Lecture 3
Finite State Automata Models

Hierarchy, Abstraction, Implementation

Forrest Brewer

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 transitions only depend on the current state, the machine is a *Clock*.
  - 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.

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
- Rarely suitable for Memory and Datapaths
  - Little abstraction power for large alphabets
- Powerful algorithms for verification
- Easy to synthesize, but can be inefficient

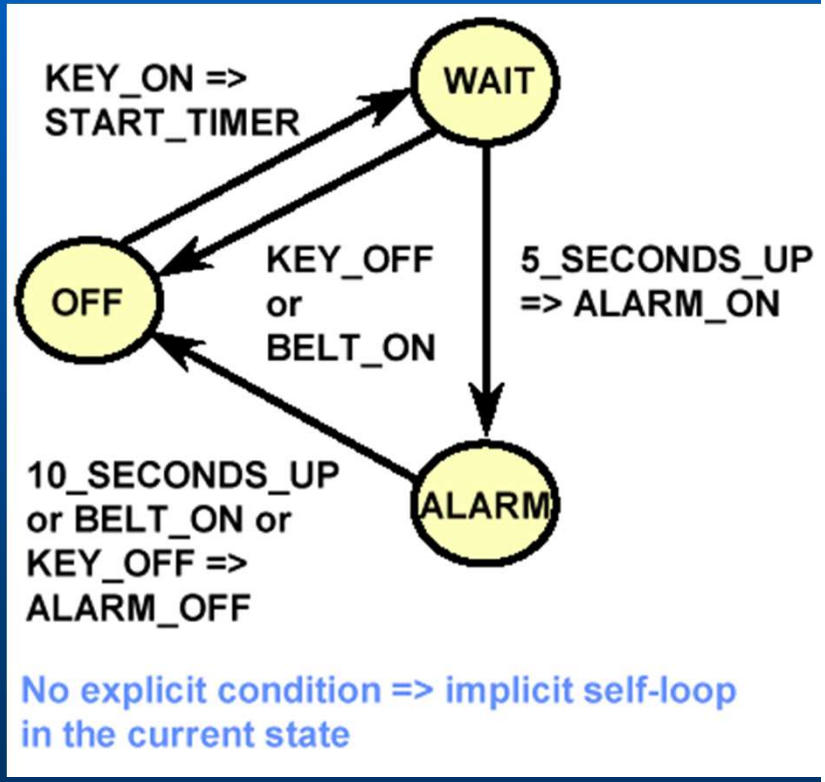
FSM Example Model

- **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
```



Standard FSM Nomenclature

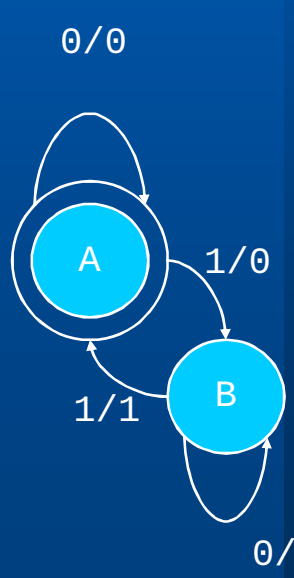
- Finite automata behavior classified by properties of the set of states, and the transition relation (next_states, output) = Φ(state, event) which describes possible next states and outputs after an event
- Finite Automata Classifications
  - "Deterministic" means that Φ(S,E) is single valued
  - "Completely Specified" means that Φ(S,E) has a value for every possible input
  - "Sound" means that outputs for a given (state, event) don't conflict – I.e. no state says to turn the light on and off at the same time
  - "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*

Black Box view of FSM

- The “behavior” of an FSM is determined by what output it makes after a sequence of known inputs assuming it starts from a known state
  - A sequence of inputs that cause two machines to output different symbols is called “distinguishing” for the two machines.
  - For machines with a single 1 or 0 output, the set of all input symbol sequences that produce “1” form a possibly infinite set S_1, similarly “0” has a set S_0. For a well-defined machine, all possible input sequences belong to one of these two sets.




```
Input: 0001011010010101010001110101000101010101010…
State: AAABBABBAAABBAABBAAAABABBAABBBBAABBAABBAABBAA…
Output:0000010010000100010000100100000100010001000010…

“recognizes all strings with an even number of 1’s”
```

Network View of FSM

- Can also find FSM behaviors tied to properties of the graph model of the transition relation $\Phi(s,e)$
- A state is re-entrant iff the FSM's graph is strongly connected
- A state s is reachable from t iff there is a path from s to t where each state visited along the path has a valid transition to the next state on the path
  - Corr: for a finite automaton, the shortest path from s to r cannot be longer than the number of states
  - If no path exists from s to r or from r to s the machine is said to be "disconnected".
  - If a path exists from r to s, but not from s to r for some s and r, the machine is not "resetable", I.e. some behaviors can only be observed once (consider the Blue Screen of Death).
- Usually, we are interested in machines with a subset of states that are strongly connected

Languages and Regular Automata

Consider the following recursive definition of a language:

- Any symbol s from the finite set S is a member of the language
- Any string t formed by concatenating two strings of the language is a member of the language
- Any string r formed by choosing one of two strings s or t, both of which are languages is a member of the language
- Any string r formed by zero or more concatenated copies of a member s is also a member of the language
- Corr: an empty string ε is a member of the language

e.g. 0*1*(01)* is all strings like: 00111010101… or 010101 or 111…
but not 10 or 100…

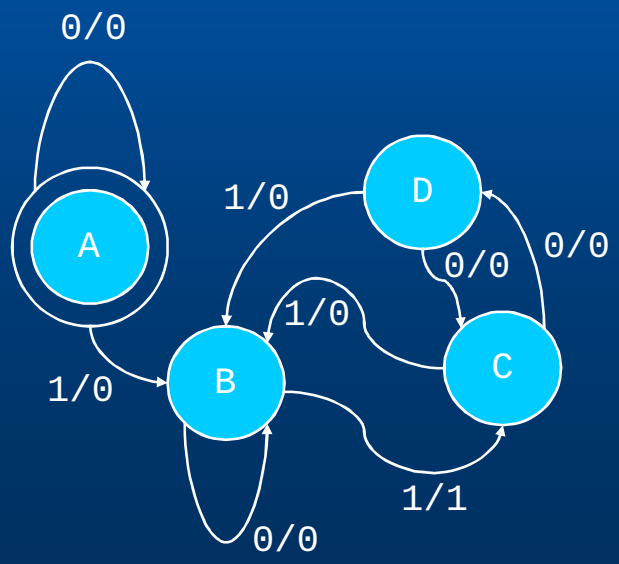
This language defines Regular Automata

FSM can act as string recognizers

- Kleene proved the strong result that every regular automata has a finite state recognizer (in fact he showed that FSM and regular languages are equivalent).
- Not every language has a finite recognizer: Consider the set of all palindrome strings of length 2k. In general, you need memory of at least k elements to determine if the palindrome holds => not finite state for unbounded length strings.
- However, Regular Automata languages are very important for embedded systems with bounded memory – and for virtually all communication protocols and encodings since they can be recognized and implemented with a bounded circuit or static memory program.

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!



Also recognizes even Numbers of 1's in a Binary string.

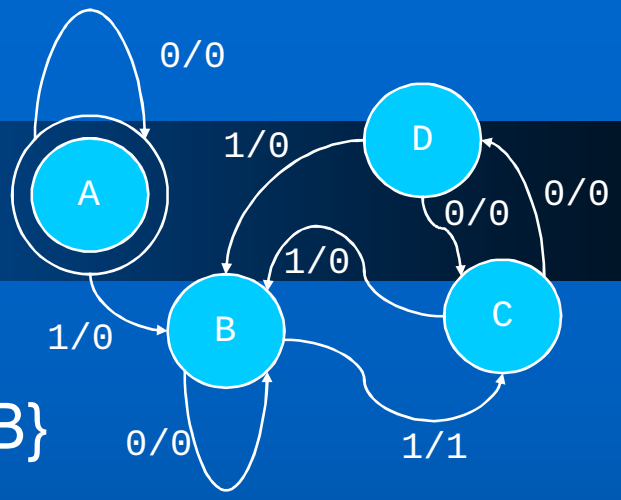
Equivalence allows for optimization

- 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.

