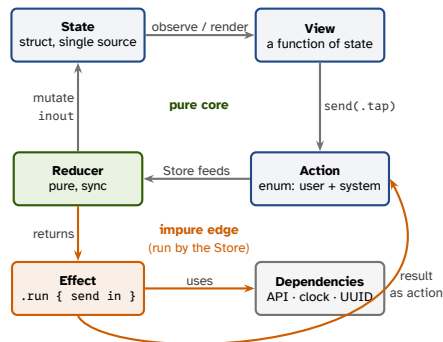
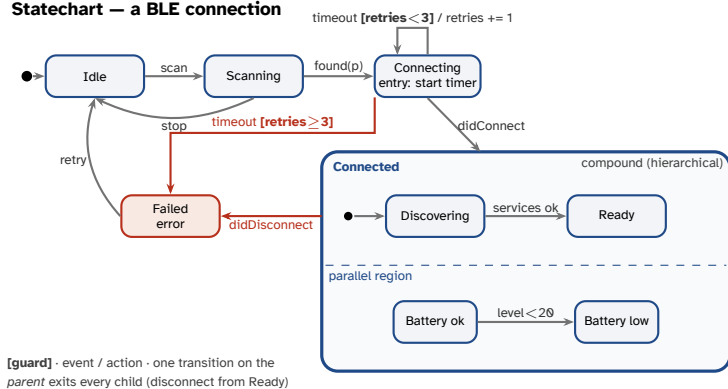


**In one line:** State lives in **one place**; the view **renders** it and sends **actions**; a **pure reducer** computes the next state and returns **effects** (side effects as values) whose results come back as actions — one loop, one direction (Elm → Redux → TCA). A **state machine** constrains *which* transitions exist; **statecharts** add hierarchy, parallel regions and guards. Both aim at “**make illegal states unrepresentable**”.

#### The loop (TCA names)



#### Statechart — a BLE connection



## How it works

- **Elm**: update returns (new model, Cmd) — the ancestor of “reducer returns effects”. **Redux**: dispatch(action); reducer (state, action) → new state; effects live in **middleware** (thunks, sagas).
- **TCA**: @Reducer = @ObservableState struct State + enum Action + Reduce { state, action in } returning Effect: .none · .send · .run { send in } · .cancellable(id:). The Store runs effects. **Dependencies** (@Dependency: API, clock, UUID) are overridden in tests and previews.
- **Composition**: parent state/action embed the child’s (Scope, .ifLet, .forEach); views get store.scope( ... ). Navigation is state (@Presents, StackState).
- **Testing**: TestStore is **exhaustive** — assert every state change and receive every effect action, or it fails.
- **FSM** = states + events + transition (state, event) → state. **Statechart** (Harel 1987) adds **hierarchy, parallel (orthogonal) regions, guards, entry/exit actions, history**. W3C **SCXML**; **XState** in JS.

## vs MVVM

| Unidirectional / TCA                                                                                                                     | MVVM                                                                                           |
|------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| one reducer mutates; replayable effects explicit, cancellable by id exhaustive, deterministic tests boilerplate, learning curve, lock-in | VM methods mutate anywhere ad-hoc Tasks per VM you pick what to assert light, native, familiar |

**Perf**: a huge root state re-renders too much — scope stores narrowly; @ObservableState tracks only the fields a view reads. Every tap is several hops (action → reducer → effect → action).

## When it pays / when it’s ceremony

**Pays**: complex shared state, many async effects + cancellation, deep navigation/links, a big team wanting one shape; flows with real protocol states (BLE, checkout, onboarding, player). **Ceremony**: small apps, forms, prototypes — plain @Observable. The *principles* (one owner, explicit transitions, effects at the edge) pay even without the framework.

## Example — illegal states unrepresentable

```

// BAD: 2^3 combos, most meaningless
struct Conn { var connecting, connected: Bool; var error: Error? }
// GOOD: only real states; data lives where it is valid
enum BLEState {
  case idle, scanning
  case connecting(Peripheral, tries: Int)
  case connected(Peripheral, services: [Service])
  case failed(Error)
}
func next(_ s: BLEState, _ e: BLEEvent) -> BLEState { // pure
  switch (s, e) {
  case (.idle, .scan): return .scanning
  case let (.scanning, .found(p)):
    return .connecting(p, tries: 0)
  case let (.connecting(p, n), .timeout) where n < 3: // guard
    return .connecting(p, tries: n + 1)
  case (.connecting, .timeout): return .failed(BLError.timeout)
  default: return s // event not allowed here: ignored
  }
}
  
```

In TCA this switch is the reducer body; the CoreBluetooth delegate is a **dependency** whose callbacks come back as actions.

## Interview traps

- ! “The reducer calls the API.” No — sync and pure; it *returns* an effect; the result re-enters as an action.
- ! State as a class — no: value types; @ObservableState gives fine-grained observation of the struct.
- ! Booleans for modes (isLoading true *and* an error set?) — use an enum.
- ! default: return s hides missing transitions — list them where the machine is safety-relevant.

## Remember & likely questions

**State → View → Action → Reducer → State; effects out, actions back in; enums, not flags.**

1. Where do side effects go? — returned Effects, run by the Store.
2. FSM vs statechart? — + hierarchy, parallel regions, guards, history.
3. TCA perf pitfall? — observing a huge state; scope stores narrowly.