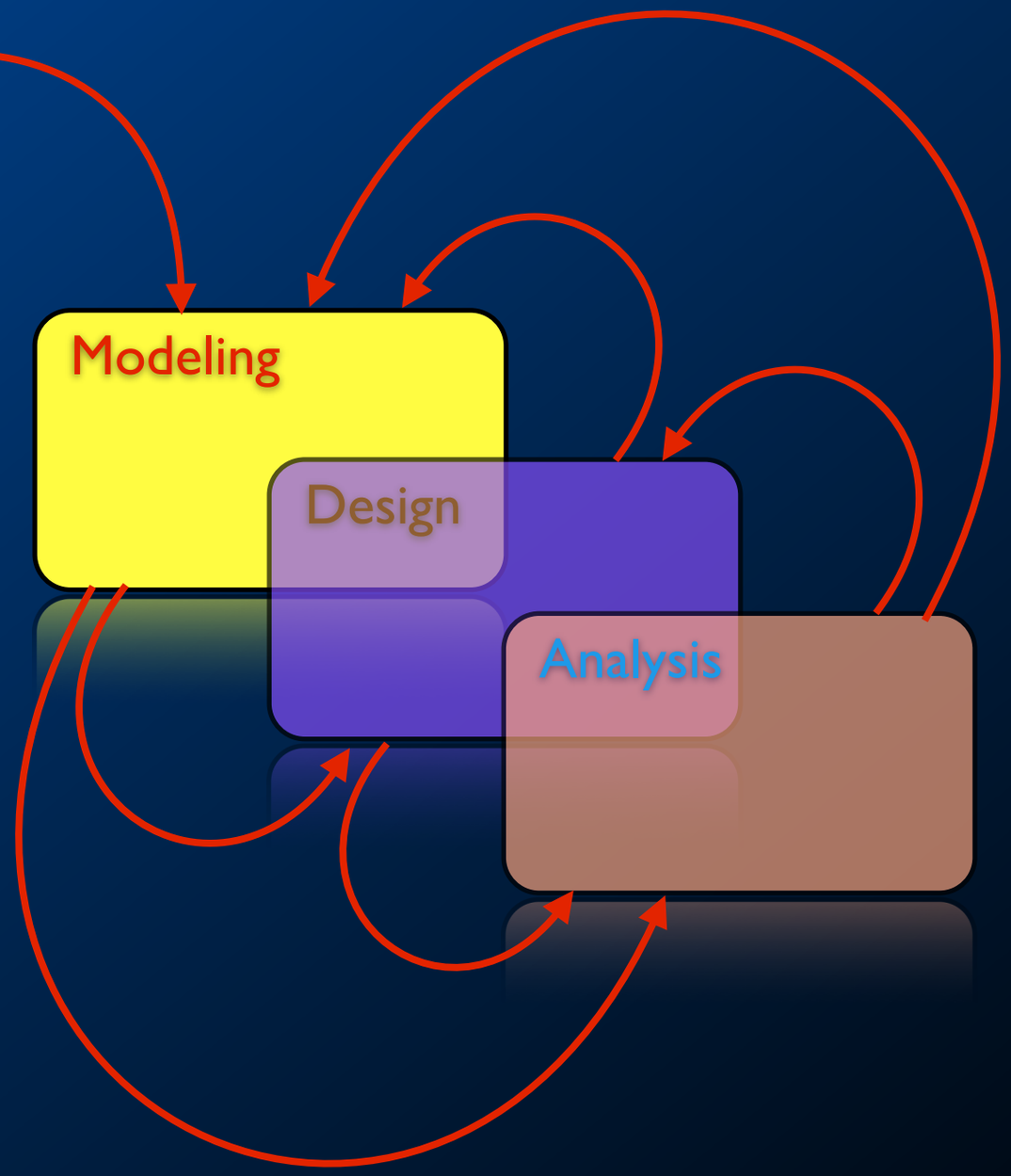
# Modelling of Embedded System

# Modeling, Design, Analysis

- Modeling is the process of gaining a deeper understanding of a system through imitation. Models specify what a system does.
- Design is the structured creation of artifacts. It specifies how a system does what it does.
- Analysis is the process of gaining a deeper understanding of a system through dissection. It specifies why a system does what it does (or fails to do what a model says it should do).




# What is Modeling?

- Developing insight about a system, process, or artifact through imitation.
- A model is the artifact that imitates the system, process, or artifact of interest.
- A mathematical model is model in the form of a set of definitions and mathematical formulas.

# Good Models

- Simple
- Amenable for development of theory
  - Theorems allow for generalizations and short-cuts
  - Should not be too general (theorems become too weak)
- High Expressive Power
  - Compact representation enables higher productivity
- Provides Ability for Critical Reasoning
- Executable
  - Simulation/Validation
- Synthesizable
  - Usually requires design orthogonality (e.g. compiler)
- Unbiased towards any specific implementation
  - Extremely hard to achieve, but worth it.
- Fit the task at hand
  - If the model doesn't fit, too much work is needed to use it…

# Common Models of Systems

- state-oriented model
  - Finite State Machines
- activity-oriented model
  - Data-Flow/Process Models，把系统描绘成一组与数据或者执行的相关性有关的活动的集合；
- structure-oriented model
  - 框图
  - 着重强调系统的物理构成
- data-oriented model
  - 实体-关系图
- heterogeneous-oriented model
  - 综合前四种模型特征
  - 表达复杂系统的多种不同视图

# What is Model-Based Design?

- Create a mathematical model of all the parts of the embedded system
  - Physical world
  - Control system
  - Software environment
  - Hardware platform
  - Network
  - Sensors and actuators
- Construct the implementation from the model
  - Goal: automate this construction, like a compiler
  - In practice, only portions are automatically constructed

# Modeling Techniques

- Models that are abstractions of system dynamics (how things change over time)
- Examples:
  - Modeling physical phenomena – ODEs
  - Feedback control systems – time-domain modeling
  - Modeling modal behavior – FSMs, hybrid automata
  - Modeling sensors and actuators – calibration, noise
  - Modeling software – concurrency, real-time models
  - Modeling networks – latencies, error rates, packet loss

# Taxonomy of Computations

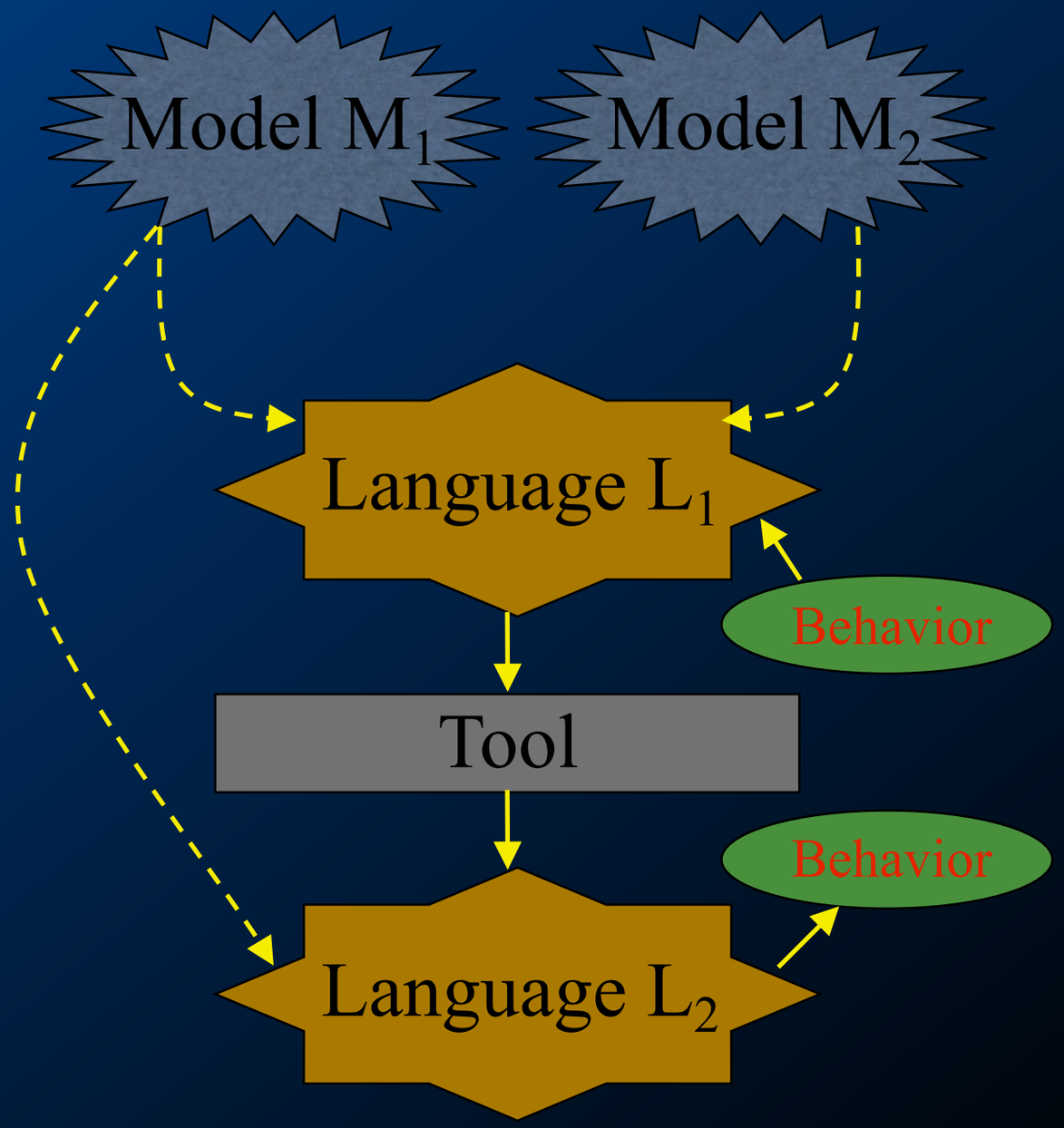
- "Regular" vs "Event-Driven"
  - Regular examples: streaming applications, processors,…
  - Event-driven: "reactive" (triggered by changes in environment)
- Real-Time
  - Hard vs. Soft deadlines
  - Periodic vs. Aperiodic
- Control, Data, Communication
  - How much of each?
- Parallelism, Geographic distances
  - Highly concurrent vs. sequential; distributed systems
- Deterministic vs. Non-deterministic
  - Degree of predictability

# Models, Languages, & Tools

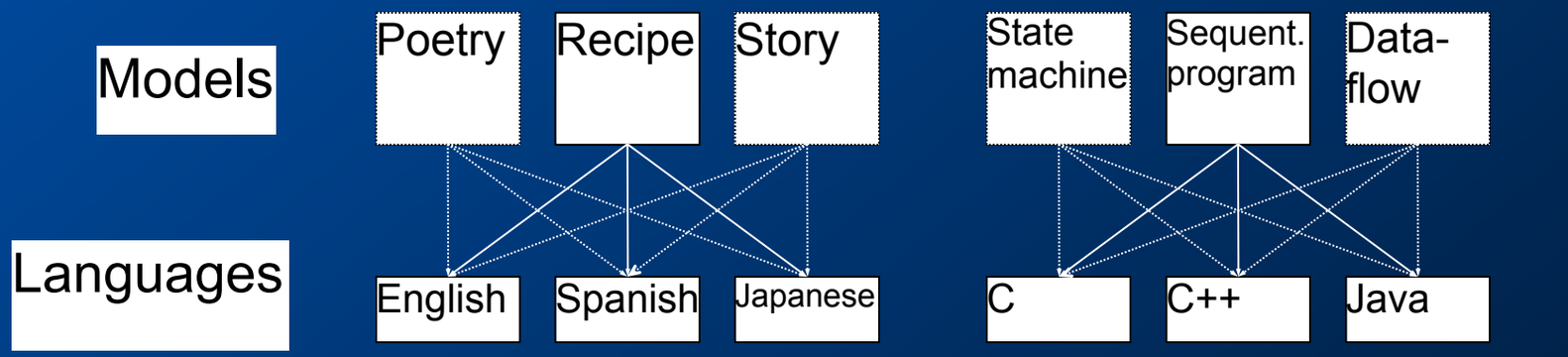
- A model of computation is a conceptual notions used to capture a system behavior, e.g.:
  - A set of objects
  - Composition rules
  - Execution semantics

- A languages defines the syntax to capture a models of computation

- A tool is a "compiler" transforming a model captured in one language to a model captured in another language




# Models vs. languages




- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program
- Languages capture models
  - Concrete form, e.g., English, C
- Variety of languages can capture one model
  - E.g., sequential program model  C,C++, Java
- One language can capture variety of models
  - E.g., C++ → sequential program model, object-oriented model, state machine model
- Certain languages better at capturing certain computation models

# Finite State Machines

- FSM = (
  {Input symbols},
  {Output Symbols},
  {States},
  {Initial States},
  Transition Relation (mapping of Input Symbol, Current State to next State(s)
  Output Function (mapping of Input Symbol, Current State to Current Output Symbol)
  )
- Often suitable for controllers, protocols
- Powerful algorithms for verification
- Easy to synthesize, but can be inefficient

# What is Modeled?

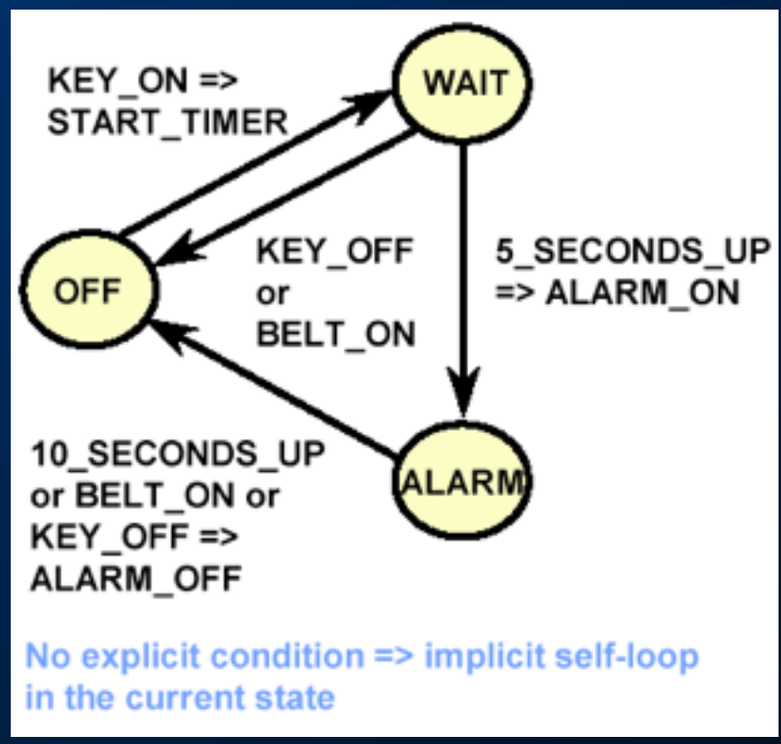
- System Input/Output sequences modeled
  - "State" set of properties occurring at a given time
    - "Inputs" and "Outputs" are the observable features of the system
  - "Transitions" allowed or observed pair-wise sequencing of states
    - Both states and transitions are inferred from the input/output sequence
  - If the outputs depend only on the state Moore else Mealy FSM
- Key assumption: the total memory available for states and the alphabet of input and output symbols is finite.