Overview of Minimization

- Proceed by iteration on the length of distinguishing sequences of inputs– equivalence relation implies that state sets will form partitions of the states
  1. Divide the states into sets that are output distinguishable (i.e. different outputs for same input). This set of state sets is labeled S_1 and each subset of S_1 is labeled S_{1i} for each different output i.
  2. Consider the states in each of the sets of S_{1i}. Given an input, determine the next state for all states in S_{1i}. If all the next states fall in a common subset S_{1j} then that input does not help distinguish those states. After checking all inputs for a subset, add it to the refined partition S_2.
  3. If the next states occur in different subsets of states in S_1, they can be distinguished in two steps. Partition the subset S_{1i} into smaller subsets having next states in common subsets S_1. Add these new smaller subsets into S_2.
  4. Continue with new inputs and new subsets until all subsets have been visited.
  5. Iterate steps 2-4 updating the indices until no new smaller sets are created.
  6. At this point, all states in a subset are indistinguishable by an input sequence of any length. For each subset, chose a new state name and infer transitions by looking at any of the states in the partition.

Example Minimization



- 1. Ouput Disjoint states: {A, C, D} {B}
  - Note B outputs 1 on input 1
- 2. Input 0: A->A, B->B, C->D, D->C thus {A, C, D} all go to one of {A, C, D}
  Input 1: A->B, B->C, C->B, D->B again {A,C,D} behave the same.
- 3. No new partitions were needed so {A,D,C} are equivalent states. Re-label {A,D,C} as S and {B} as T
- 4. Derive S,0 -> S,0; S,1 -> T,0; T,0 -> T,0; T,1 -> S,1

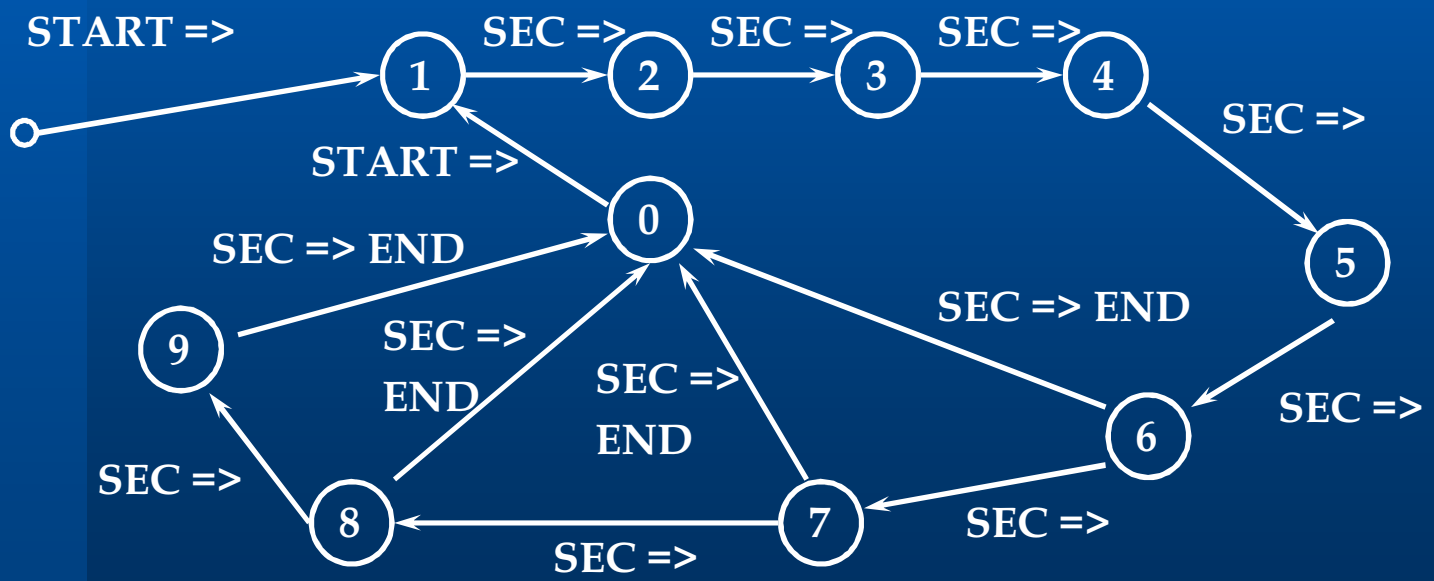
Non-Deterministic FSM

- 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

NDFSM (NFA): time range

- Special case of unspecified/unknown behavior, but so common to deserve special treatment for efficiency

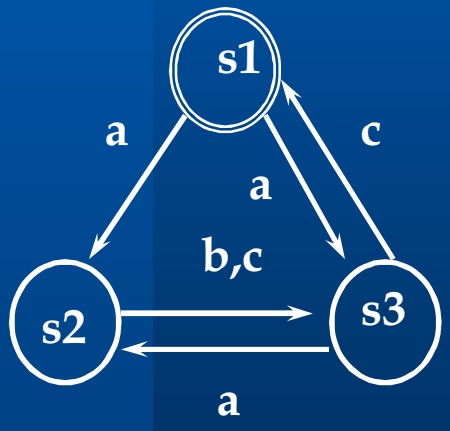
E.g. delay between 6 and 10 s

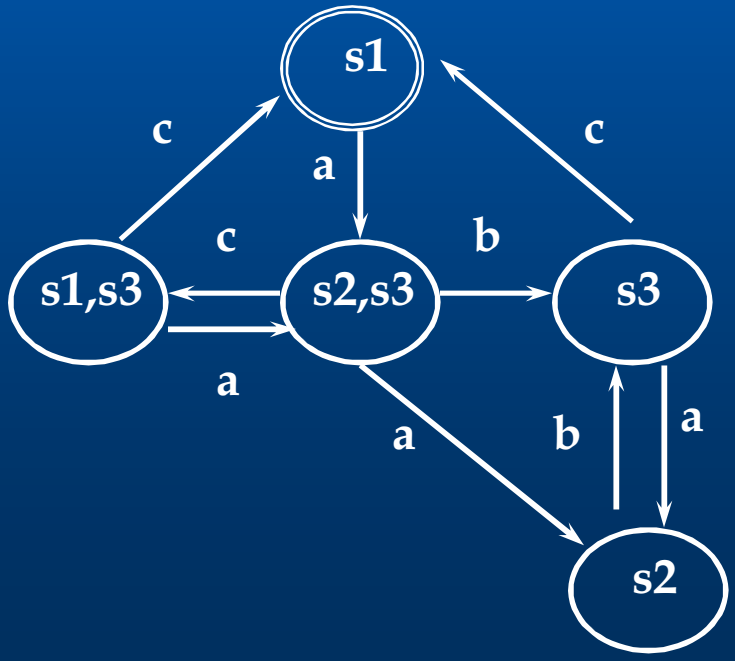


Are NFA and FSA equivalent?

