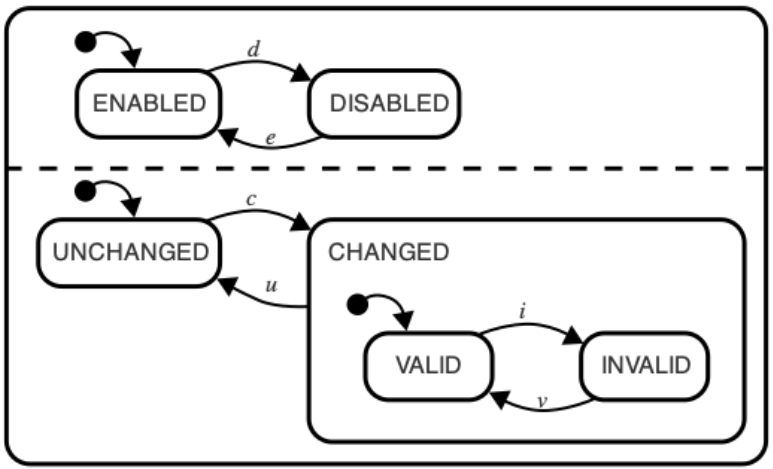
https://en.wikipedia.org/wiki/Finite-state_machine#/media/File:Turnstile_state_machine_colored.svg

State machines don’t scale well. (state explosion)



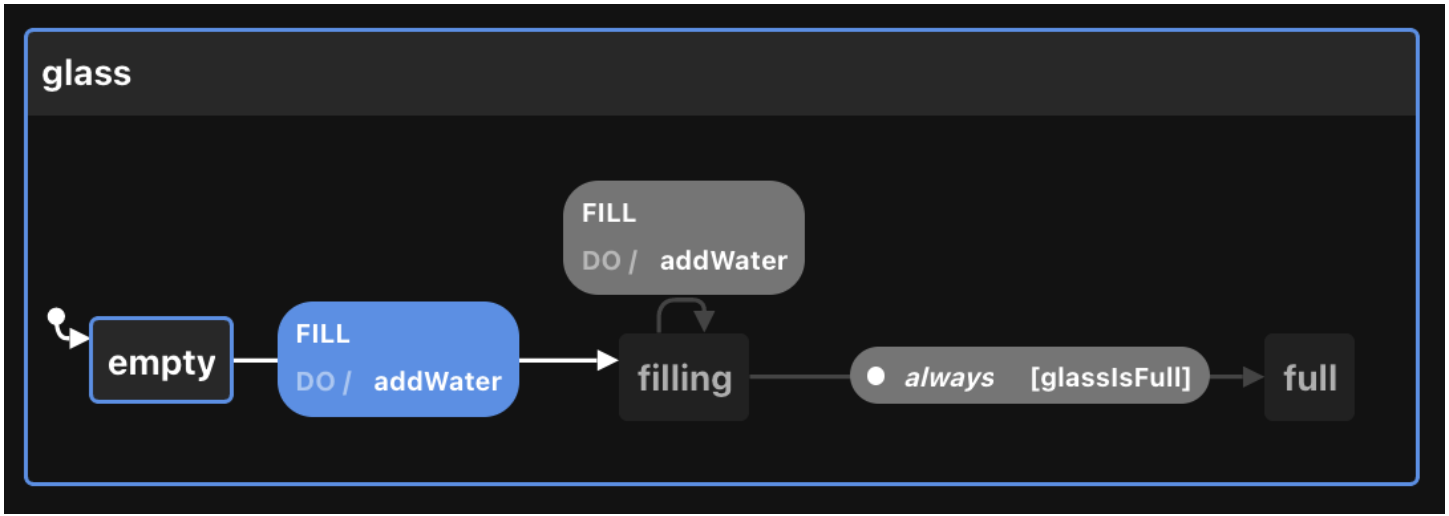
https://statecharts.dev/valid-invalid-enabled-disabled-changed-unchanged.svg

