State Machines

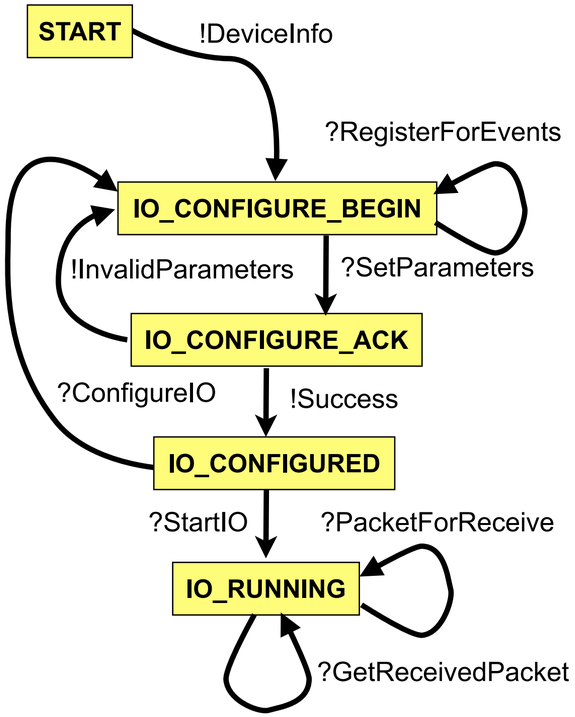
- What is a state machine?
- Implementation using **`enum`** types
- Implementation using **`struct`** types

CC BY ND

State Machines

- State machines common in systems code
  - Network protocols
  - File systems
  - Device drivers

- System behaviour modelled as a finite state machine comprising:
  - **States** that reflect the status of the system
  - **Events** that trigger transitions between states
  - **State** variables that hold system configuration
- Clean high-level model of a system
  - Captures the essence of the behaviour
  - Easy to reason about and prove properties such as termination, absence of deadlocks, reachability, etc.



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Implementing State Machines

- Hard to cleanly model state machine in code
  - Structure of code tends not to match structure of state machine; not easy to visualise transitions
  - Difficult to validate code against specification

- Approaches to modelling state machines in strongly-typed functional languages:
  - Encode states and events as enumerations, pattern match on (state, event) tuples
  - Encode states as types and transitions as functions
  - Add first-class state machine support to language
    - Microsoft Singularity research operating system
    - **`async`**/**`await`** asynchronous code

```
contract NicDevice {
  out message DeviceInfo(...);
  in  message RegisterForEvents(NicEvents.Exp:READY
  c);
  in  message SetParameters(...);
  out message InvalidParameters(...);
  out message Success();
  in  message StartIO();
  in  message ConfigureIO();
  in  message PacketForReceive(byte[] in ExHeap p);
  out message BadPacketSize(byte[] in ExHeap p, int
  m);
  in  message GetReceivedPacket();
  out message ReceivedPacket(Packet * in ExHeap p);
  out message NoPacket();

  state START: one {
    DeviceInfo! → IO_CONFIGURE_BEGIN;
  }
  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents? →
      SetParameters? → IO_CONFIGURE_ACK;
  }
  state IO_CONFIGURE_ACK: one {
    InvalidParameters! → IO_CONFIGURE_BEGIN;
    Success! → IO_CONFIGURED;
  }
  state IO_CONFIGURED: one {
    StartIO? → IO_RUNNING;
    ConfigureIO? → IO_CONFIGURE_BEGIN;
  }
  state IO_RUNNING: one {
    PacketForReceive? → (Success! or BadPacketSize!)
      → IO_RUNNING;
    GetReceivedPacket? → (ReceivedPacket! or
    NoPacket!)
      → IO_RUNNING;
    ...
  }
}
```

Listing 1. Contract to access a network device driver.

G. Hunt and J. Larus. "Singularity: Rethinking the software stack", ACM SIGOPS OS Review, 41(2), April 2007. DOI:10.1145/1243418.1243424

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Enumerations for modelling state machines

- Possible state machine representation:
  - An enumerated type (**`enum`**) models alternatives
    - Define an **`enum`** to represent the states
    - Define an **`enum`** to represent the events
  - Functions represent transitions and actions:
    - Define a function to map from (state, events) tuples to next state
    - Define a function to perform the actions associated with each state
- Builds on the intuition that **`enum`** types express alternatives, and a state machine comprises a list of alternative states

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Using `enum` to Model State Machines: Example (1/3)

```
enum ApcState {
    Initialize,
    WaitForConnect,
    Accept(TcpStream),
    StartTransfer(TcpStream),
    Waiting(TcpStream),
    ReceiveMsg(TcpStream, Vec<u8>),
    SendNop(TcpStream),
    Closed,
    Finish,
    Failure(String),
}
```

```
enum ApcEvent {
    TcpConnected(TcpStream),
    ResponseValid(bool),
    IncomingTcpClosed,
    AspMsgIn(Vec<u8>),
    NopTimeout,
    Finished,
    Uct,
}
```

Example adapted from comment on
https://hoverbear.org/2016/10/12/rust-state-machine-pattern/