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
  - Difficult to compose (outputs inherit input timing issues)

Hierarchical FSM

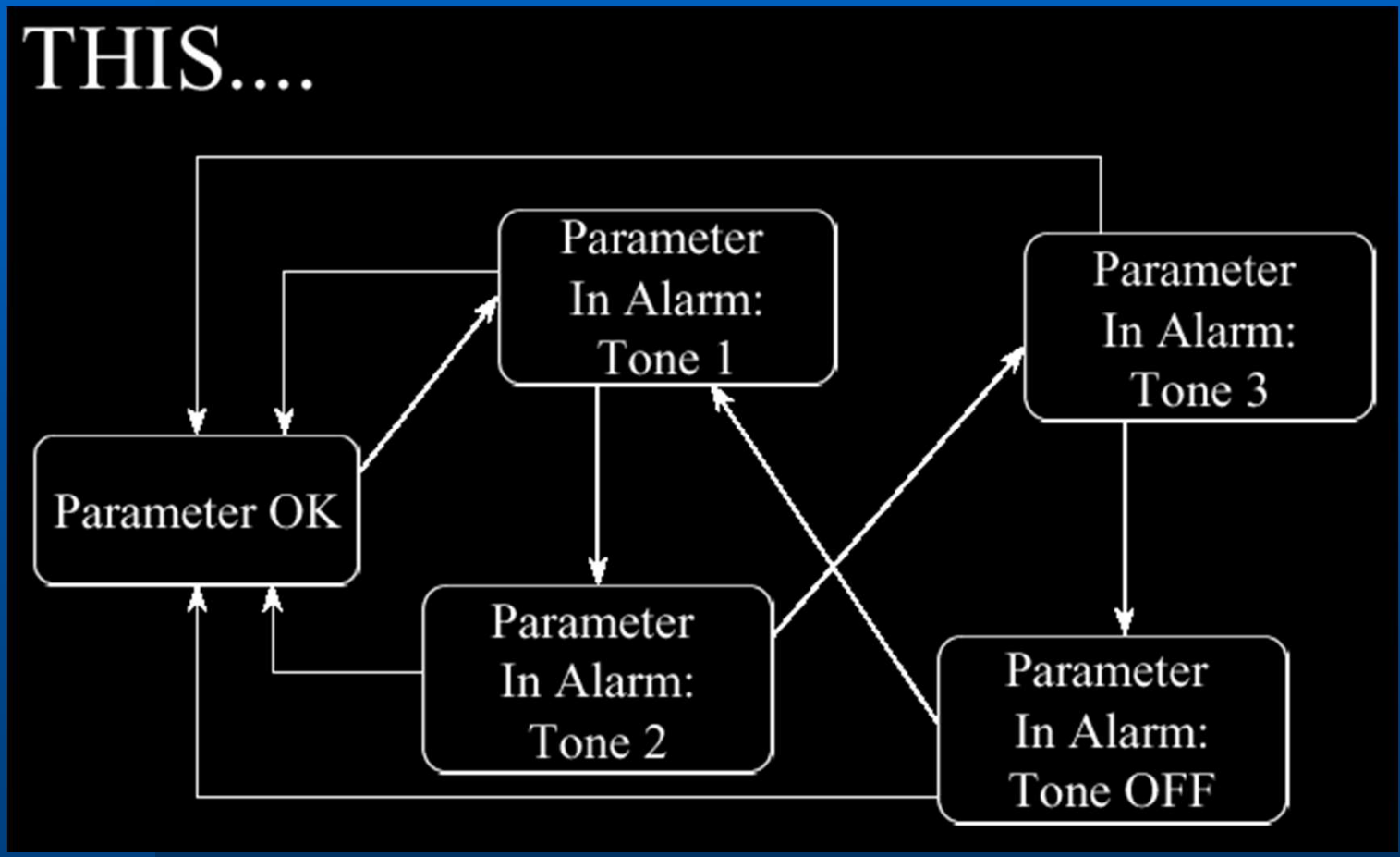
- Support Concurrency and Hierarchy
- Natural rendition for software behavior
  - Same issues as MEALY for hardware designs
  - Irrelevant for Software, leads to more compact code

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