# Application

- Vending Machines
- Traffic Lights
- Video Games
- Text Parsing
- Regular Expression Matching
- CPU Controllers
- Protocol Analysis
- Natural Language Processing
- Speech Recognition

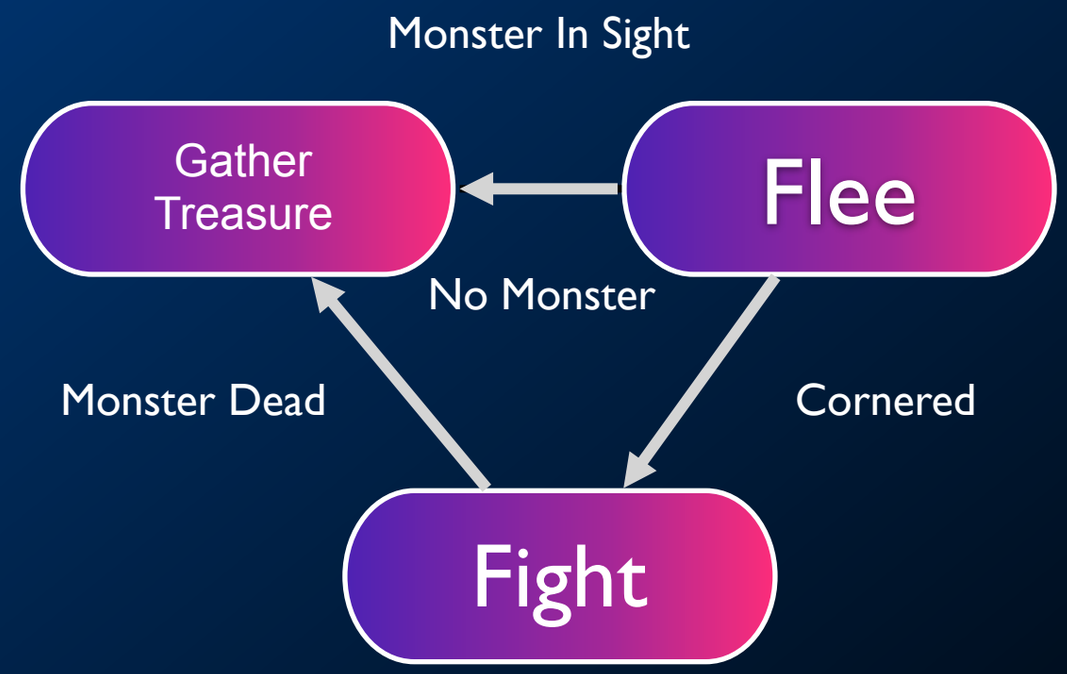
# FSM Example Model

- Informal specification
  - if driver turns on the key and does not fasten seat belt within 5 seconds then sound the alarm for 5 seconds or until driver fastens the seat belt or turns off the key
- Formal representation
  - Inputs = {KEY_ON, KEY_OFF, BELT_ON, BELT_OFF, 5_SECONDS_UP, 10_SECONDS_UP}
  - Outputs = {START_TIMER, ALARM_ON, ALARM_OFF}
  - States = {Off, Wait, Alarm}
  - Initial State = off
  - NextState: CurrentState, Inputs -> NextState
    - e.g. NextState(WAIT, {KEY_OFF}) = OFF
  - Outs: CurrentState, Inputs -> Outputs
    - e.g. Outs(OFF, {KEY_ON}) = START_TIMER



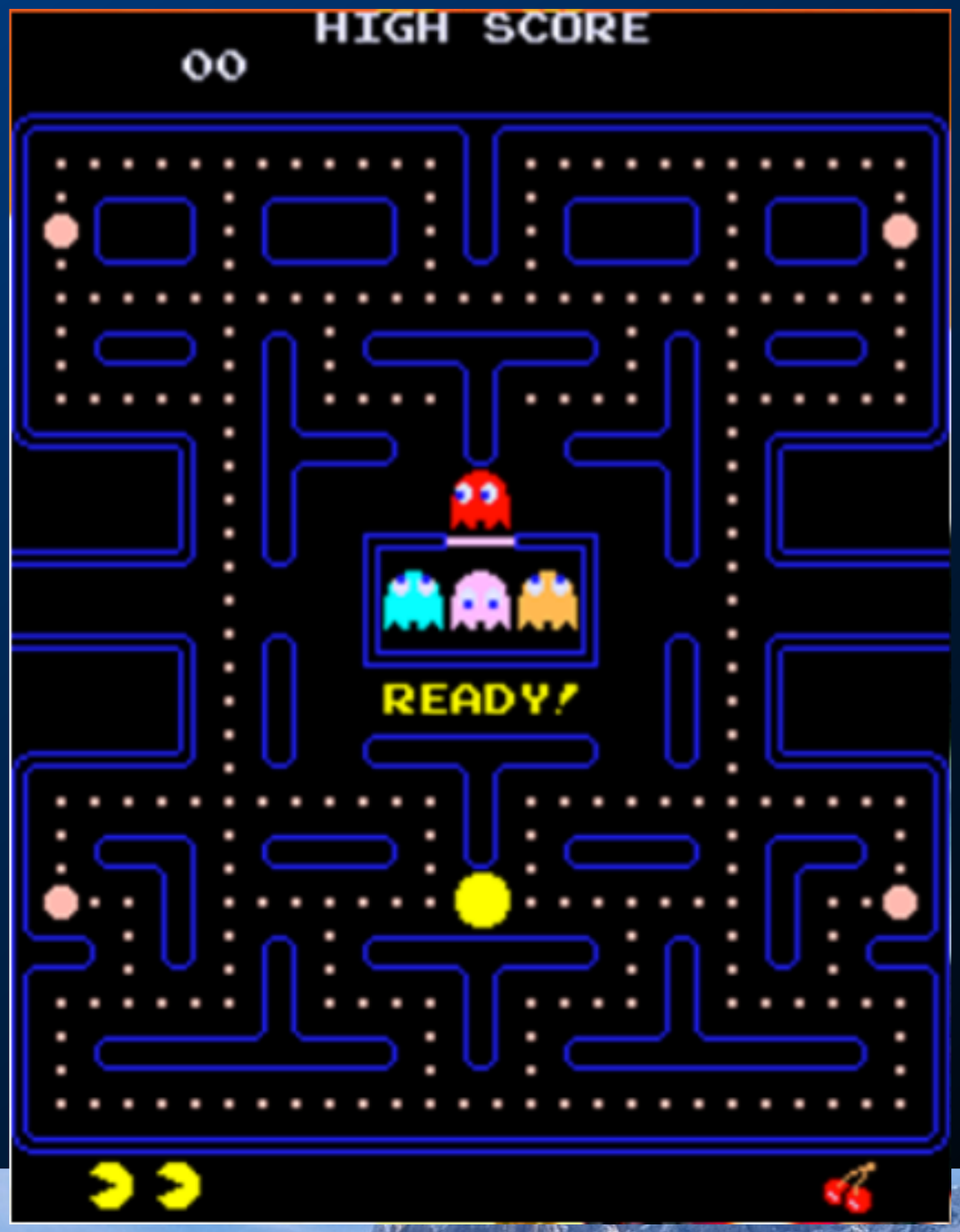

# FSM's in Games

- Character AI can be modeled as a sequence of mental states.
- World events can force a change in state.
- The mental model is easy to grasp, even for non-programmers.



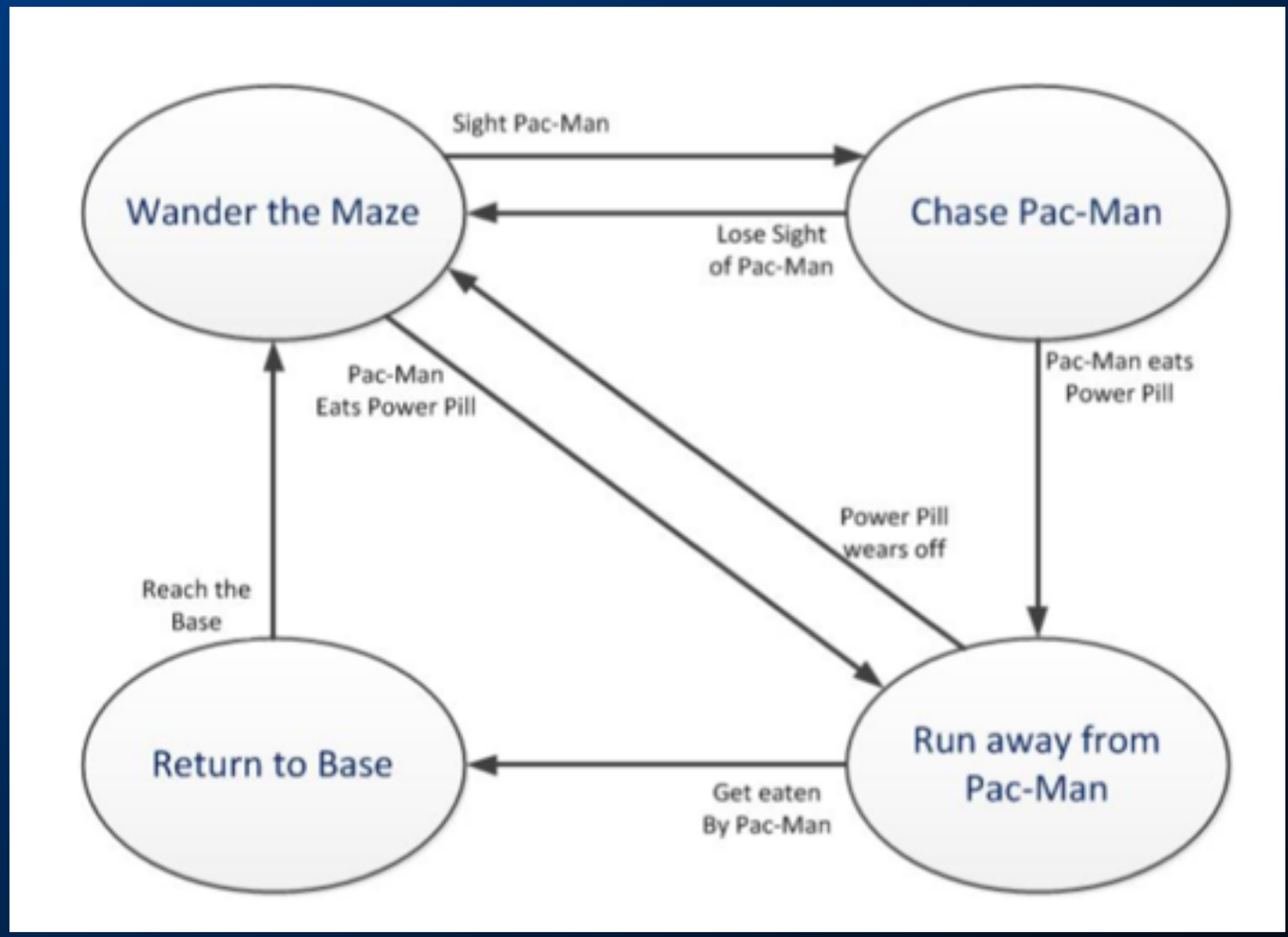

# Pac-Man

- Pac-Man requires the player to navigate through a maze, eating pellets and avoiding the ghosts who chase him through the maze
- Occasionally, Pac-Man can turn the tables on his pursuers by eating a power pellet, which temporarily grants him the power to eat the ghosts
- When this occurs, the ghosts' behavior changes, and instead of chasing Pac-Man they try to avoid him



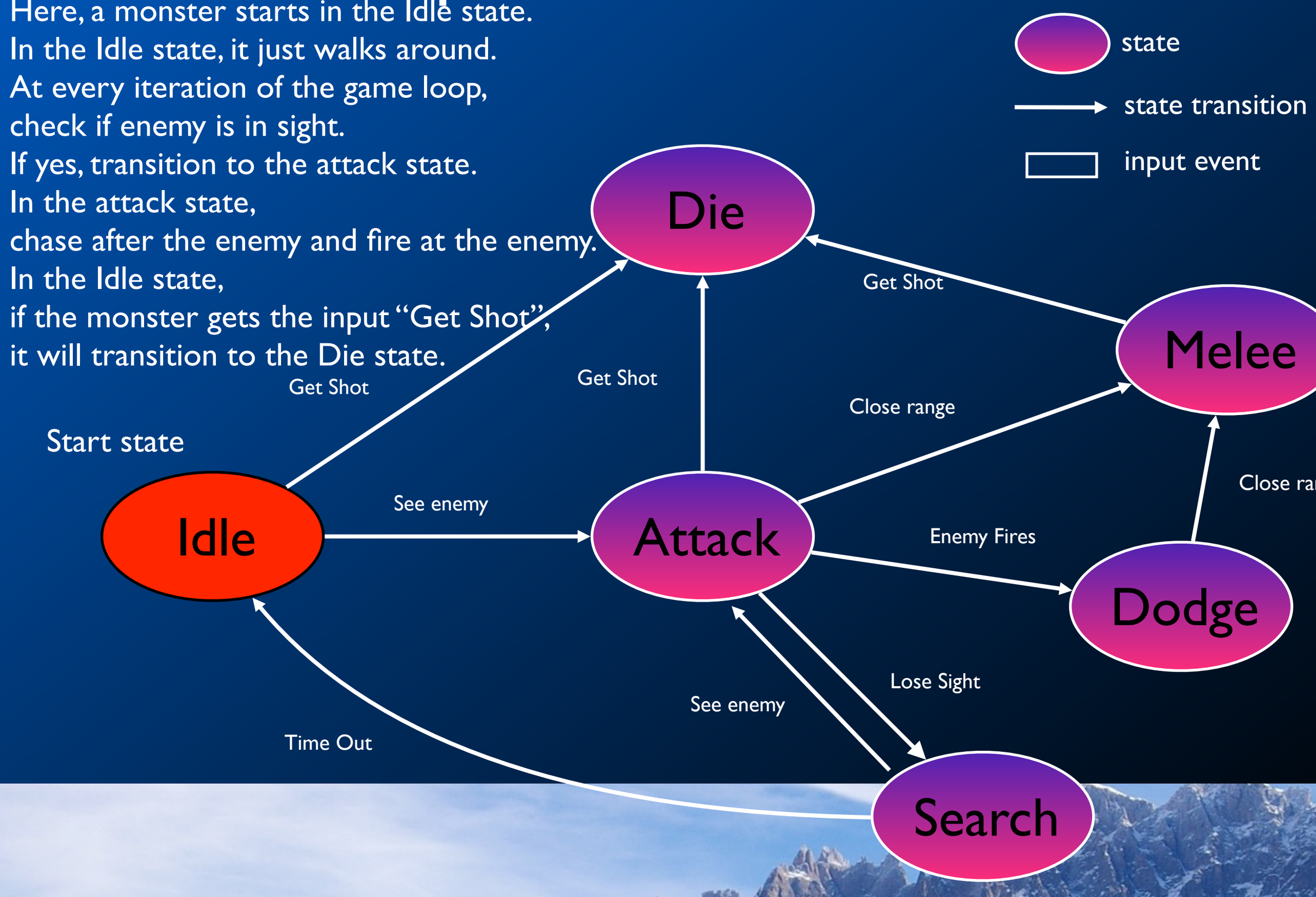
# Pac-man

- The ghosts in Pac-Man have four behaviors:
  - Randomly wander the maze
  - Chase Pac-Man, when he is within line of sight
  - Flee Pac-Man, after Pac-Man has consumed a power pellet
  - Return to the central base to regenerate