- Define an **`enum`** to represent states and another for the events
- Encode state variables as **`enum`** parameters

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Using `enum` to Model State Machines: Example (2/3)

```
impl ApcState {
    fn next(self, event: ApcEvent) -> Self {
        use self::ApcState::*;
        use self::ApcEvent::*;

        match (self, event) {
            (Initialize,         TcpConnected(tcp)   ) => Accept(tcp),
            (Initialize,         Finished            ) => Finish,
            (Accept(tcp),        ResponseValid(true) ) => StartTransfer(tcp),
            (Accept(_),          ResponseValid(false)) => Closed,
            (StartTransfer(tcp), Uct                 ) => Waiting(tcp),
            (Waiting(_),         IncomingTcpClosed   ) => Closed,
            (Waiting(_),         Finished            ) => Finish,
            (Waiting(tcp),       AspMsgIn(msg)       ) => ReceiveMsg(tcp, msg),
            (Waiting(tcp),       NopTimeout          ) => SendNop(tcp),
            (ReceiveMsg(tcp, _), Uct                 ) => Waiting(tcp),
            (SendNop(tcp),       Uct                 ) => Waiting(tcp),
            (s, e) => Failure(format!("Invalid State/Event combination: {:?}/{:?}", s, e)),
        }
    }
}
```

- Match against states and events: clean representation of state-transition table
- Straight-forward to validate against specification

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Using `enum` to Model State Machines: Example (3/3)

```
pub struct ApcStateMachine {
    state   : ApcState,
    addr    : SocketAddr,
    timeout: u64,
}

impl ApcStateMachine {
    fn new() -> ApcStateMachine {
        ...
    }

    fn run_once(&self) -> ApcEvent {
        match self.state {
            Initialize          => ...
            WaitForConnect      => ...
            Accept(tcp)         => ...
            StartTransfer(_)    => ...
            Waiting(tcp)        => ...
            ReceiveMsg(_, msg)  => ...
            SendNop(_)          => ...
            Closed              => ...
            Finish              => ...
        }
    }
}
```

```
fn run_state_machine() {
    let mut sm = ApcStateMachine::new();
    loop {
        let event = sm.run_once();
        sm.state = sm.state.next(event);
        if sm.state == ApcState::Finish {
            break;
        }
    }
}
```

- **`match`** loop dispatches to functions
  - Performs actions each state, returns next event to process → determine next state
  - Parameterised `enum` with state variables makes it easy to pass parameters
- Pattern matching on **`enum`** gives a clear implementation
  - Compiler checks all alternates covered
  - Easy to pass state variables

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Structures for Modelling State Machines

- Alternative state machine representation:
  - Define a **`struct`** representing each state
  - Model an event as a method call on a **`struct`**
  - Model state transitions by returning a **`struct`** representing the new state
- Builds on the intuition that states hold concrete *state*, and events are things that happen in states

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Using `struct` to Model State Machines: Example

- Define a **`struct`** representing each state
  - State variables are fields within the **`struct`**
  - Methods implemented on the **`struct`** encode state transitions and event handlers
    - Return **`Self`** if state is unchanged
    - Return **`struct`** representing new state if state changes

```
struct UnauthenticatedConnection {
  socket : TcpSocket,
 ...
}
```

```
struct AuthenticatedConnection {
  socket : TcpSocket,
 ...
}
```

```
struct NotConnected;
```

```
impl UnauthenticatedConnection {
  fn login(self, c: Credentials) -> Result<AuthenticatedConnection, (self, LoginError)> {
    ...
  }

  fn disconnect(self) -> NotConnected {
    ...
  }
}
```

- Encodes states and state transitions in types and *ownership rules*

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Ownership and state machines

- **`struct`**-based approach to state machines uses ownership rules to enforce state transitions
  - Methods that change state take ownership of **`self`**, return new **`struct`**:

```
impl UnauthenticatedConnection {
  fn login(self, c: Credentials) -> Result<AuthenticatedConnection, (self, LoginError)> {
    ...
  }

  fn disconnect(self) -> NotConnected {
    ...
  }
}
```

  - e.g., the **`login()`** function consumes its **`UnauthenticatedConnection`** and returns a new **`AuthenticatedConnection`** on success
    - The compiler enforces this – the **`UnauthenticatedConnection`** is not accessible after this call; all resources it owned are reclaimed
    - Except any values the **`login()`** method explicitly copies to the **`AuthenticatedConnection`**

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Approaches to Representing State Machines

- **`enum`**-based approach is compact, makes states and events clear in the types, and has clear state transition table
  - Relies on expressive **`enum`** types for implementation – works well in languages like Rust, Swift, OCaml, …
  - Difficult to express in languages with weaker **`enum`** types and pattern matching
- **`struct`**-based approach encodes states and state transitions in the types, events as methods on those types
  - State transition table is less obviously explicit in the code
  - State transitions use Rust ownership rules to enforce transitions – cannot easily check state transitions in languages without explicit data ownership

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

State Machines

- What is a state machine?
- Implementation using **`enum`** types
- Implementation using **`struct`** types

CC BY ND