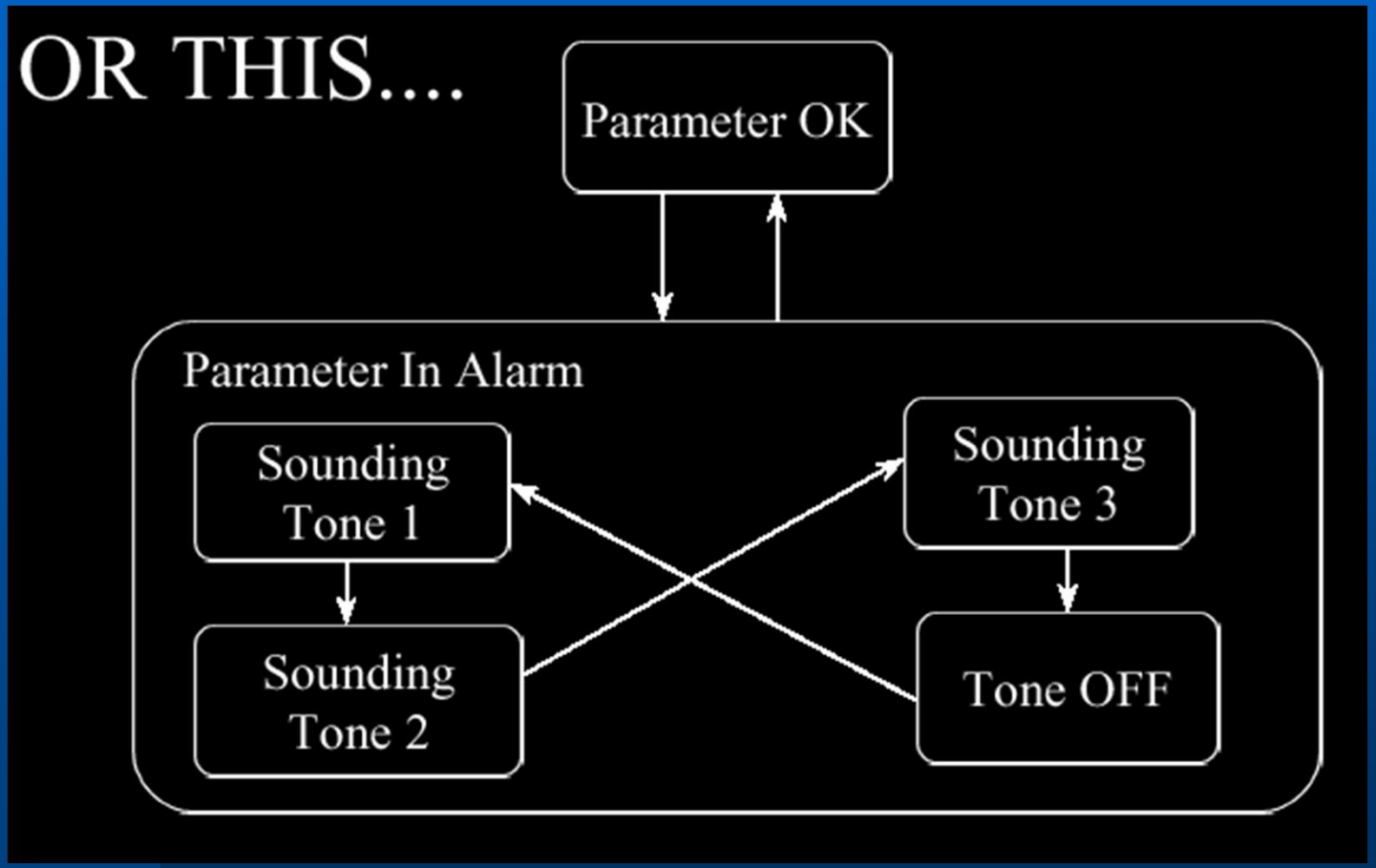
Source: B. P. Douglass & iLogix

Hierarchy



Source: B. P. Douglass & iLogix

Hierarchy

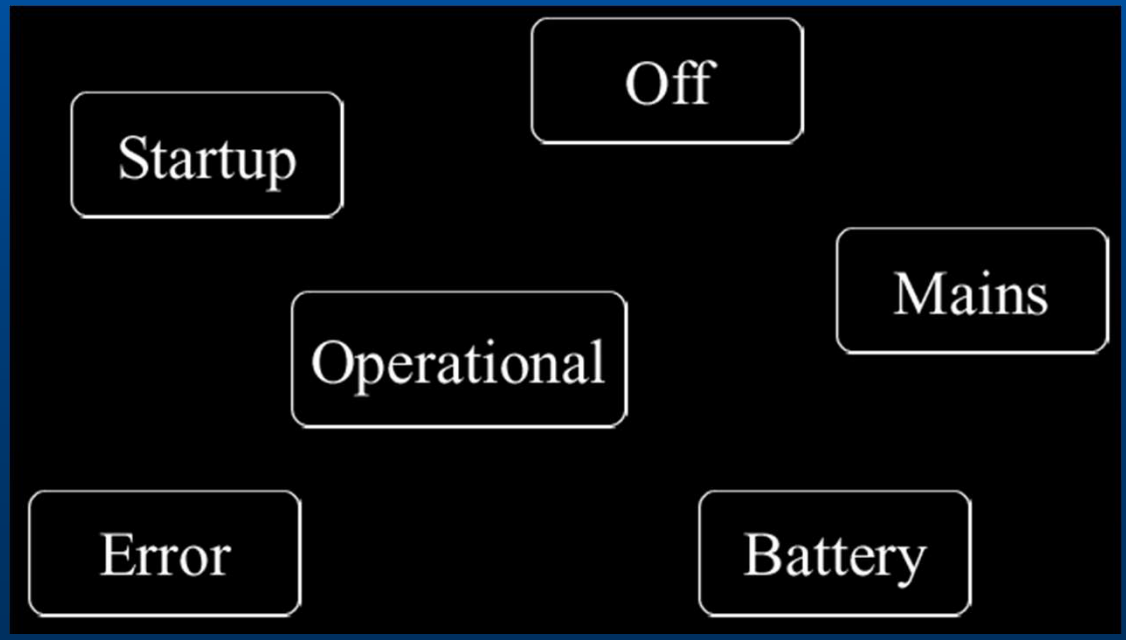


Source: B. P. Douglass & iLogix

Source: B. P. Douglass & iLogix

Concurrency

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

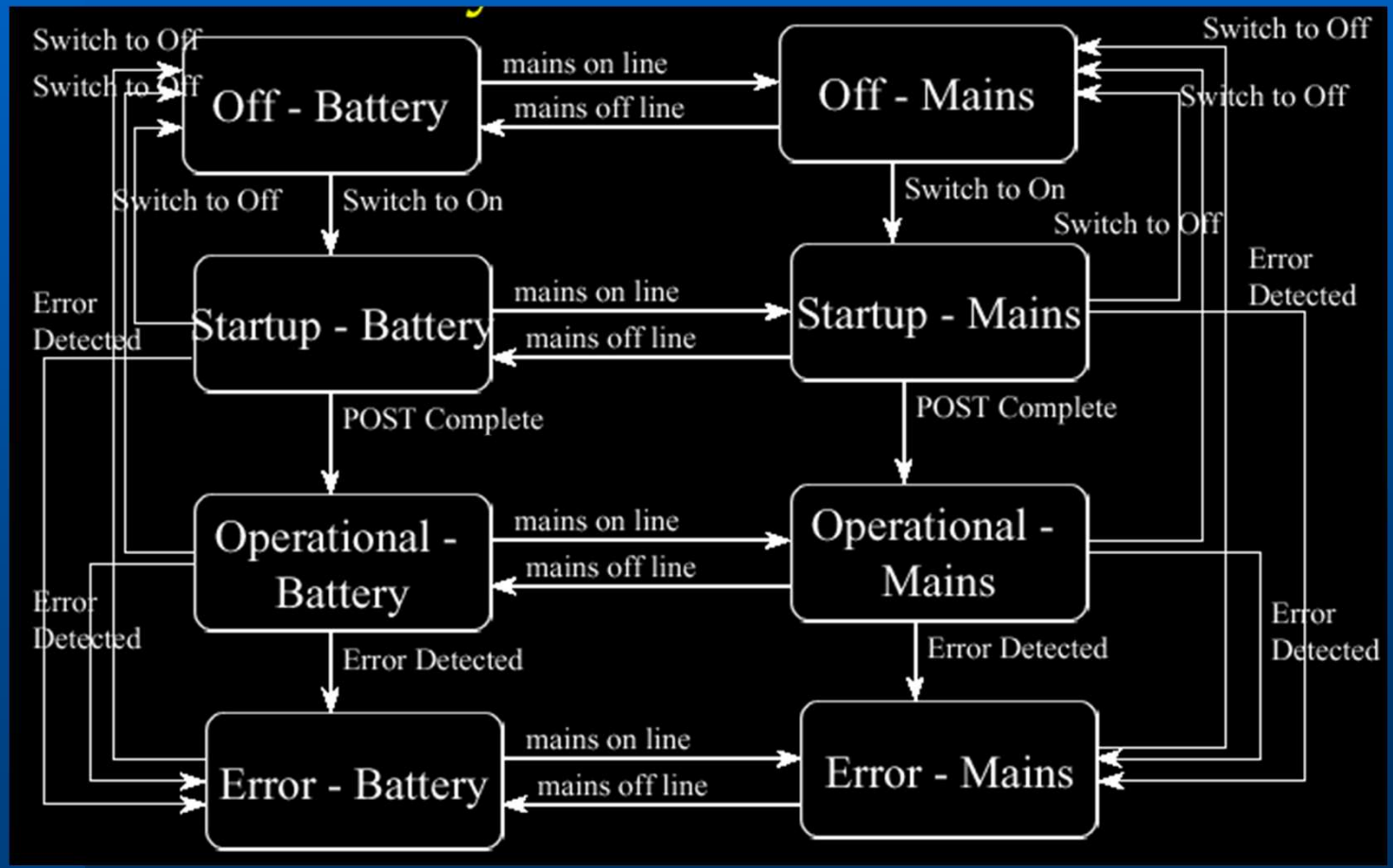


Concurrency