Statecharts solve the state explosion problem with parallel and nested states



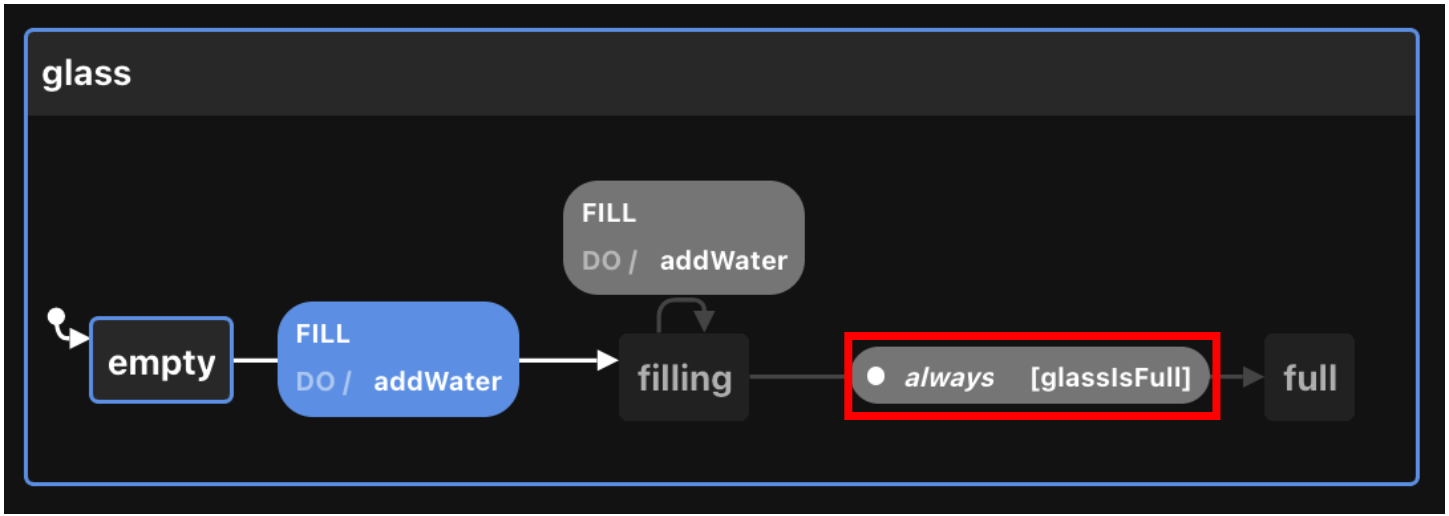
https://statecharts.dev/valid-invalid-enabled-disabled-changed-unchanged-parallel-hierarchy.svg