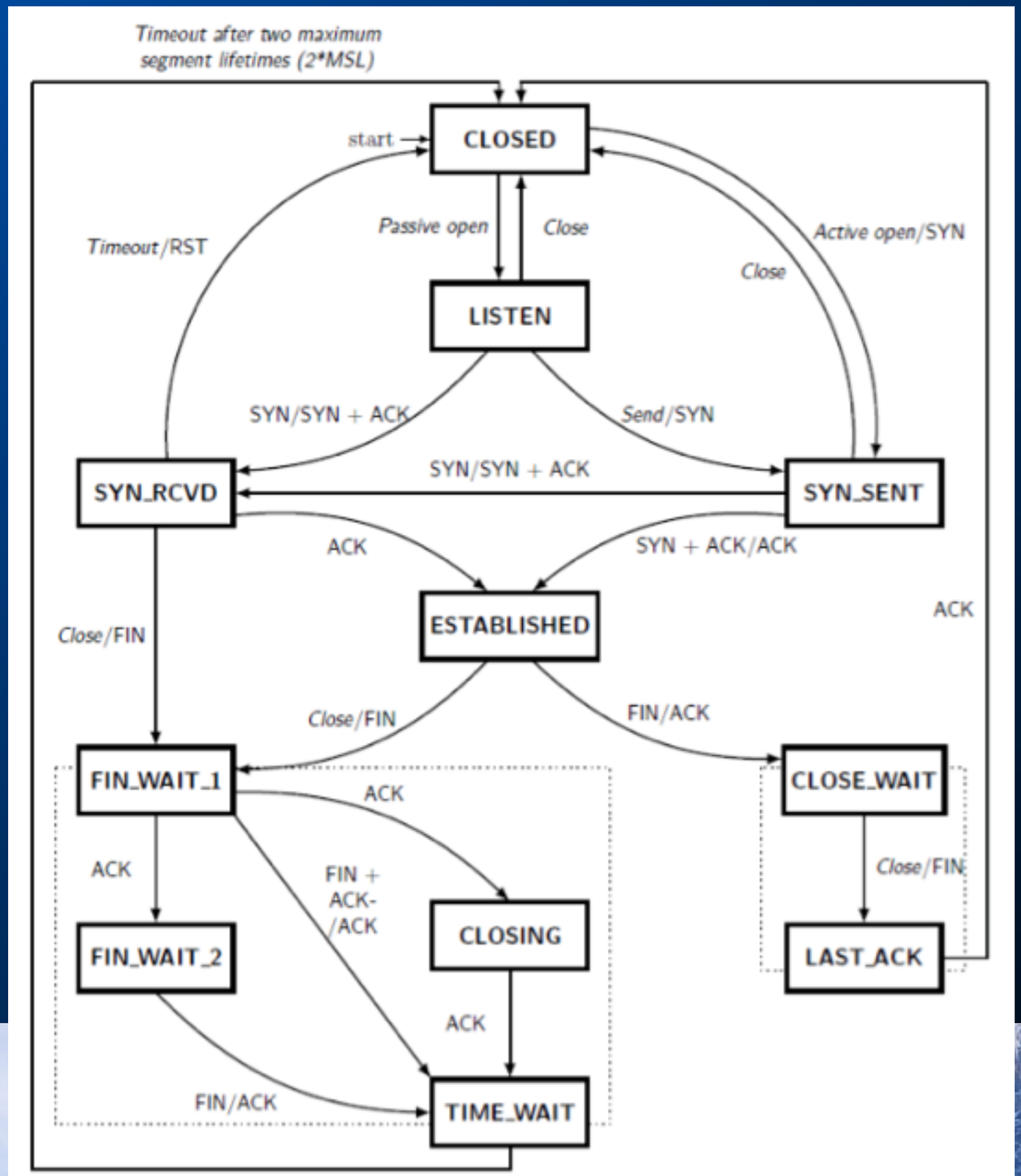



# Example of State Transitions

Here, a monster starts in the Idle state.
In the Idle state, it just walks around.
At every iteration of the game loop,
check if enemy is in sight.
If yes, transition to the attack state.
In the attack state,
chase after the enemy and fire at the enemy.
In the Idle state,
if the monster gets the input "Get Shot",
it will transition to the Die state.



# Internet Protocols: TCP as a DFA



# Standard FSM Nomenclature

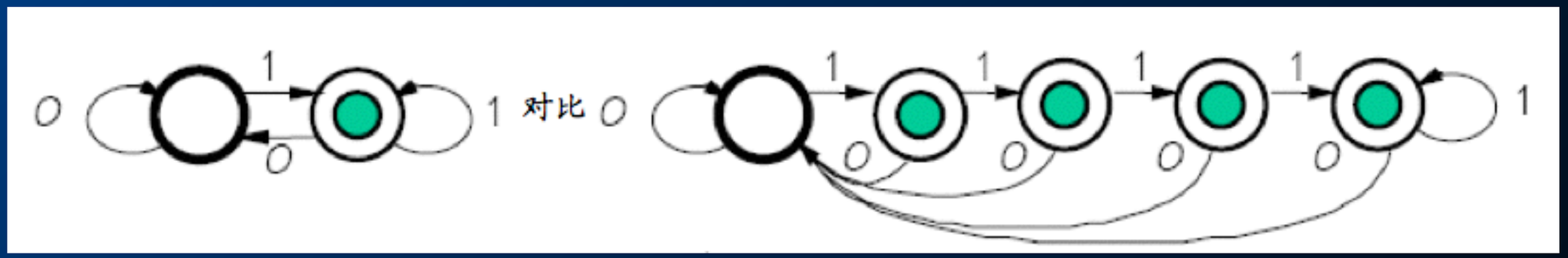
- Finite automata behavior classified by properties of the set of states, and the transition relation (next_states, output) = Φ(state, event) which describes possible next states and outputs after an event
- Finite Automata Classifications
  - "Deterministic" means that Φ(S,E) is single valued
  - "Completely Specified" means that Φ(S,E) has a value for every possible input
  - "Moore" means that outputs are fully determined by the current state – thus independent of the current event
  - "Mealy" means that outputs depend on both the current state and on the current event
  - "Synchronous" means that states change only on clocked intervals (events are polled)
  - "Asynchronous" means that events can happen at any time and the FSM updates on event

# Sampling (clocked) FSM

- sampled and event automata model most embedded system FSM components
  - Sampled automata query possible transitions every clock, transitions occur when sampled inputs change.
    - Commonly used to model clocked automata or Regular FSM models as well as software based dispatch and protocols
  - Samples are polled or interrupt sampled
- Event (asynchronous) automata comprise the hardware-based event interfaces that "latch" changes or signals
  - State transitions are immediate on event
  - Events are often signal transitions, not every sequence is feasible
  - E.g. flip-flop model, bus arbiter

# Equivalence

- Two FSM with identical outputs on all input strings are said to be equivalent
  - Equivalent does not require Isomorphic (Same shape) – I.e. the two equivalent machines need not have the same graph or even the same number of states!



# Equivalence allows for optimization

- In software, simple program implementations of FSM have complexity proportional to the number of transitions in a FSM
  - Often useful to minimize the number of states hence then number of transitions of a FSM
  - If the machine is deterministic and completely specified, this process can be done in time $O(s^2t)$ where s is the number of states and t is the number of transitions of the FSM
  - Problem is NP-hard if non-deterministic or incompletely specified.

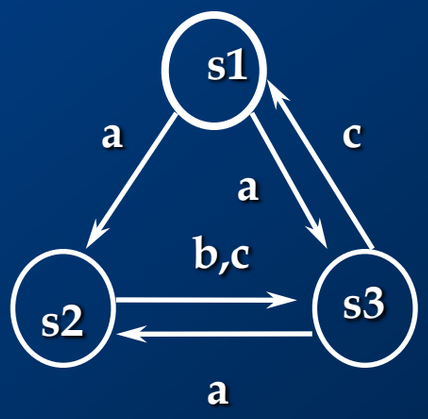
# Non-Deterministic FSM

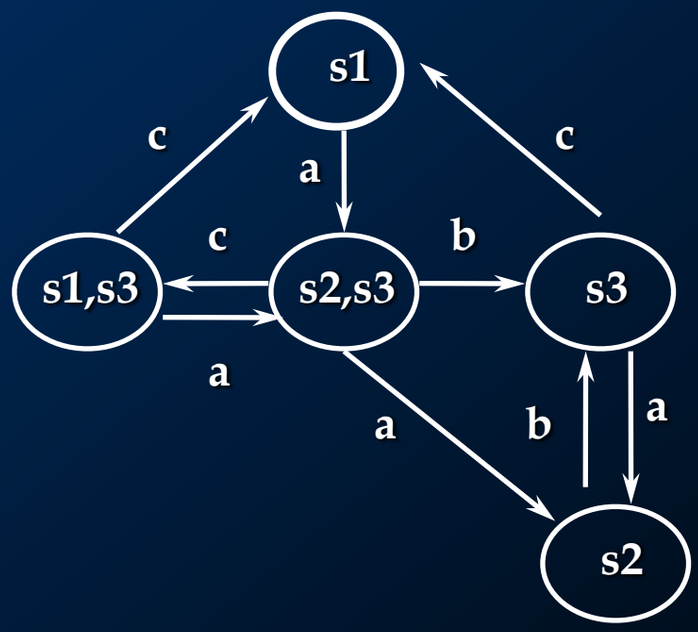
- A FSM is called non-deterministic when the Next State or Output functions are multiple valued
  - Next State is a relation, but Outputs agree:
    - Compact model (logic circuits)
  - Outputs disagree:
    - Formal non-determinism (often required for human semantic models)

- Non-determinism allows modeling:
  - Unspecified behavior
    - Incomplete specification
  - Unknown behavior
    - e.g., the environment model
  - Compact Models
    - Can be fully ‘deterministic’, just smaller model

# NFAs and FSMs

- Formally FSMs and NDAs are equivalent
  - (Rabin-Scott construction, Rabin ‘59)
- In practice, NFAs are often more compact
  - (exponential blowup for determinization)




# Mealy-Moore FSMs

- State models are single threaded
  - Only a single state can be valid at any time
- Moore state models: all actions are upon state entry
  - Non-reactive (response delayed by a cycle)
  - Easy to compose (always well-defined)
  - Good for implementation (i.e. good circuit analog)
- Mealy state models: all actions are in transitions
  - In software, leads to more compact code
  - Difficult for Hardware timing and composability

# Hierarchical FSM

- Support Concurrency and Hierarchy
- Natural rendition for software behavior
  - Same issues as MEALY for hardware designs
  - Irrelevant for Software, leads to more compact code

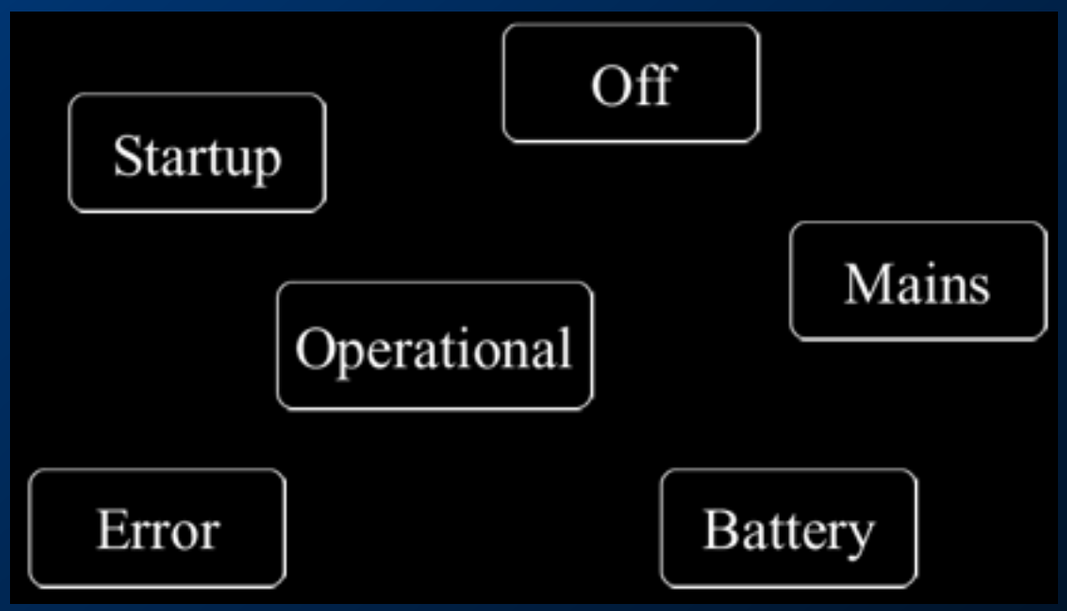
# Conventional FSM as Model

- Often over-specify implementation
  - Sequencing is fully specified (even when don't care)
- Poor Scalability due to lack of metaphor for composition
  - Number of states can be unmanageable
- No concurrency support
  - Often desire to reason about local sub-properties of a composite machine, but how to relate behavior of sub-states?

- Simple solution: Introduce hierarchy to model
  - Not as simple as it sounds

# Concurrency

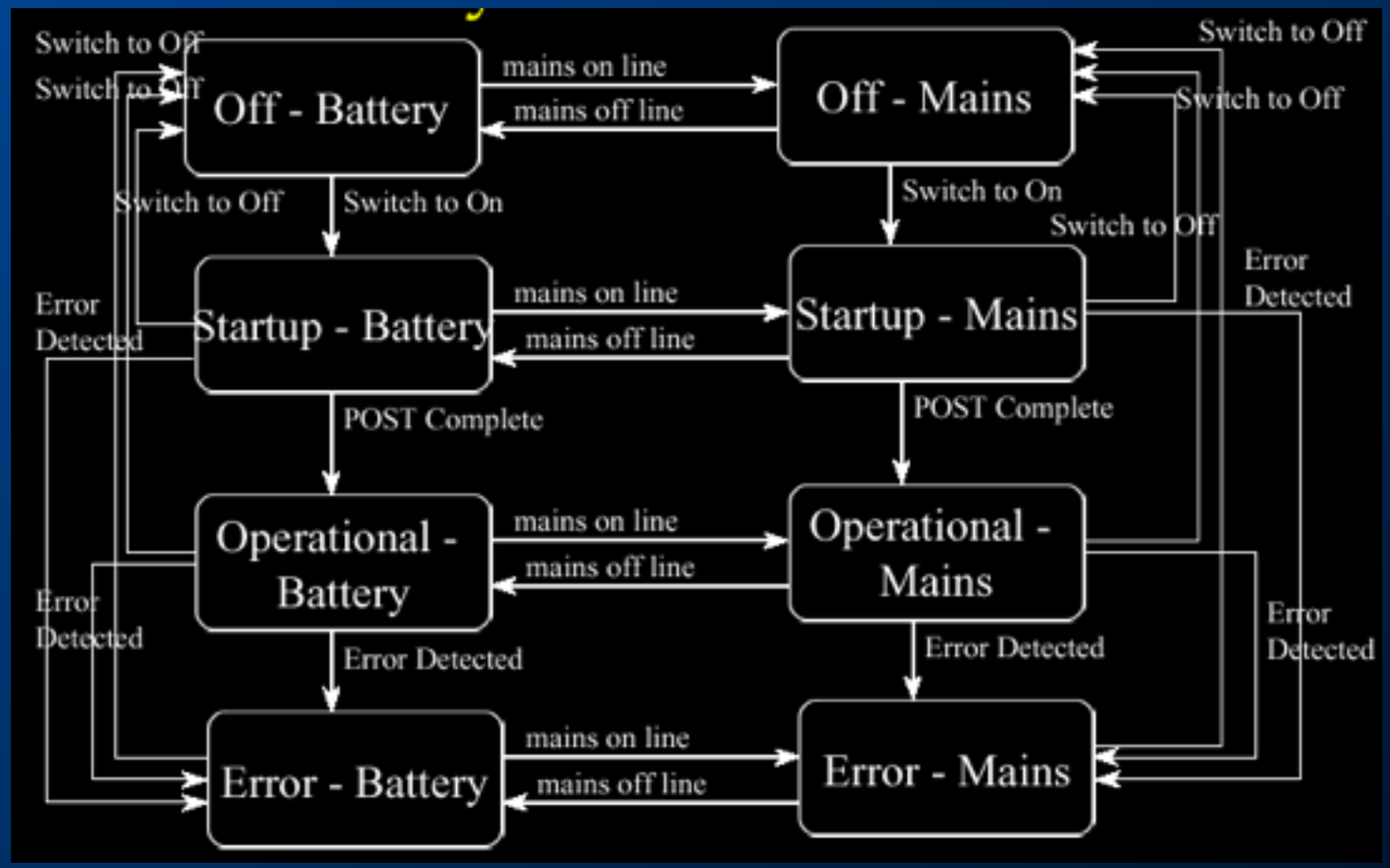
- Example:
  - a device can be in states {Off, Starting-up, Operational, Error}
  - while running from {Mains, Battery}
- How to arrange these states?