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

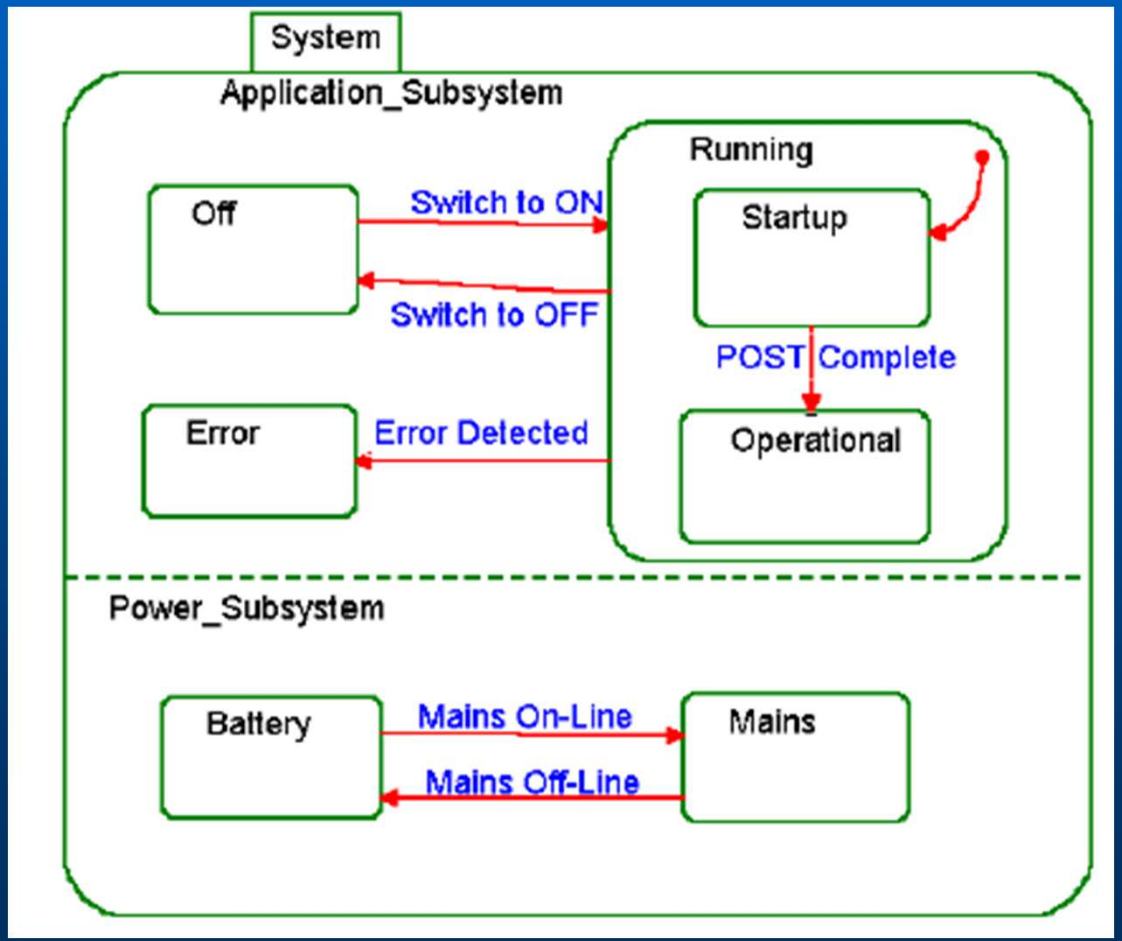
Source: B. P. Douglass & iLogix

Mealy-Moore Solution



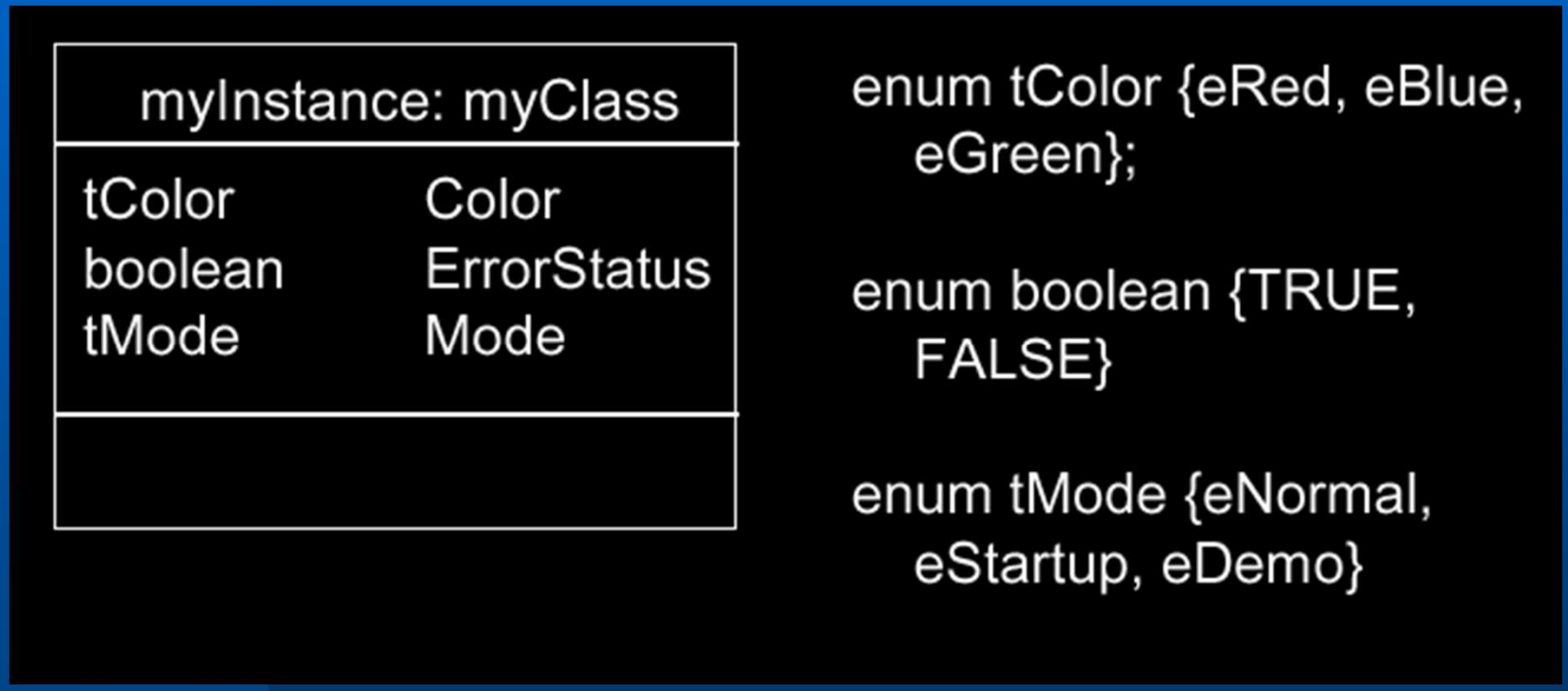
Source: B. P. Douglass & iLogix

State Charts Solution



Source: B. P. Douglass & iLogix

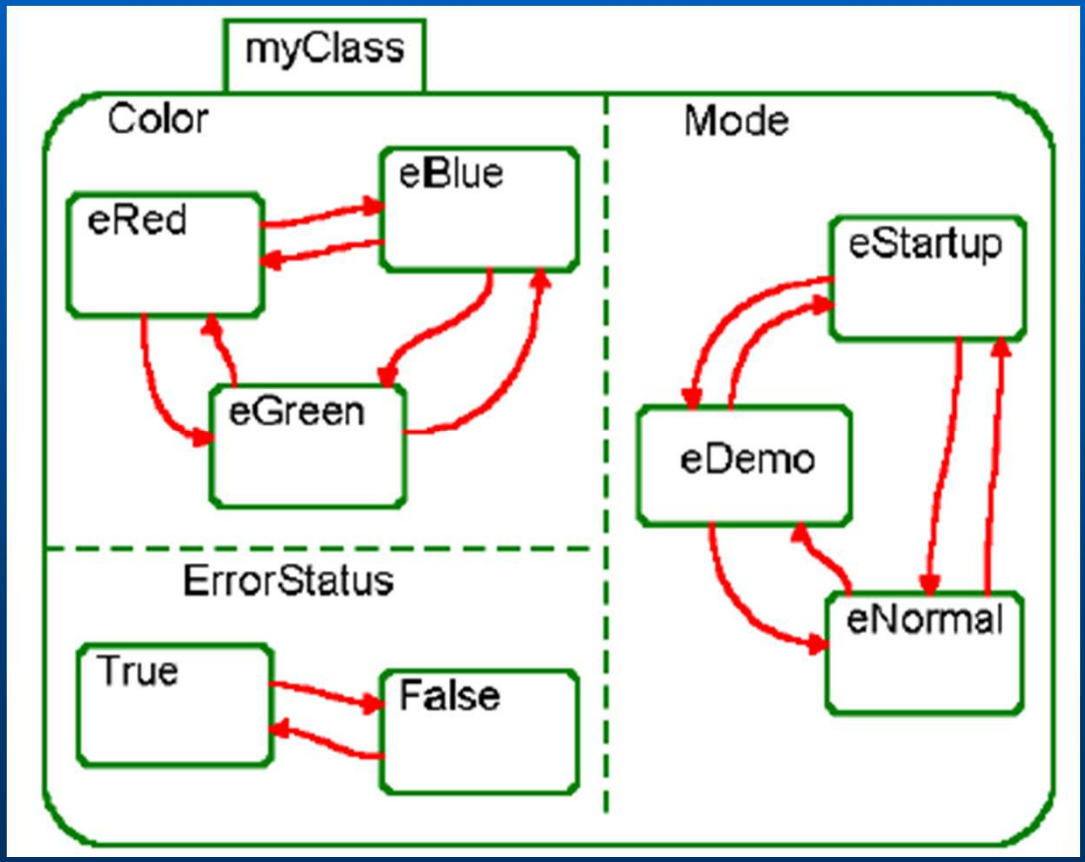
Orthogonal Components



How do you draw the state of this object?