Statecharts are extended state machines

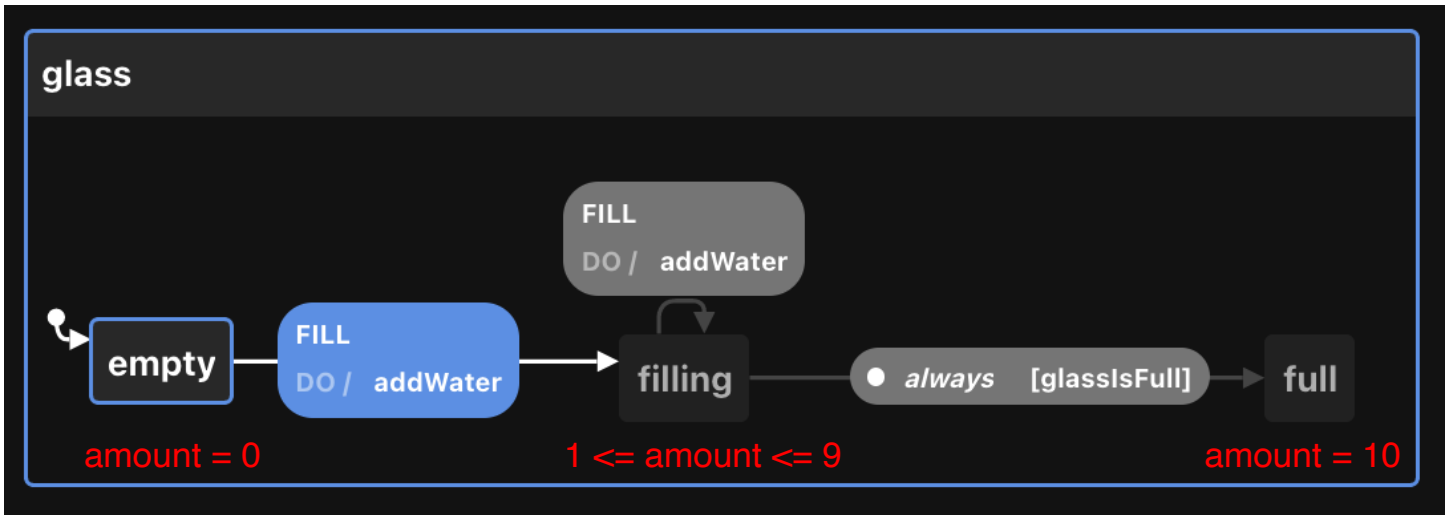


https://stately.ai/viz/790f79f2-abcd-424d-a3ce-1fcd755be863

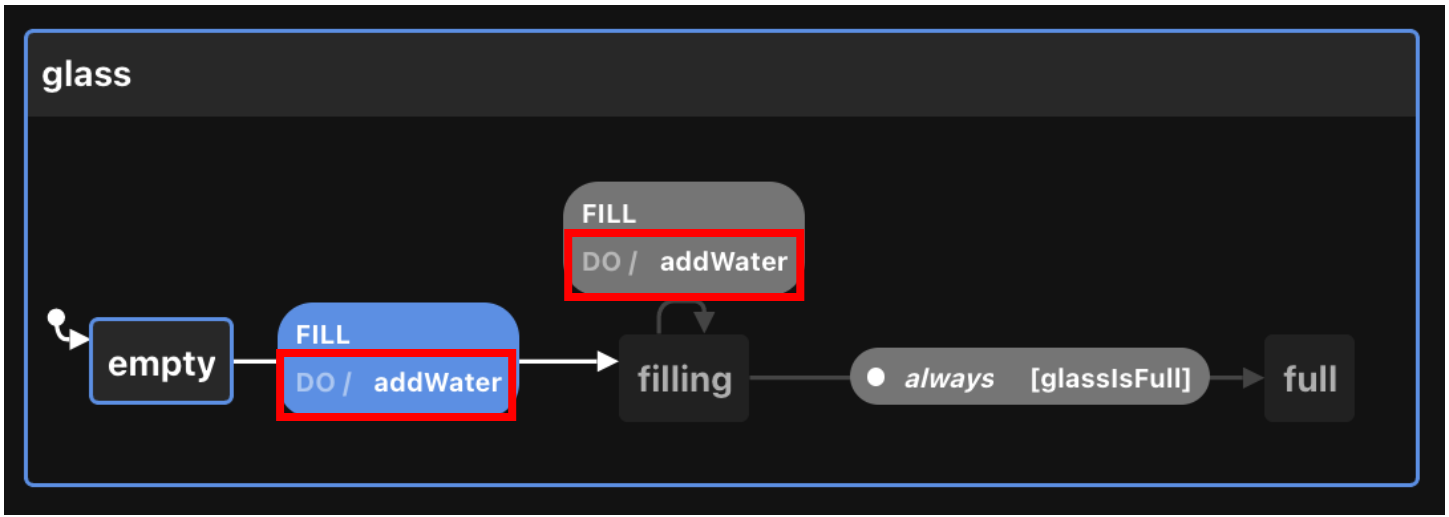
Guarded transitions are the if-else logic for statecharts



Extended state (context) allows you to save additional data

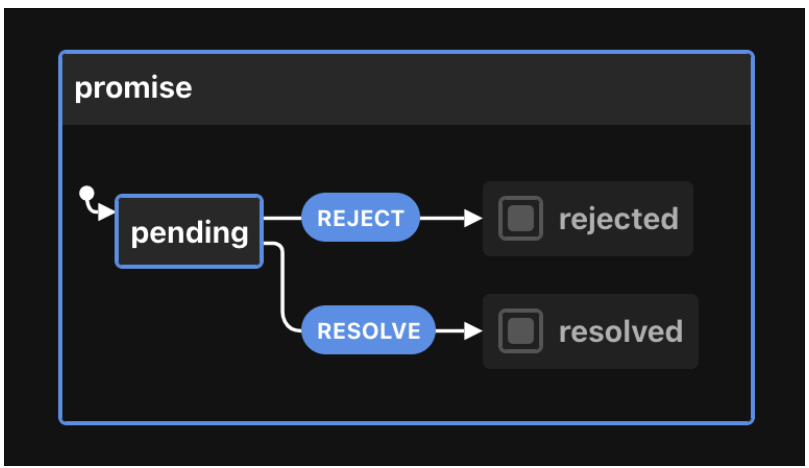


Actions allow you to fire side-effects on entry, exit or transition.