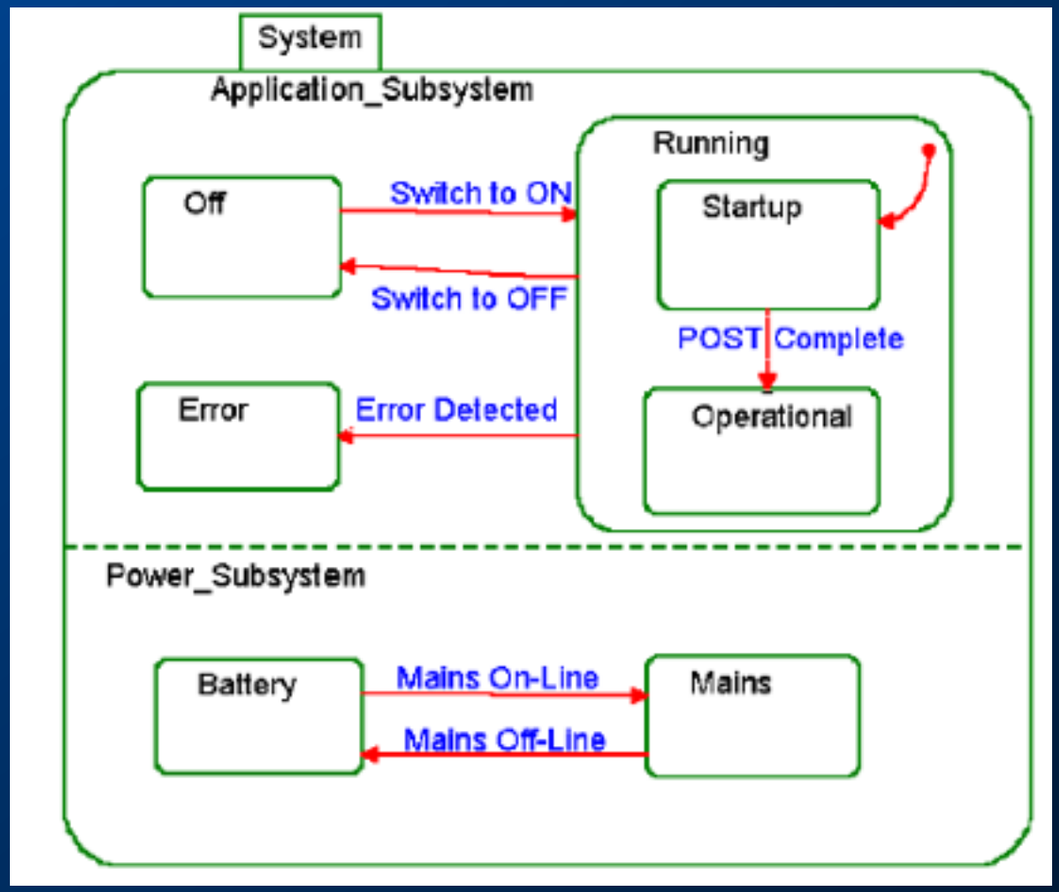
# Concurrency

- Different states in Mealy/Moore view:
  - Operation with battery
  - Operation with mains
- Leads to state explosion
- Solution?
  - Allow states to operate concurrently (Embrace NFA!)

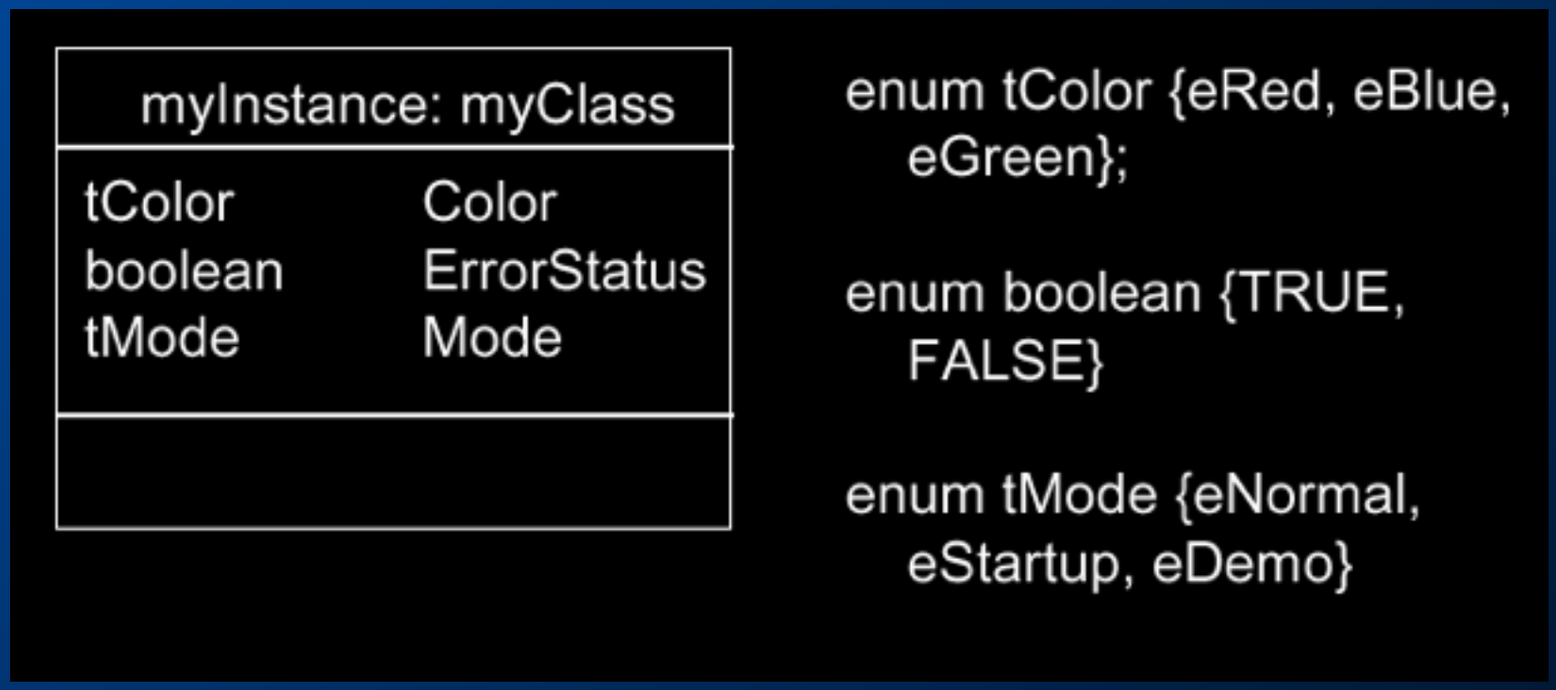
# Mealy-Moore Solution



# State Charts Solution



# Orthogonal Components

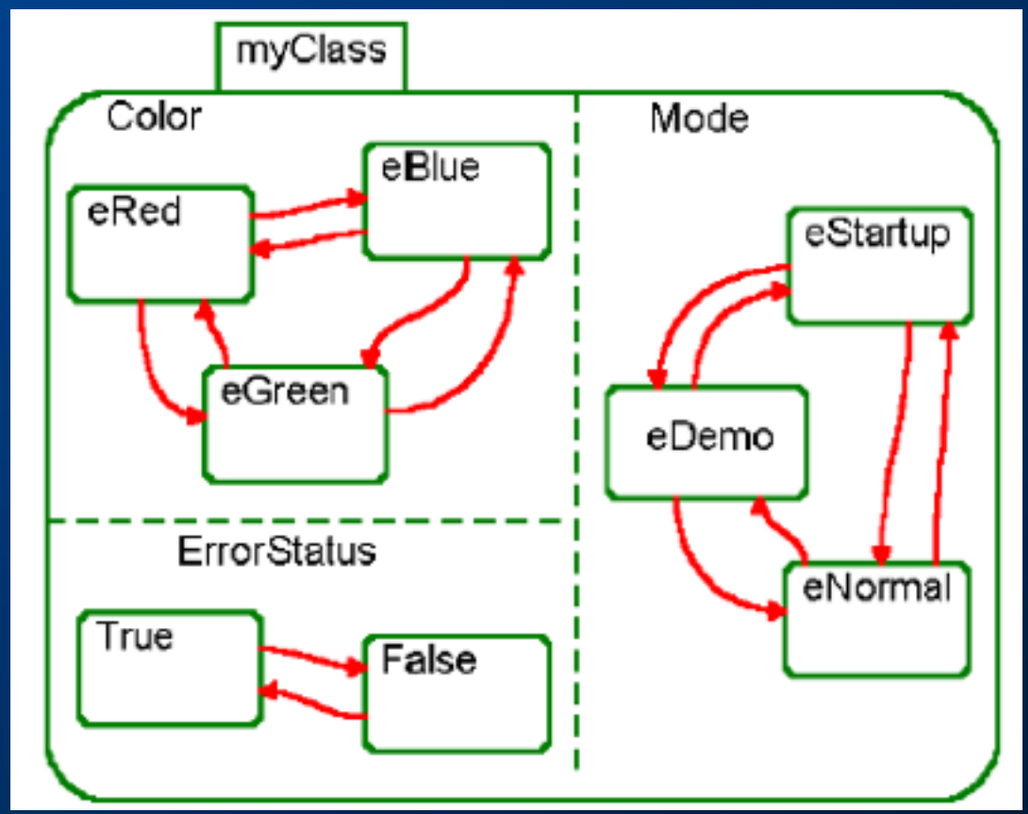



*How do you draw the state of this object?*

# Approach 1: Enumerate all

| | | |
|---|---|---|
| eRed, FALSE, eDemo | eBlue, FALSE, eDemo | eGreen, FALSE, eDemo |
| eRed, TRUE, eDemo | eBlue, TRUE, eDemo | eGreen, TRUE, eDemo |
| eRed, FALSE, eNormal | eBlue, FALSE, eNormal | eGreen, FALSE, eNormal |
| eRed, TRUE, eNormal | eBlue, TRUE, eNormal | eGreen, TRUE, eNormal |
| eRed, FALSE, eStartup | eBlue, FALSE, eStartup | eGreen, FALSE, eStartup |
| eRed, TRUE, eStartup | eBlue, TRUE, eStartup | eGreen, TRUE, eStartup |

# Hierarchical (and-composition) Model

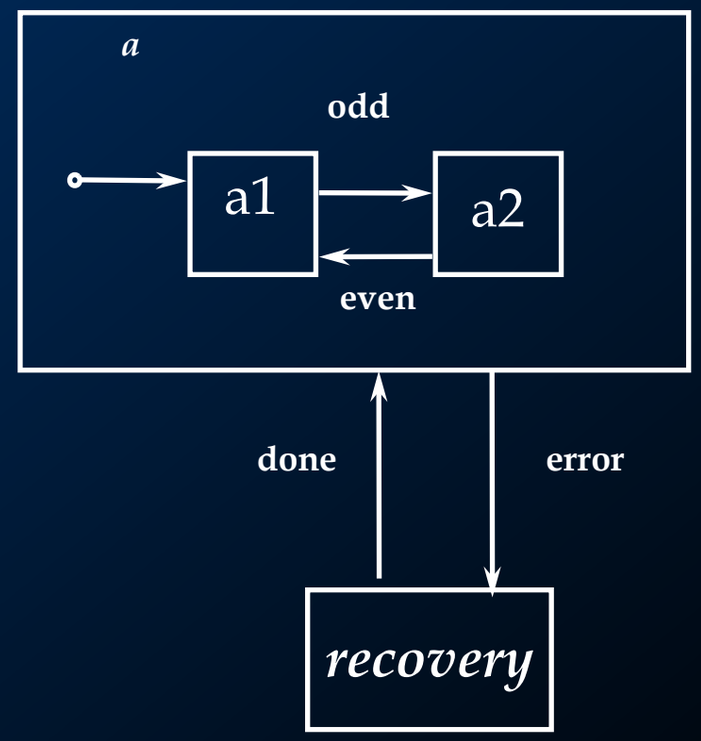


# Harel's StateCharts: Hierarchy of FSMs

- StateCharts support:
  - Repeated decomposition of states into AND/OR sub-states
    - Nested states, concurrency, orthogonal components
  - Actions (may have parameters)
  - Activities (functions executed as long as state is active)
  - Guards
  - History
  - A synchronous (instantaneous broadcast) communication mechanism

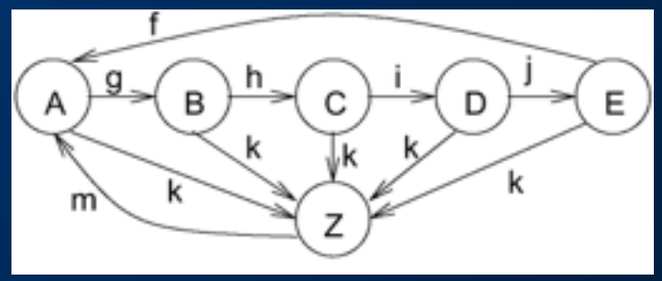
# Hierarchical FSM model

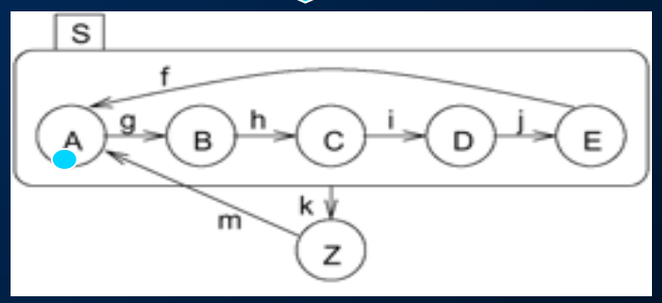
- Problem: how to reduce the size of the representation?
- Harel's classical papers on StateCharts (language) and bounded concurrency (model): 3 orthogonal exponential reductions
- Hierarchy:
  - State *a* "encloses" an FSM
  - Being in *a* means FSM in a is active
  - States of *a* are called OR states
  - Used to model pre-emption and exceptions
- Concurrency:
  - Two or more FSMs are simultaneously active
  - States are called AND states
- Non-determinism:
  - Used to abstract behavior




# Hierarchical States

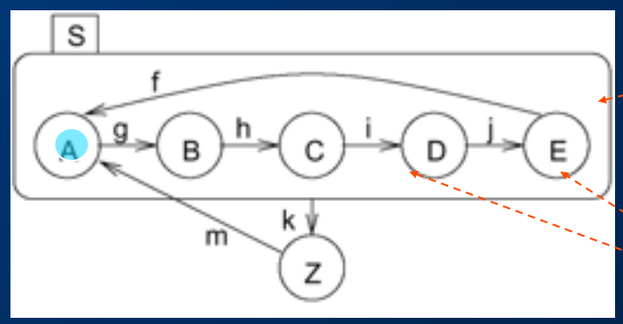
- FSM will be in exactly one of the sub-states of S
- Transitions from and to substates supported by
  - Initial (default)
  - History




# Definitions

- Current states of FSMs are also called active states.
- States which are not composed of other states are called basic states.
- States containing other states are called super-states.
- For each basic state s, the super-states containing s are called ancestor states.
- Super-states S are called OR-super-states, if exactly one of the sub-states of S is active whenever S is active.