Source: B. P. Douglass & iLogix

Hierarchical (and-composition) Model



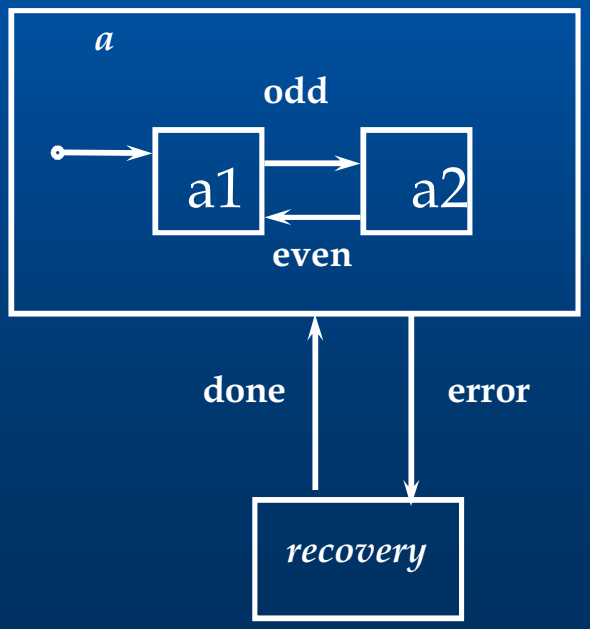
Source: B. P. Douglass & iLogix

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) communcation mechanism

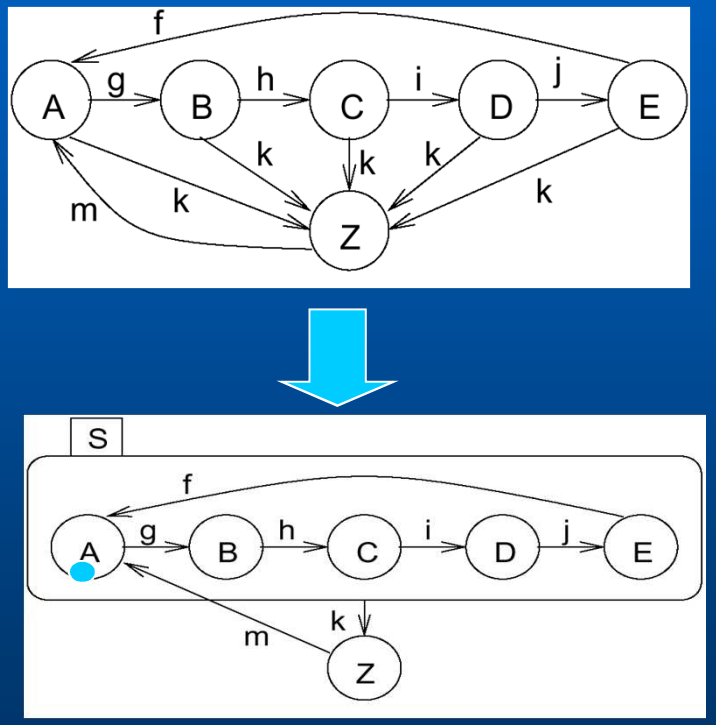
Hierarchical FSM model

- 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



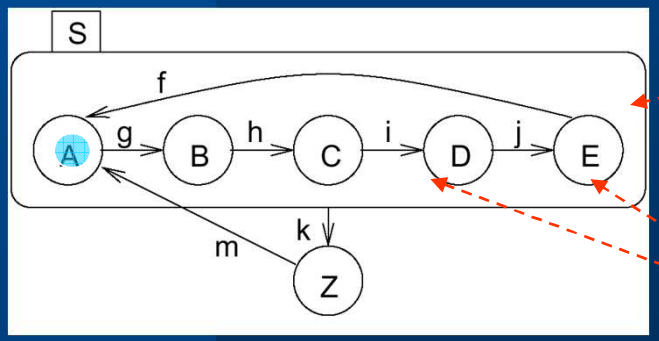
Hierarchical States

- 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.