XState is a frameword-agnostic JS/TS-library for creating executable statecharts

- Visualizer: code -> diagram
- Editor (beta): diagram -> code




```
import { createMachine } from 'xstate';

const promiseMachine = createMachine({
  id: 'promise',
  initial: 'pending',
  states: {
    pending: {
      on: {
        RESOLVE: { target: 'resolved' },
        REJECT: { target: 'rejected' }
      }
    },
    resolved: {
      type: 'final'
    },
    rejected: {
      type: 'final'
    }
  }
});
```

XState example: Guards, context and the assign-action

```
const states = {
  empty: {
    on: {
      FILL: {
        target: 'filling',
        actions: 'addWater'
      }
    }
  },
  filling: {
    // Transient transition
    always: {
      target: 'full',
      cond: 'glassIsFull'
    },
    on: {
      FILL: {
        target: 'filling',
        actions: 'addWater'
      }
    }
  },
  full: {}
}
```

```
import { createMachine, assign } from 'xstate';
// Action to increment the context amount
const addWater = assign({
  amount: (context, event) => context.amount + 1
});
// Guard to check if the glass is full
const glassIsFull = function (context, event) {
  return context.amount >= 10;
};

const glassMachine = createMachine({
  id: 'glass',
  // Extended state
  context: {
    amount: 0
  },
  initial: 'empty',
  states,
},
{
  actions: { addWater },
  guards: { glassIsFull }
});
```

XState example: Delayed transitions

```
const lightDelayMachine = createMachine({
  id: 'lightDelay',
  initial: 'green',
  states: {
    green: {
      after: {
        // after 1 second, transition to yellow
        1000: { target: 'yellow' }
      }
    },
    yellow: {
      after: {
        // after 0.5 seconds, transition to red
        500: { target: 'red' }
      }
    },
    red: {
      after: {
        // after 2 seconds, transition to green
        2000: { target: 'green' }
      }
    }
  }
});
```

XState example: Invoking a service (promise)

```
const fetchUser = (userId) ⇒
  fetch(`url/to/user/${userId}`)
    .then((response) ⇒ response.json());

const loading = {
  invoke: {
    id: 'getUser',
    src: (context, event) ⇒
      fetchUser(context.userId),
    onDone: {
      target: 'success',
      actions: assign({ user: (context, event) ⇒ event.data })
    },
    onError: {
      target: 'failure',
      actions: assign({ error: (context, event) ⇒ event.data })
    }
  }
}
```

```
const userMachine = createMachine({
  id: 'user',
  initial: 'idle',
  context: {
    userId: 42,
    user: undefined,
    error: undefined
  },
  states: {
    idle: {
      on: {
        FETCH: { target: 'loading' }
      }
    },
    loading,
    success: {},
    failure: {
      on: {
        RETRY: { target: 'loading' }
      }
    }
  }
});
```

XState example: Spawning actors (another machine)

```
import { createMachine, spawn } from 'xstate';
import { todoMachine } from './todoMachine';

const todosMachine = createMachine({
  // ...
  on: {
    'NEW_TODO.ADD': {
      actions: assign({
        todos: (context, event) => [
          ...context.todos,
          {
            todo: event.todo,
            // add a new todoMachine actor with a unique name
            ref: spawn(todoMachine, `todo-${event.id}`)
          }
        ]
      })
    }
    // ...
  }
});
```

Some XState features not introduced in this presentation

- History states
- Activities
- Delayed events
- Invoking callbacks & observables
- Parallel states
- …

Workshop: Implement UI logic for Wordle using XState

https://github.com/KnowitJSTSGuild/ui-modeling-with-statecharts