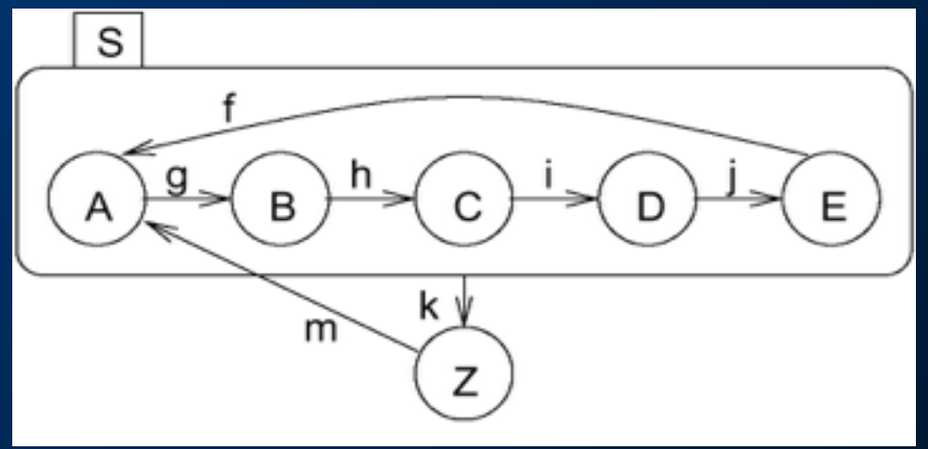


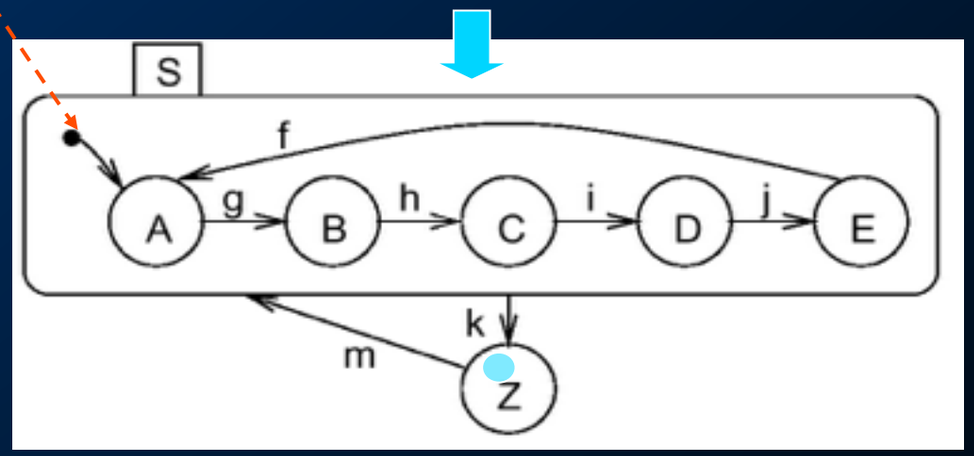

Superstate = ancestor of E

Sub-states

# Default state mechanism

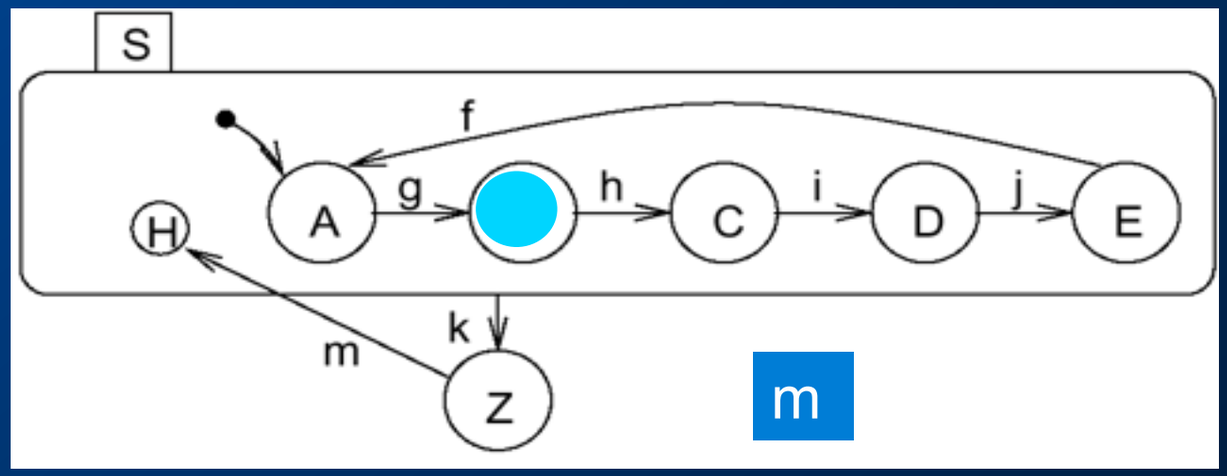
- Default State is a pseudo-state defining a default start state for S
  - Not a state itself



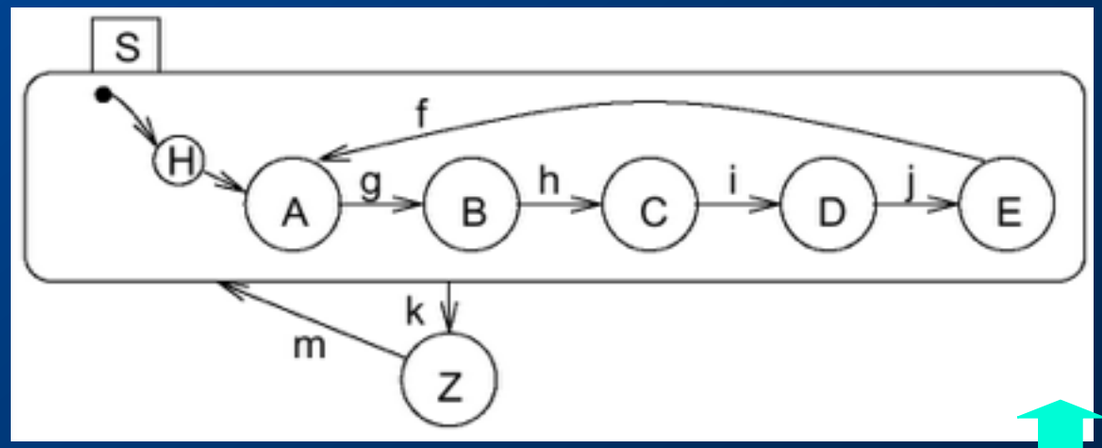

# History Mechanism

- Given input m, S returns to the state it was in before S was left (could be A, B, C, D, or E).
- On first time entry to S, the default mechanism applies.
- History and default mechanisms can be used hierarchically.

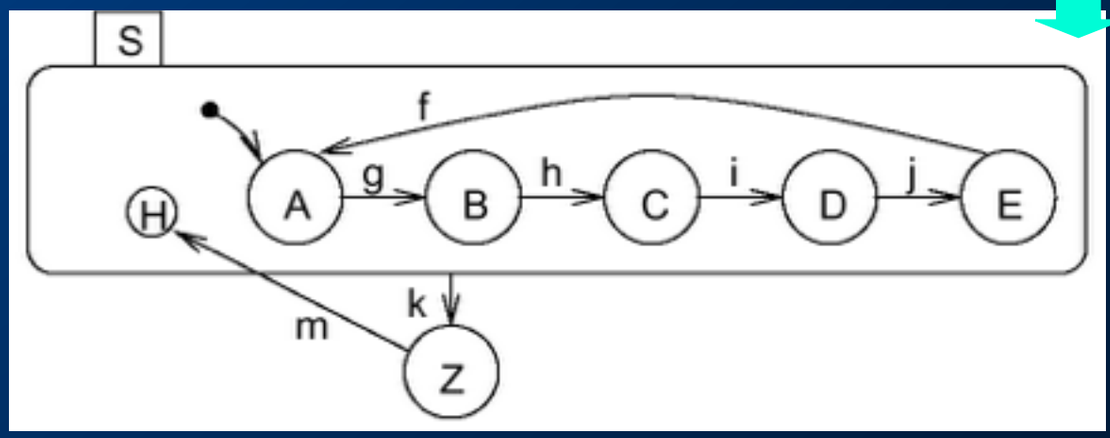



(different behavior from last slide!)

# Combining history and default state mechanism

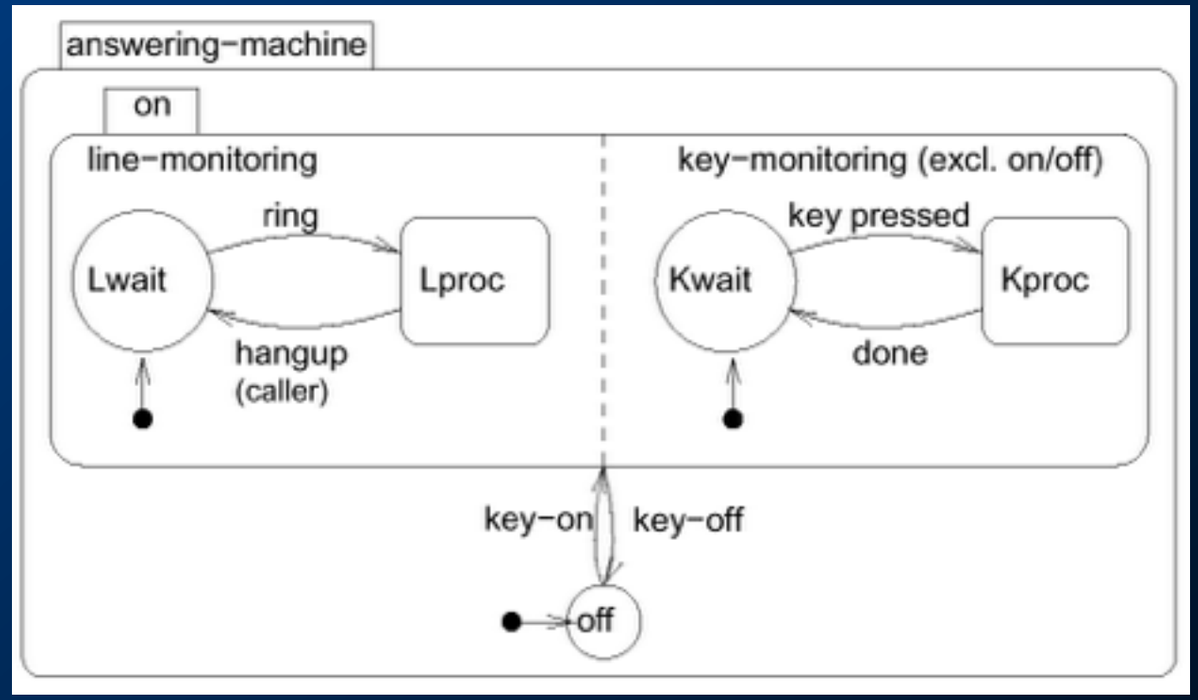



same meaning

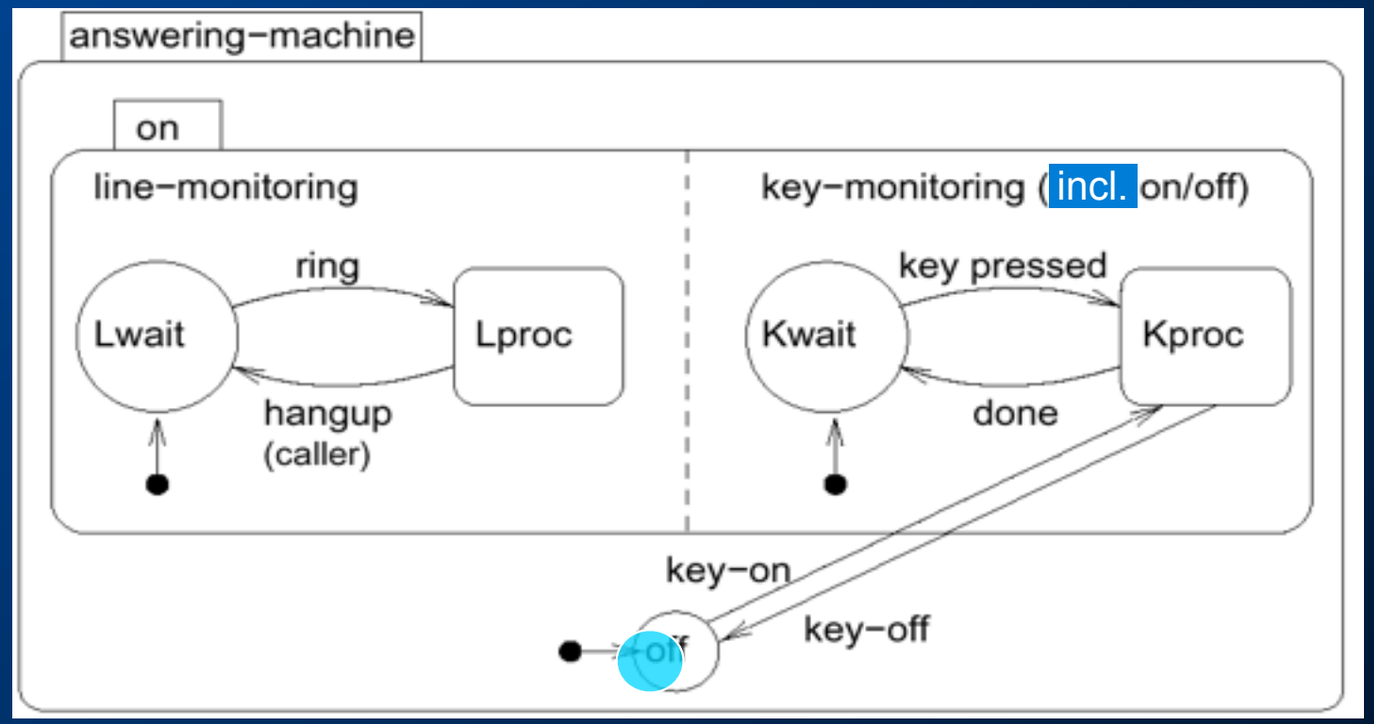



# Concurrency

- AND-super-states
  - FSM is in all (immediate) sub-states of a super-state
  - Unlike Or-Supers, formally require multiple control points



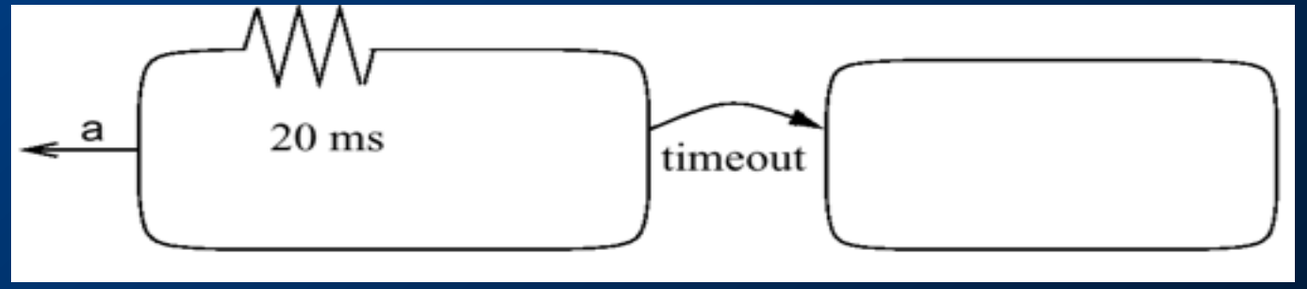
# Entering and leaving AND-super-states




- Line-monitoring and key-monitoring are entered and left, when service switch is operated.