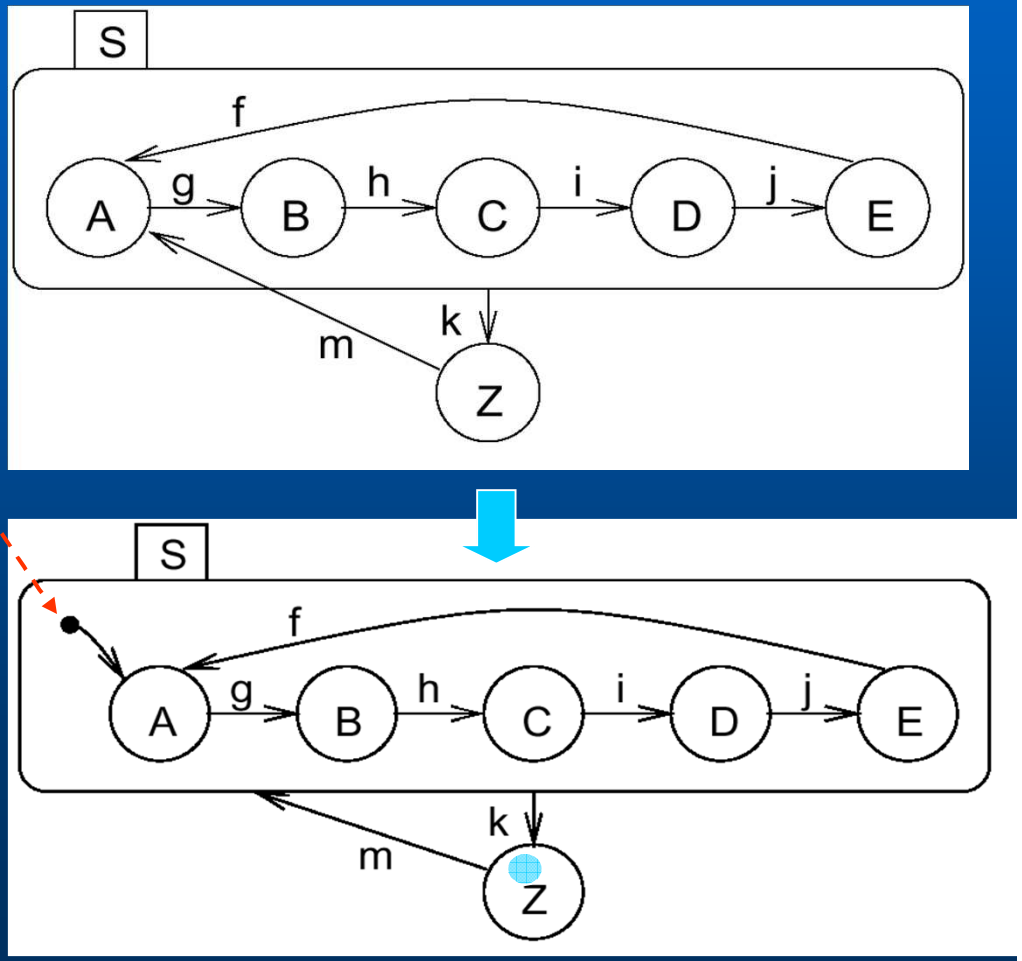


Superstate = ancestor of E

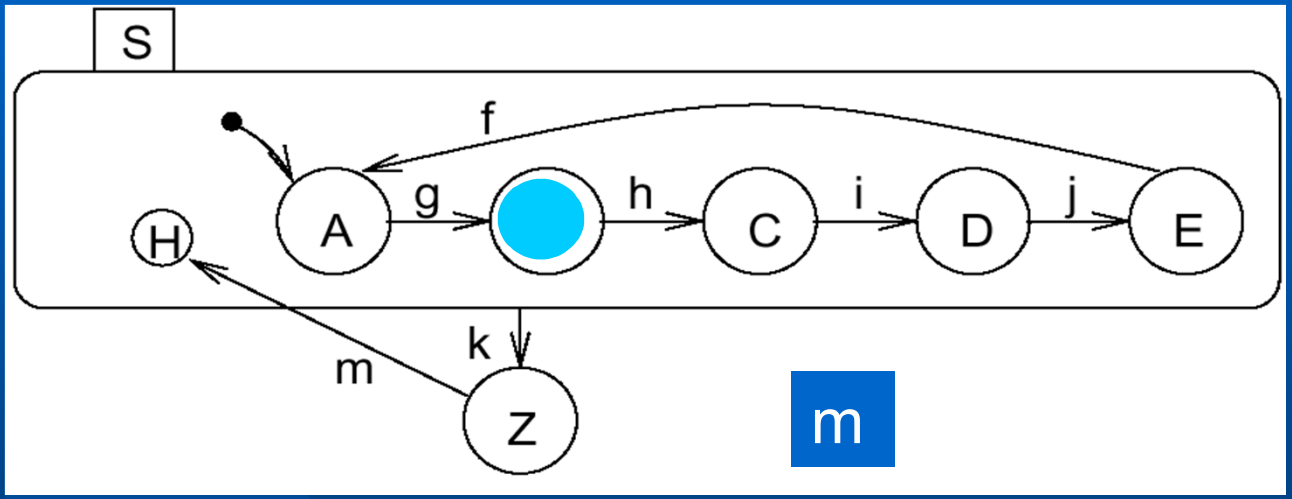
Sub-states

Default state mechanism

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



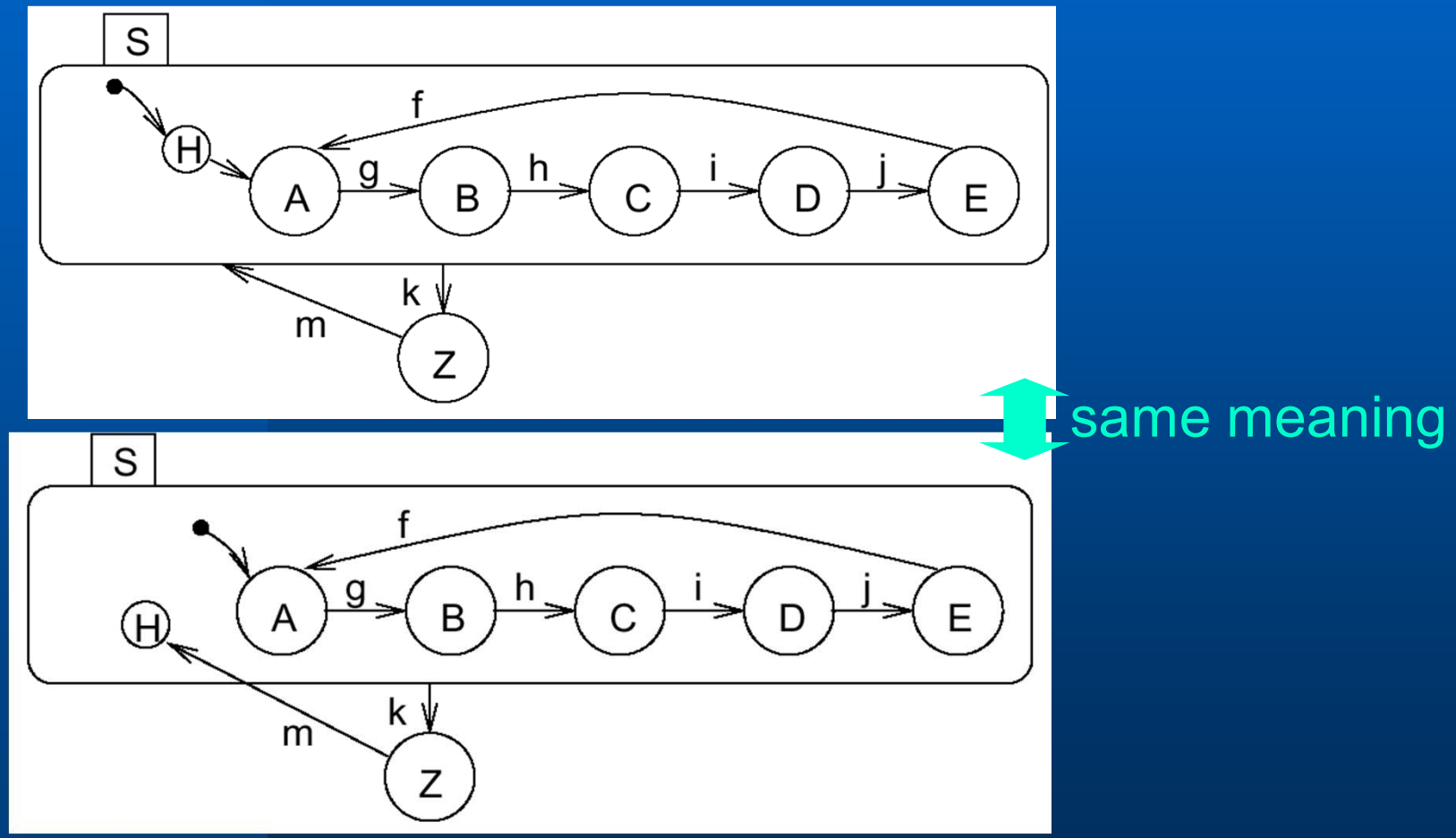
History Mechanism



(different behavior from last slide!)

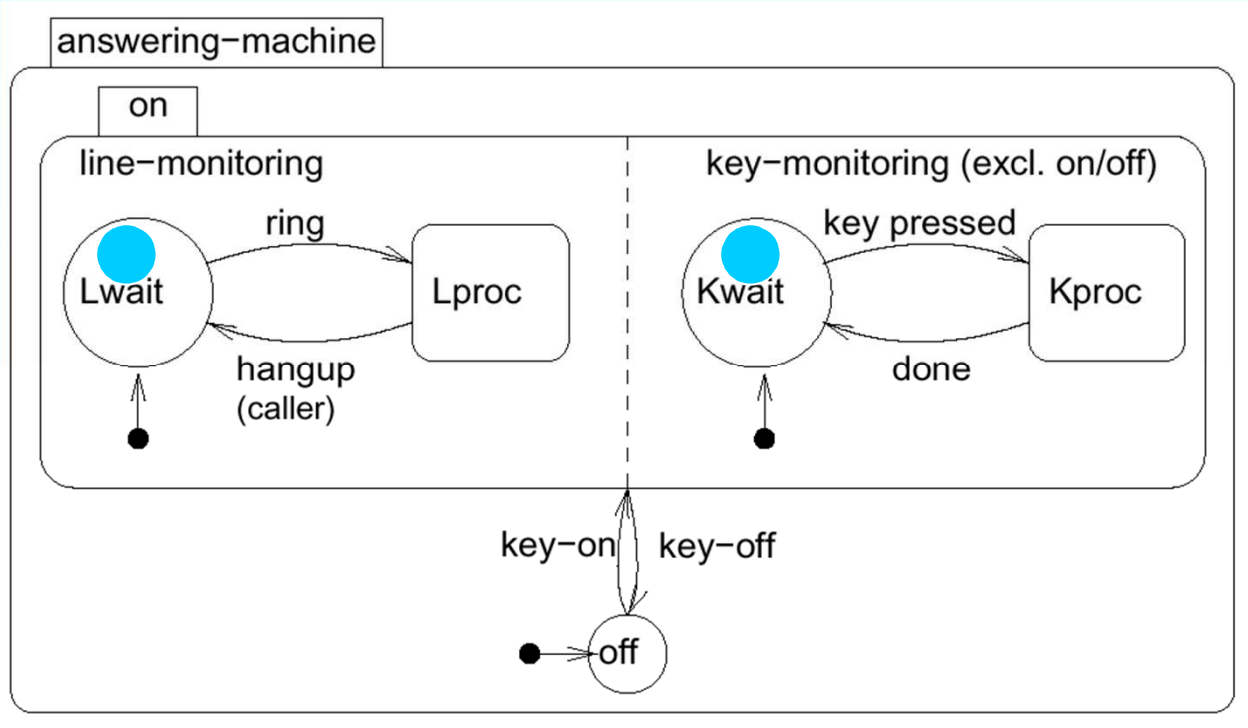
- Given input m, S returns to the state it was in before S was left (could be A, B, C, D, or E).