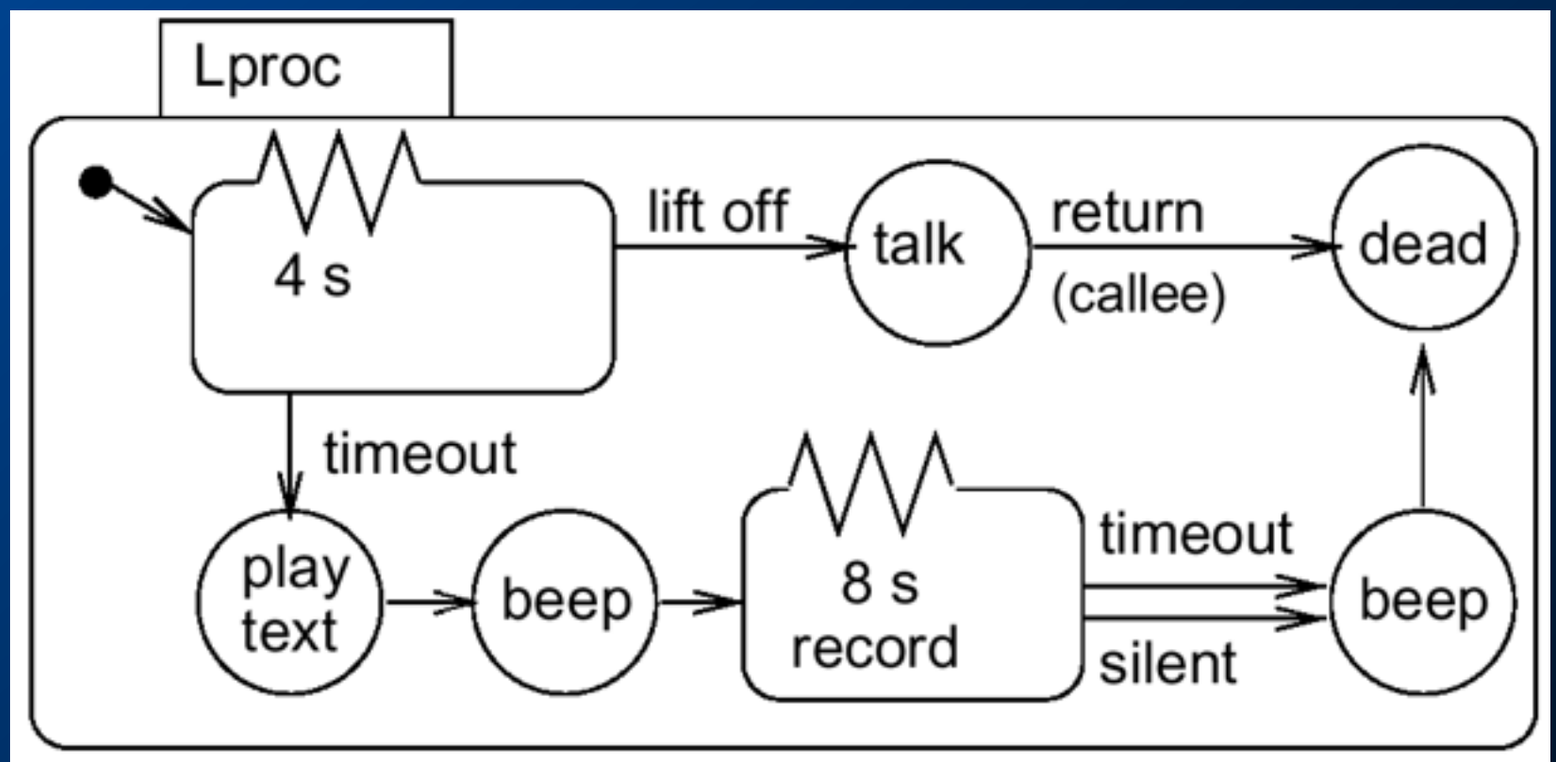
# Timers

- Since time needs to be modeled in embedded systems, timers need to be modeled.
- In StateCharts, special edges can be used for timeouts.




If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

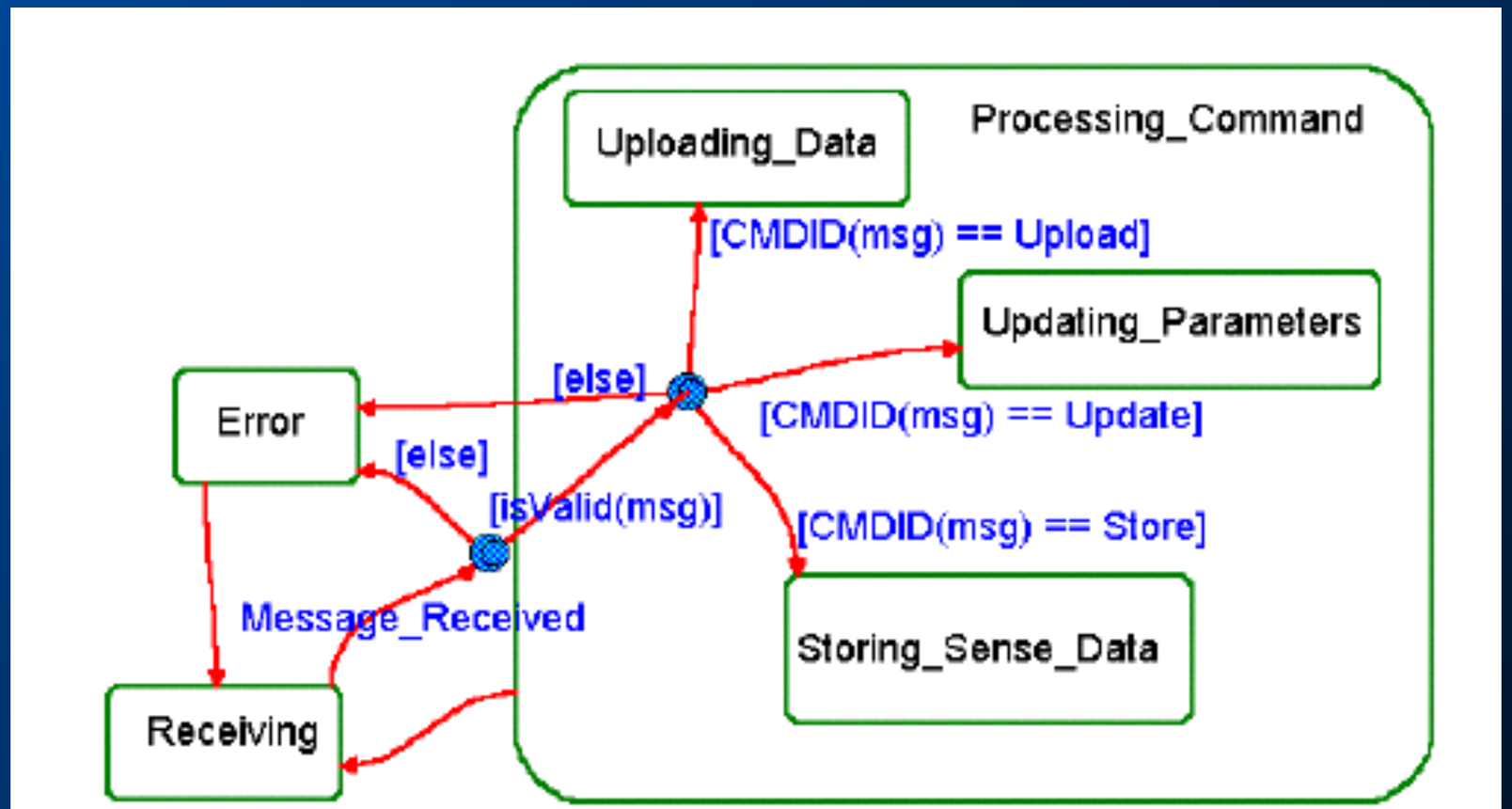
# Answering machine timers



# General edge labels in StateCharts

- The general syntax of an expression labeling a transition in a StateChart is n[c]/a, where
  - n is the event that triggers the transition
  - c is the condition that guards the transition
  - a is the action that is carried out if and when the transition is taken
- For each transition label, event, condition and action are optional
  - an event can be the changing of a value
  - standard comparisons are allowed as conditions and assignment statements as actions


# Conditional Transitions



# Evaluation of StateCharts

- Pros:
  - Hierarchy allows arbitrary nesting of AND- and OR-super states.
  - (StateMate-)Another advantage is that the semantics of StateMate is defined at a sufficient level of detail
  - Large number of commercial simulation tools available (StateMate, StateFlow, BetterState, ...)
  - Available „back-ends“ translate StateCharts into C or VHDL, thus enabling software or hardware implementations.

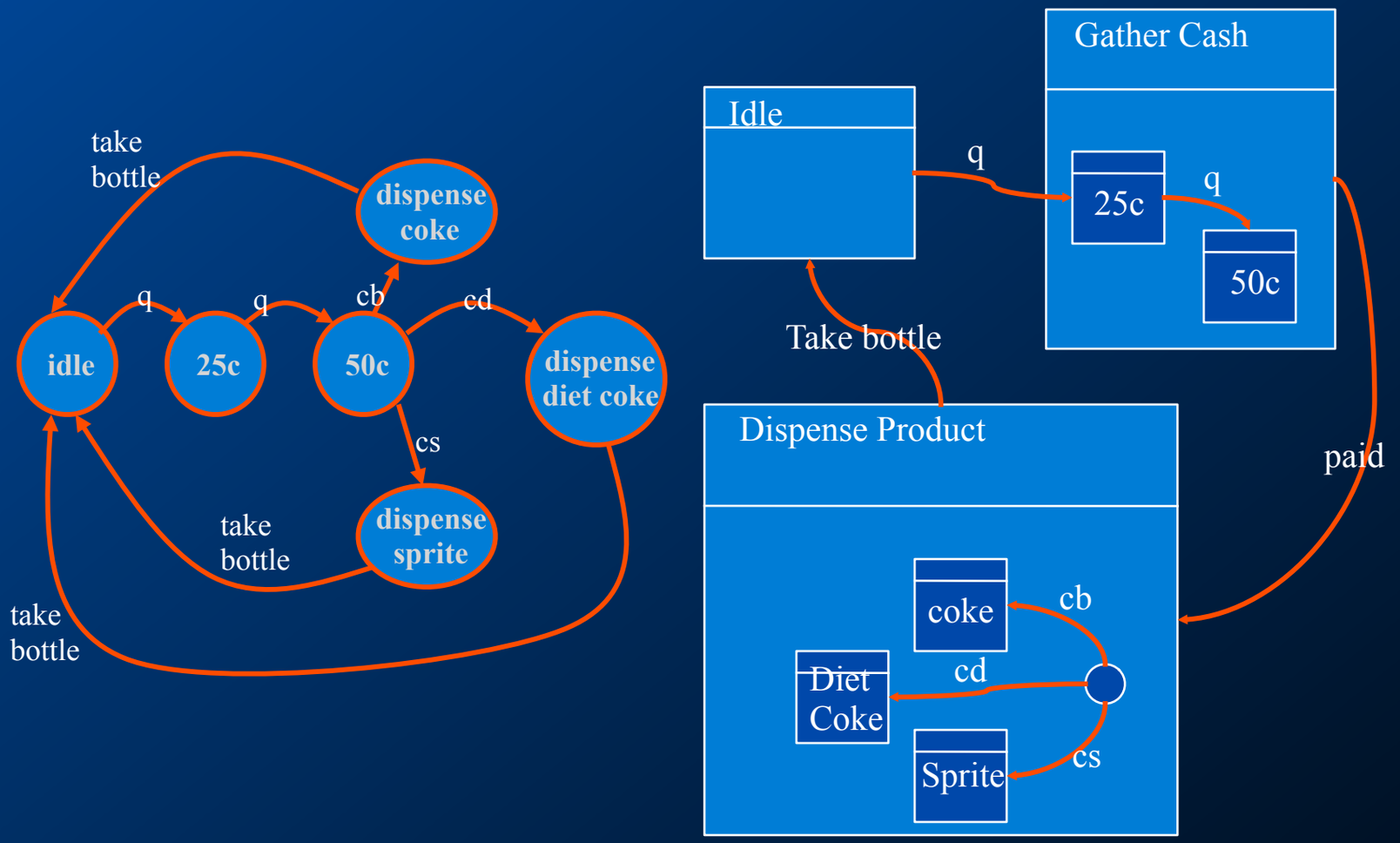
# Evaluation of StateCharts

- Cons:
  - Generated C programs can be inefficient
  - Generated Hardware can be worse
  - Difficult to apply to distributed applications
  - No description of structural hierarchy

# Example: Coke Machine

- Suppose you have a soda machine:
  - When turned on, the machine waits for money
  - When a quarter is deposited, the machine waits for another quarter
  - When a second quarter is deposited, the machine waits for a selection
  - When the user presses "COKE," a coke is dispensed
  - When the user takes the bottle, the machine waits again
  - When the user presses either "SPRITE" or "DIET COKE," a Sprite or a diet Coke is dispensed
  - When the user takes the bottle, the machine waits again

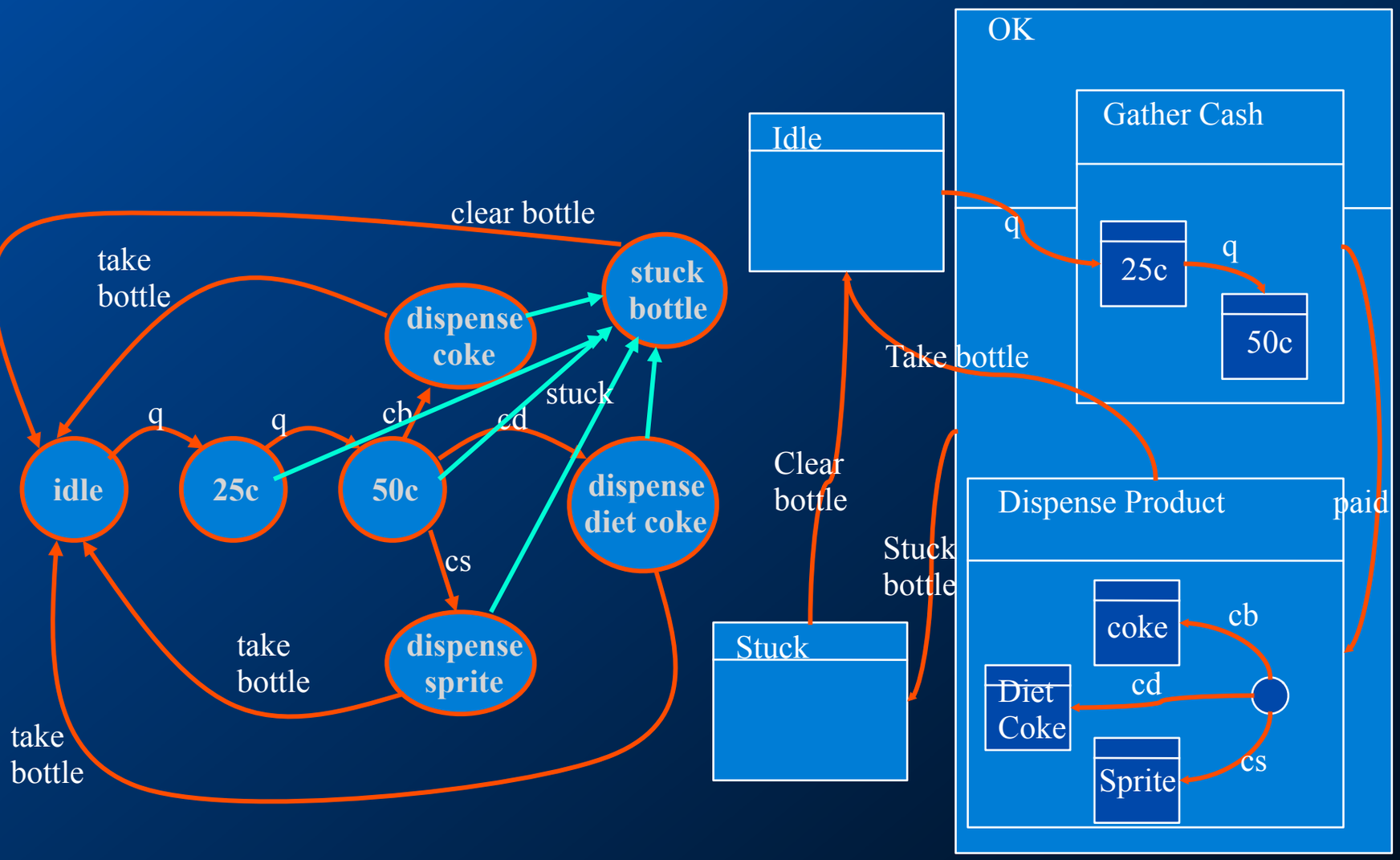
# Coke Machine 1.0




# Coke Machine, Version 1.1

- Bottles can get stuck in the machine
  - An automatic indicator will notify the system when a bottle is stuck
  - When this occurs, the machine will not accept any money or issue any bottles until the bottle is cleared
  - When the bottle is cleared, the machine will wait for money again
- State machine changes
  - How many new states are required?
  - How many new transitions?

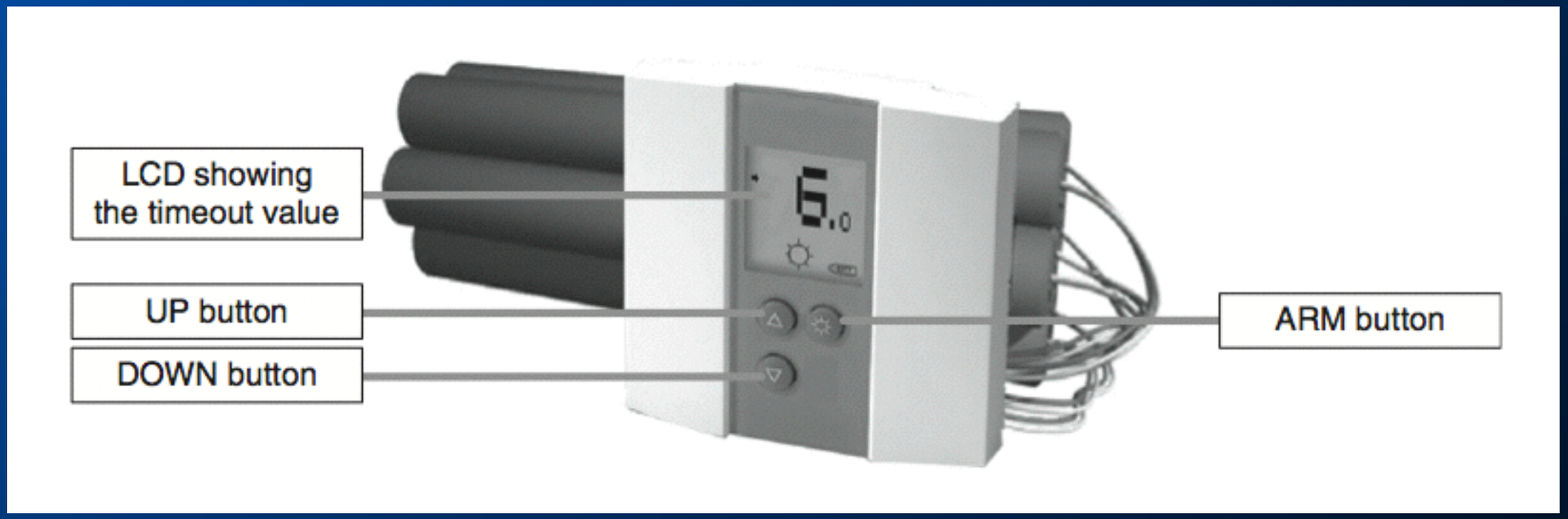
# Coke Machine V1.1




# Hierarchical FSM

- Hierarchy allows for
  - Sensible default activity
  - Multiple places to augment behavior
    - Consider behavior on stuck recovery– eats your money…
  - Large reduction in number of states required
  - Easy to add extended state semantics
    - Augmented state and guarded actions
- How can we make efficient renditions in Software?
  - HFSM are more complex and costly than FSM
    - But provide tradeoff of expression vs. complexity and maintenance

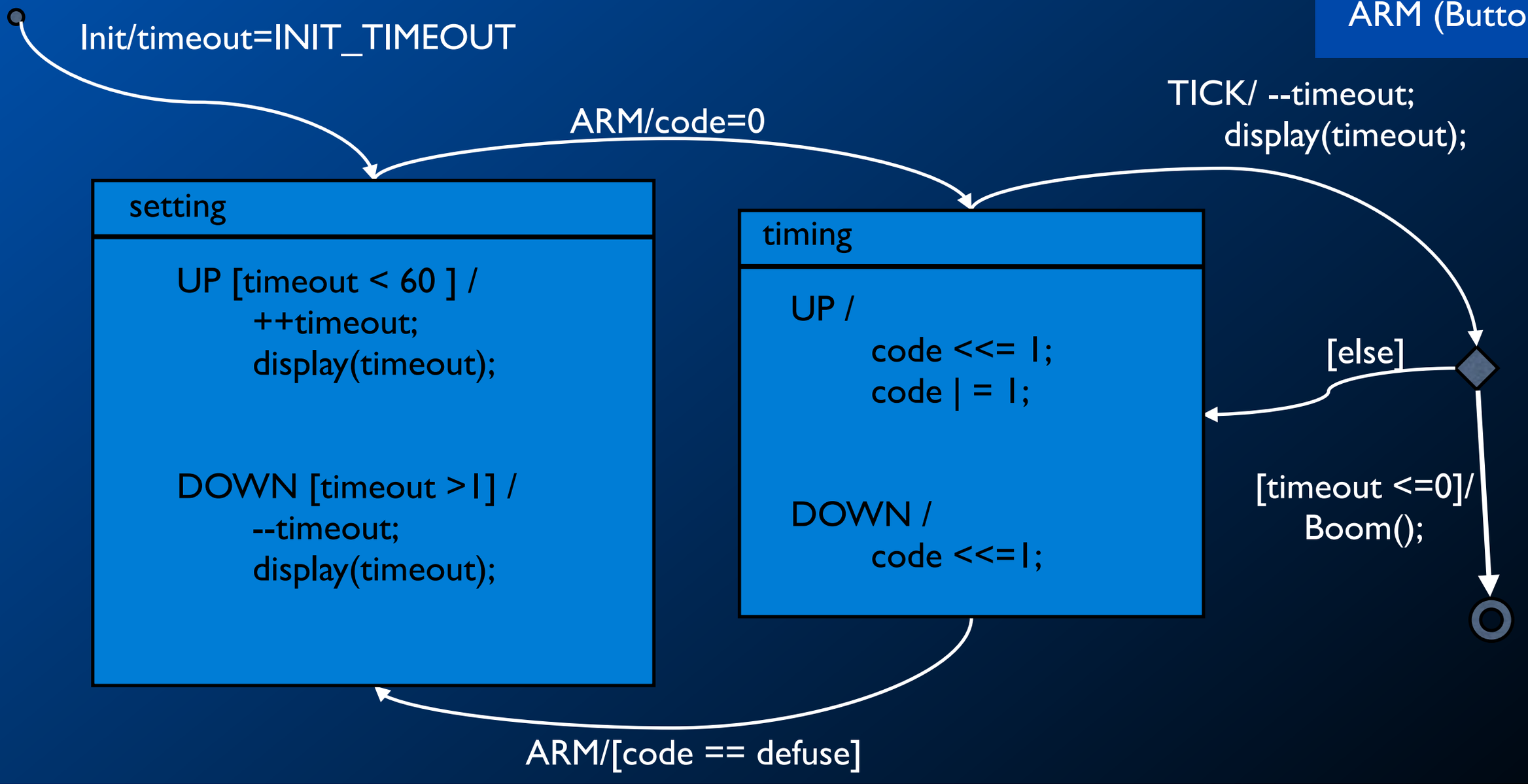
# Time-Bomb Example




Time bomb controller user interface.

# UML State Diagram Representing the Time-Bomb State Machine

Events:
UP (Button)
DOWN (Button)
ARM (Button)




# Nested Switch FSM

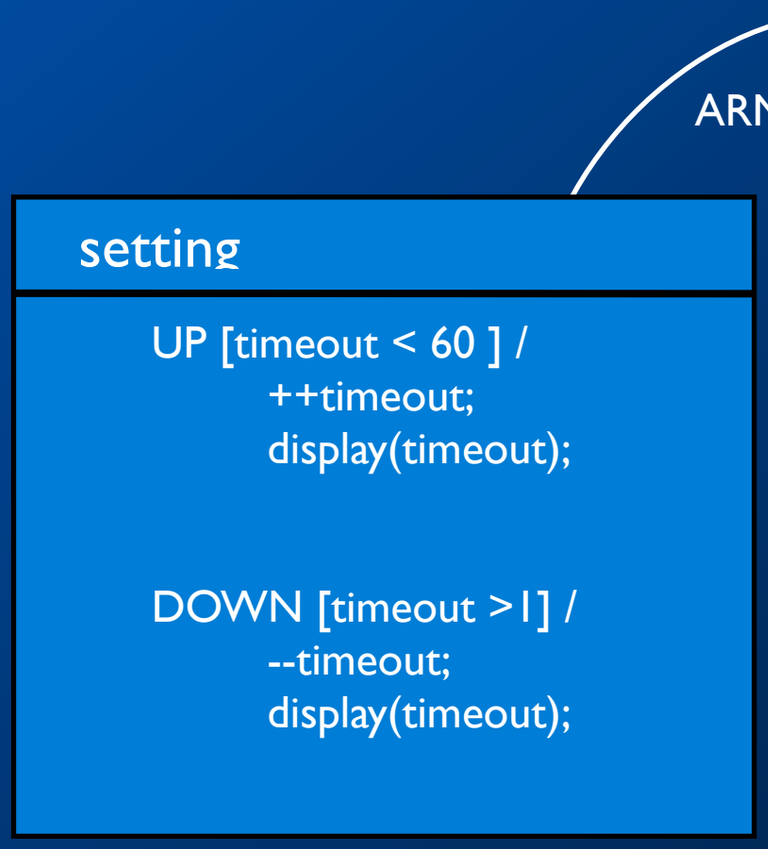
- “State” is an enumeration of a static type
- “event” is a typed signal (Also an enumeration)
- “init” sets up the timers and initializes the variables
- “dispatch” is called on each event to pass control to FSM

# Bomb Example: Declarations

```
enum BombSignals {
    UP_SIG,
    DOWN_SIG,
    ARM_SIG,
    TICK_SIG
};

enum BombStates {
    SETTING_STATE,
    TIMING_STATE
};

typedef struct EventTag {
    uint16_t sig;
} Event;

typedef struct TickEvtTag {
    Event super;
    uint8_t fine_time;
} TickEvt;
```

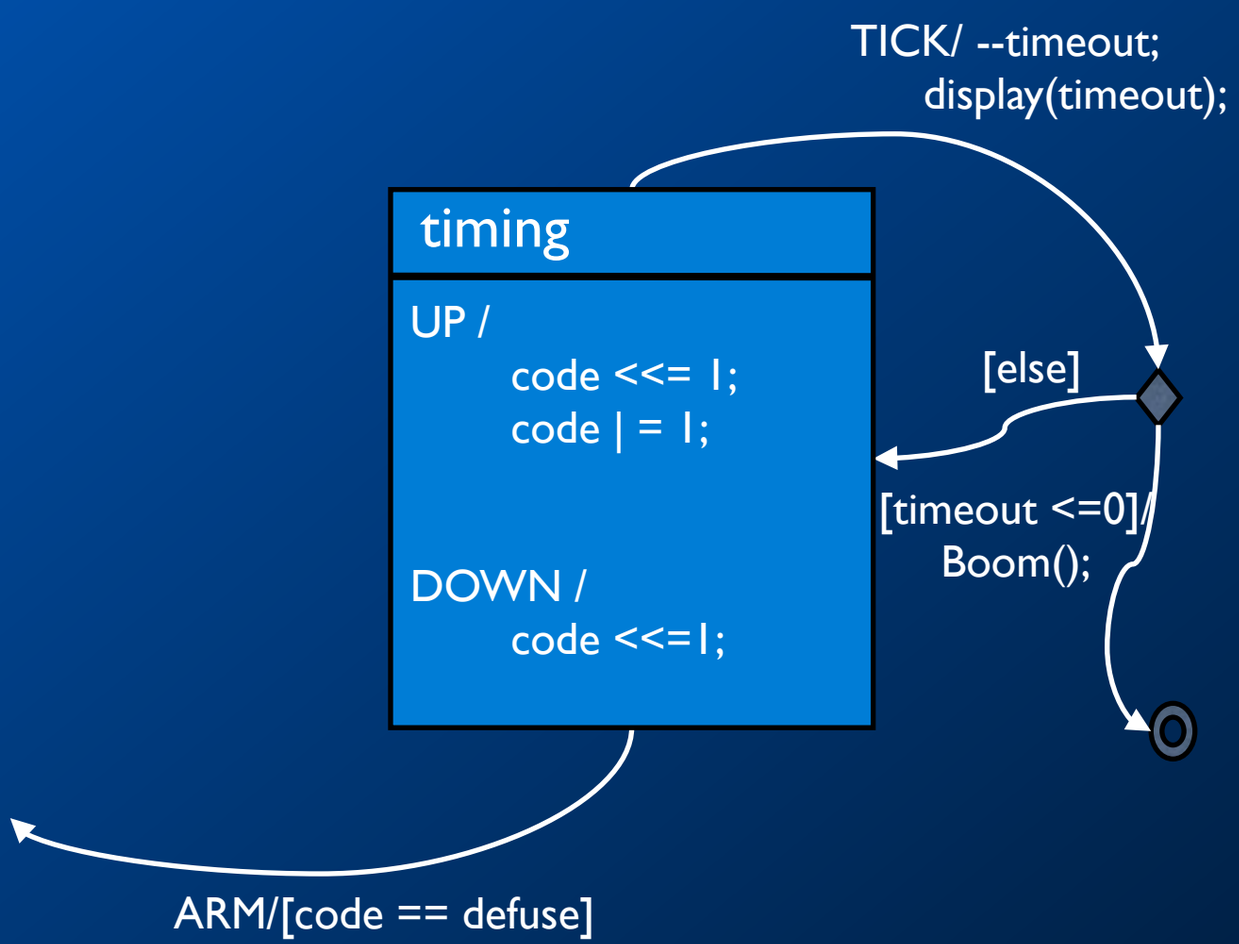
```
typedef struct Bomb1Tag {
    uint8_t state;
    uint8_t timeout;
    uint8_t code;
    uint8_t defuse;
} Bomb1;

void Bomb1_constructor
    (Bomb1 *me, uint8_t defuse);

void Bomb1_init
    (Bomb1 *me);

void Bomb1_dispatch
    (Bomb1 *me, Event const *e);

#define TRAN(target_)
    (me->state = (uint8_t)(target_))
#define INIT_TIMEOUT   10
```

# Bomb Example: Dispatch (Setting State)




```
void Bomb1_dispatch(Bomb1 *me, Event const *e) {
    switch (me->state) {
        case SETTING_STATE: {
            switch (e->sig) {
                case UP_SIG: {
                    if (me->timeout < 60)
                        display(++me->timeout);
                    break;
                }
                case DOWN_SIG: {
                    if (me->timeout > 1)
                        display(--me->timeout);
                    break;
                }
                case ARM_SIG: {
                    me->code = 0;
                    TRAN(TIMING_STATE);
                    break;
                }
            }
            break;
        } /* end case SETTING_STATE */
```

# Bomb Example: Dispatch (Timing State)




```
case TIMING_STATE: {
    switch (e->sig) {
        case UP_SIG: {
            me->code <<= 1; me->code |= 1;
            break;
        }
        case DOWN_SIG: {
            me->code <<= 1; break;
        }
        case ARM_SIG: {
            if (me->code == me->defuse)
                TRAN(SETTING_STATE);
            break;
        }
        case TICK_SIG: {
            if (e->fine_time == 0) {
                display(--me->timeout);
                if (me->timeout == 0)
                    boom();
            }
            break;
        }
    }
    break;
} /* end case TIMING_STATE */
```

# Bomb Example: Setup Code

```
void Bomb1_constructor(Bomb1 *me, uint8_t defuse) {
    me->defuse = defuse; }

void Bomb1_init(Bomb1 *me) {
    me->timeout = INIT_TIMEOUT;
    TRAN(SETTING_STATE); }

static Bomb1 l_bomb;

int main() {
    Bomb1_constructor(&l_bomb, 0x0D);
    for (;;) {
        static TickEvt tick_evt = { TICK_SIG, 0};

        usleep(100000);
        if (++tick_evt.fine_time == 10) {
            tick_evt.fine_time = 0;
        }

        Bomb1_dispatch(&l_bomb, (Event*)&tick_evt);
        if (kbhit())
            Bomb1_dispatch(&l_bomb, e);
    }
}
```

- Init sets Initial transition
- Events on single entry queue
- Event dispatch time requires 2-levels of nested switches
- No clean way to support hierarchy of states since need to replicate code of super-state in many sub-states

# Consequences

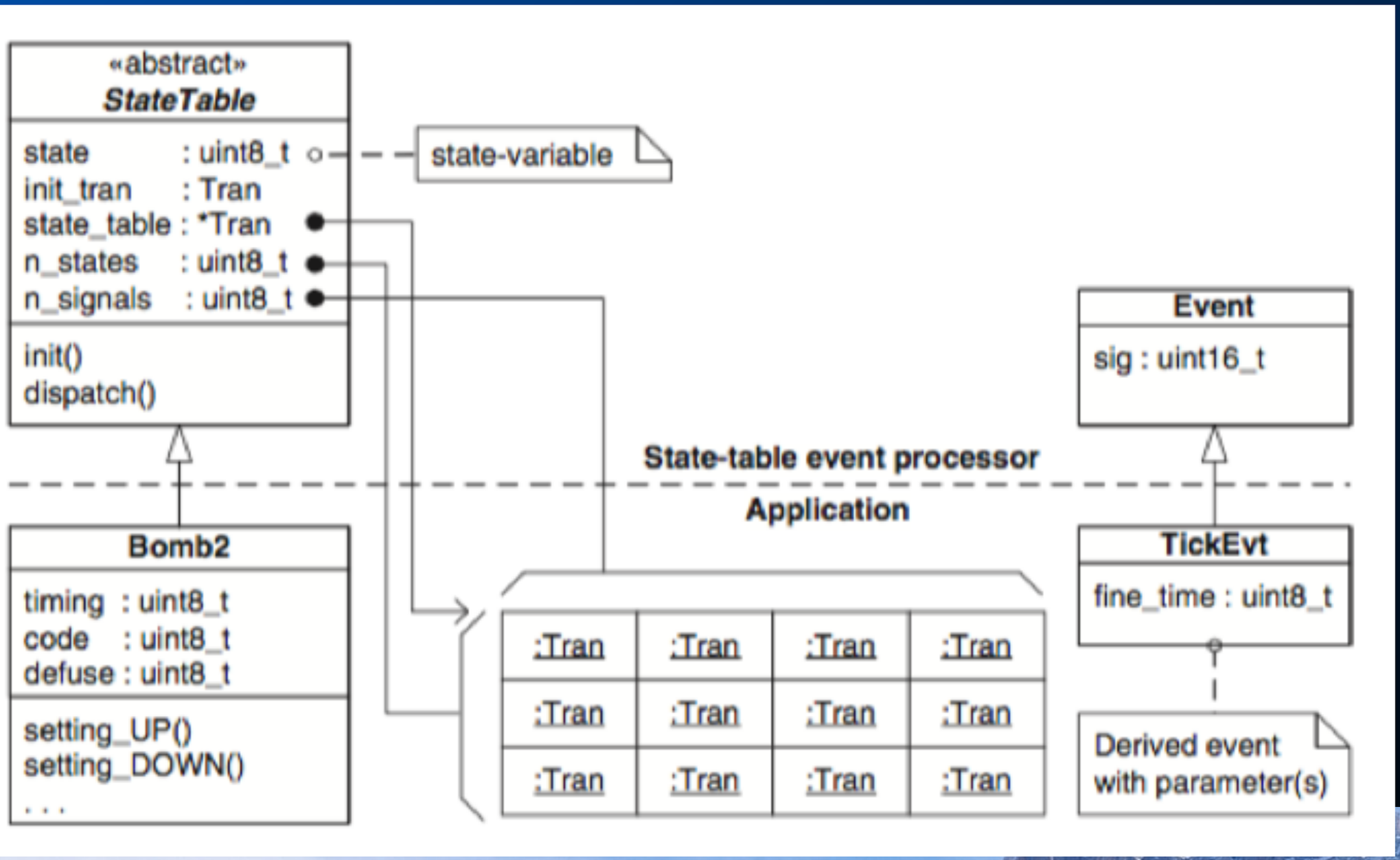
- It is simple.
- It requires enumerating both signals and states.
- It has a small memory footprint, since only one small scalar state variable is necessary to represent the current state of a state machine.
- It does not promote code reuse because all elements of a state machine must be coded specifically for the problem at hand.
- The whole state machine is coded as one monolithic function, which easily can grow too large.
- Event dispatching time is not constant but depends on the performance of the two levels of switch statements, which degrade with increasing number of cases (typically as O(log n), where n is the number of cases).
- The implementation is not hierarchical. You could manually code entry/exit actions directly in every transition, but this would be prone to error and difficult to maintain in view of changes in the state machine topology. This is mainly because the code pertaining to one state (e.g., an entry action) would become distributed and repeated in many places (on every transition leading to this state).
- The latter property is not a problem for code-synthesizing tools, which often use a nested switch statement type of implementation.

# State Table FSM

- Generic Table-Driven Event Processor
- Application provides: Action functions, table, events
- State Table is 2 dimensional:
  - For each event, state the table has (action, next-state)

| | UP | DOWN | ARM | TICK |
|---|---|---|---|---|
| Setting | Set_up(), setting | Set_down(), setting | Set_arm(), timing | Null(), setting |
| Timing | Tim_up(), timing | Tim_down(), timing | Tim_arm(), Setting* | Tim_tick(), Timing* |

# The structure of a generic state table-based event processor



# Table Event Processor: Data Structures

```
typedef struct EventTag { uint16_t sig;
} Event;

struct StateTableTag;

typedef void (*Tran)(struct StateTableTag *me, Event const *e);

typedef struct StateTableTag {
    Tran const *state_table;
    uint8_t state, n_states, n_signals;
    Tran initial;                   /* initial transition */
} StateTable;

void StateTable_ctor(StateTable *me,
        Tran const *table, uint8_t n_states, uint8_t n_signals, Tran initial);
void StateTable_init(StateTable *me);
void StateTable_dispatch(StateTable *me, Event const *e);
void StateTable_empty(StateTable *me, Event const *e);

#define TRAN(target_)  (((StateTable *)me)->state = (uint8_t)(target_))
```

# Table Event Processor: Code

```
void StateTable_ctor(StateTable *me,
        Tran const *table, uint8_t n_states, uint8_t n_signals, Tran initial)
{
    me->state_table = table; me->n_states = n_states;
    me->n_signals   = n_signals; me->initial     = initial;
}

void StateTable_init(StateTable *me) {
    (*me->initial)(me, (Event *)0);
}
```

```
void StateTable_dispatch(StateTable *me, Event const *e) {
    Tran t;

    t = me->state_table[me->state*me->n_signals + e->sig];
    (*t)(me, e);
}

void StateTable_empty(StateTable *me, Event const *e) {
    (void)me;               /* avoid compiler warning */
    (void)e;               /* avoid compiler warning */
}
```

# Table Event Processor: User Code 1

```
enum BombSignals {              /* signals for Bomb FSM */
    UP_SIG, DOWN_SIG, ARM_SIG, TICK_SIG, MAX_SIG    /* the number of signals */
};

enum BombStates {               /* all states for the Bomb FSM */
    SETTING_STATE, TIMING_STATE, MAX_STATE
};
```

```
typedef struct TickEvtTag {
    Event super;
    uint8_t fine_time;
} TickEvt;

typedef struct Bomb2Tag {     /* the Bomb FSM */
    StateTable super;               /* derive from the StateTable structure */
    uint8_t timeout, defuse, code;
} Bomb2;
```

# Table Event Processor: User 2

```
void Bomb2_ctor(Bomb2 *me, uint8_t defuse) {
    /* state table for Bomb state machine */
    static const Tran bomb2_state_table[MAX_STATE][MAX_SIG] = {
        { (Tran)&Bomb2_setting_UP,  (Tran)&Bomb2_setting_DOWN,
          (Tran)&Bomb2_setting_ARM, &StateTable_empty },
        { (Tran)&Bomb2_timing_UP,   (Tran)&Bomb2_timing_DOWN,
          (Tran)&Bomb2_timing_ARM,  (Tran)&Bomb2_timing_TICK  }
    };
    StateTable_constructor(&me->super,
        &bomb2_state_table[0][0], MAX_STATE,
        MAX_SIG, (Tran)&Bomb2_initial);
    me->defuse = defuse;
}

void Bomb2_initial(Bomb2 *me) {
    me->timeout = 10;
    TRAN(SETTING_STATE);
}
```

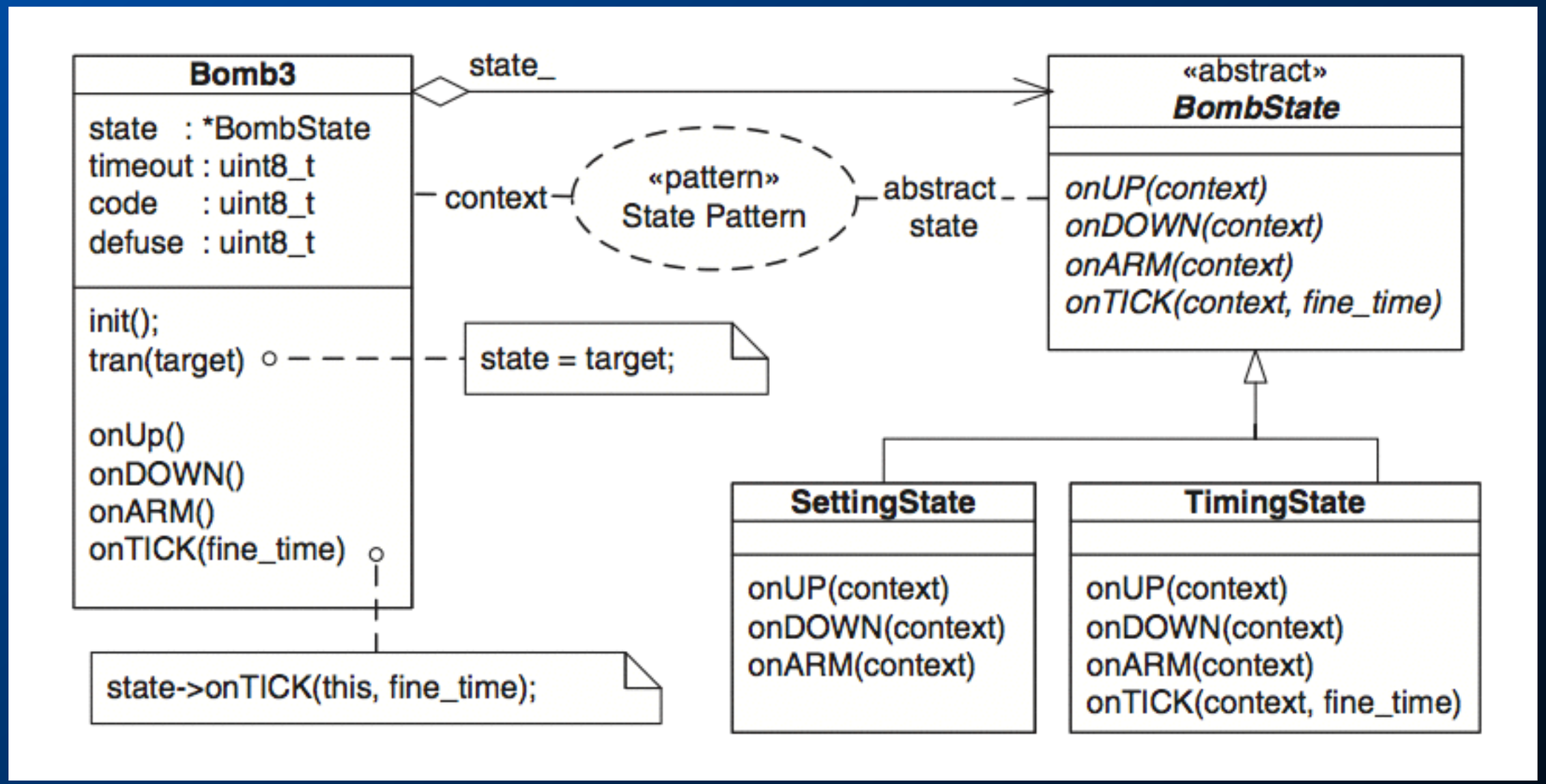
# Table Event Processor: User Actions

```
void Bomb2_setting_UP(Bomb2 *me, Event const *e) {
    (void)e;
    if (me->timeout < 60)
        display(++me->timeout);
}
void Bomb2_setting_DOWN(Bomb2 *me, Event const *e) {
    (void)e;
    if (me->timeout > 1)
       display(--me->timeout);
}
void Bomb2_setting_ARM(Bomb2 *me, Event const *e) {
    (void)e;
    me->code = 0;
    TRAN(TIMING_STATE);
}
…
void Bomb2_timing_TICK(Bomb2 *me, Event const *e) {
    if (((TickEvt const *)e)->fine_time == 0) {
        display(--me->timeout);
        if (me->timeout == 0)
            BSP_boom();
    }
}
```

# Table Event Processor: Issues

- Regular Structure for FSM
  - Enumerations used as indices
  - Need “filler” states and functions
    - Some state,event pairs are empty
- Dispatch is Constant time
- State Table is ROM candidate, but is sparse
- Change potentially requires rebuilding entire Table
- Every Action requires own function
- Does not support State Hierarchy

# Object-Oriented State Design Pattern




# Consequences

- It relies heavily on polymorphism and requires an object-oriented language like C ++.
- It makes state transitions efficient (reassigning one pointer).
- It provides very good performance for event dispatching through the late binding mechanism (O(const), not taking into account action execution).
- It allows you to customize the signature of each event handler. Event parameters are explicit, and the typing system of the language verifies the appropriate type of all parameters at compile time (e.g., onTICK() takes a parameter of type uint8_t).
- The implementation is memory efficient.
- It does not require enumerating states,  enumerating events.
- It enforces indirect access to the context's parameters from the methods of the concrete state subclasses (via the context pointer).
- Adding states requires subclassing the abstract state class.
- Handling new events requires adding event handlers to the abstract state class interface.
- The event handlers are typically of fine granularity, as in the state table approach.
- The pattern is not hierarchical.

# Better Table Event Processor: QEP (Samek)

- Idea: Keep the extensible notion of Tables, but eliminate the 2-d lookup
  - Use Switch statements to discriminate events for each state
  - Use table of pointers to state handler functions to discriminate states
- Benefits:
  - Table dispatch is faster than nested switch, but switch only needs to dispatch ***handled*** events
  - New states can be easily added, as can new events
    - Only need to change parts of implementation that are directly effected…

# References

- 嵌入式系统导论：CPS方法。
- Practical UML STATECHARTS in C/C++, Second Edition. Chapter 3

# Thank you!