- On first time entry to S, the default mechanism applies.
- History and default mechanisms can be used hierarchically.

Combining history and default state mechanism

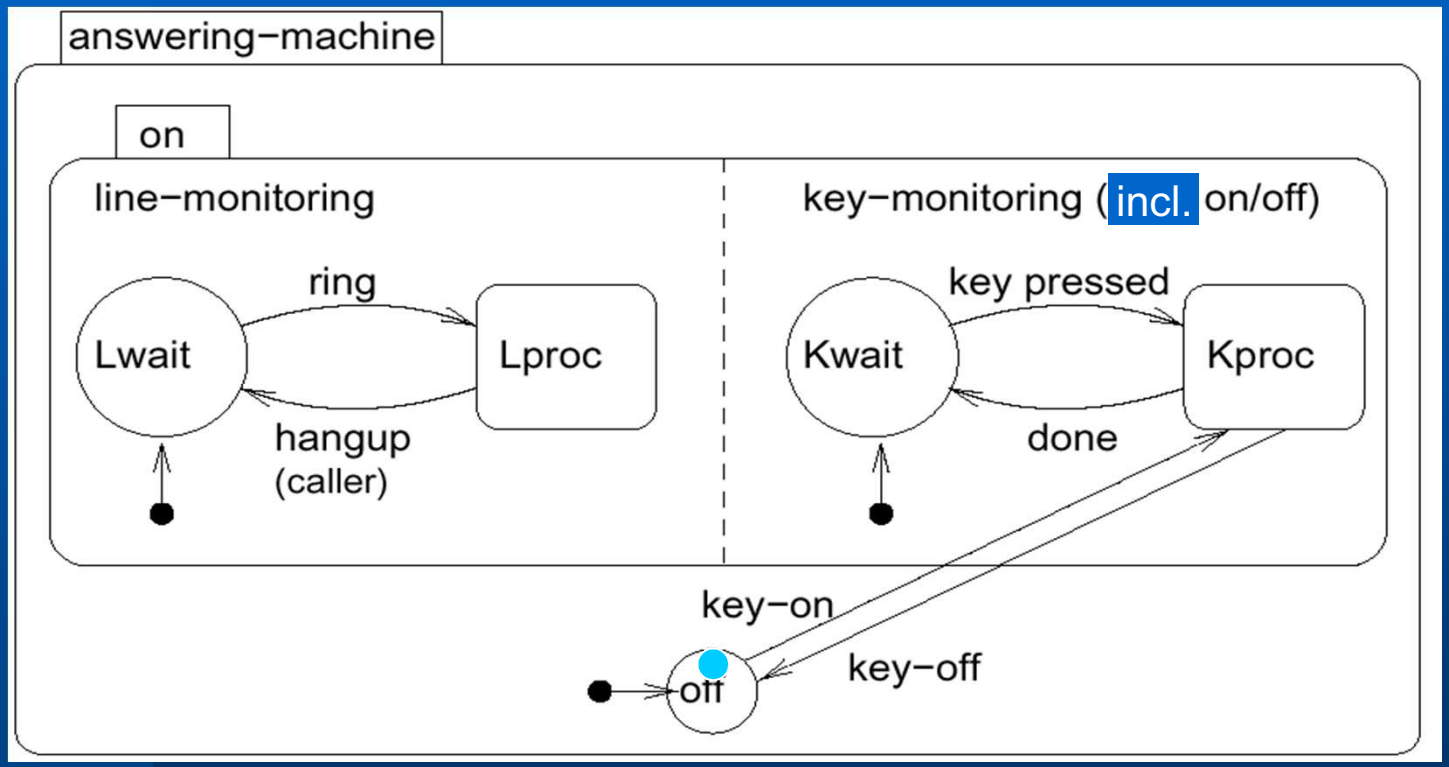


Concurrency

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

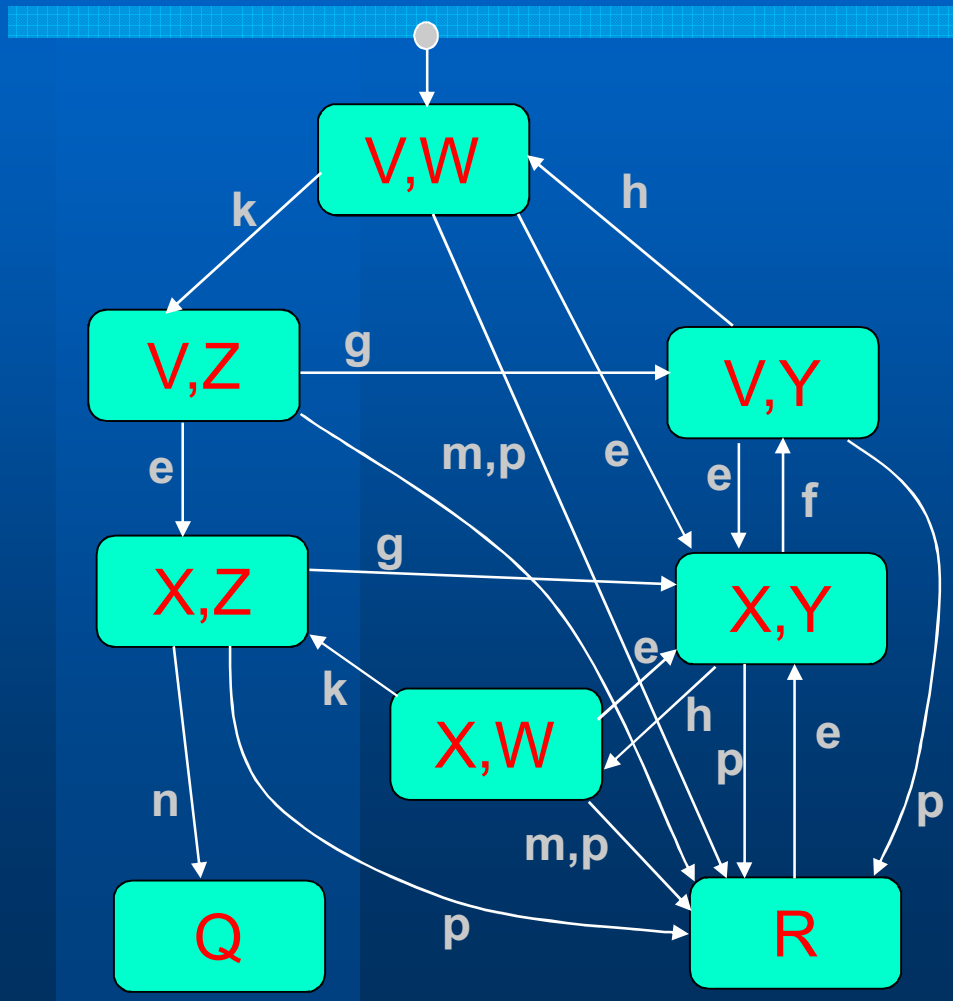


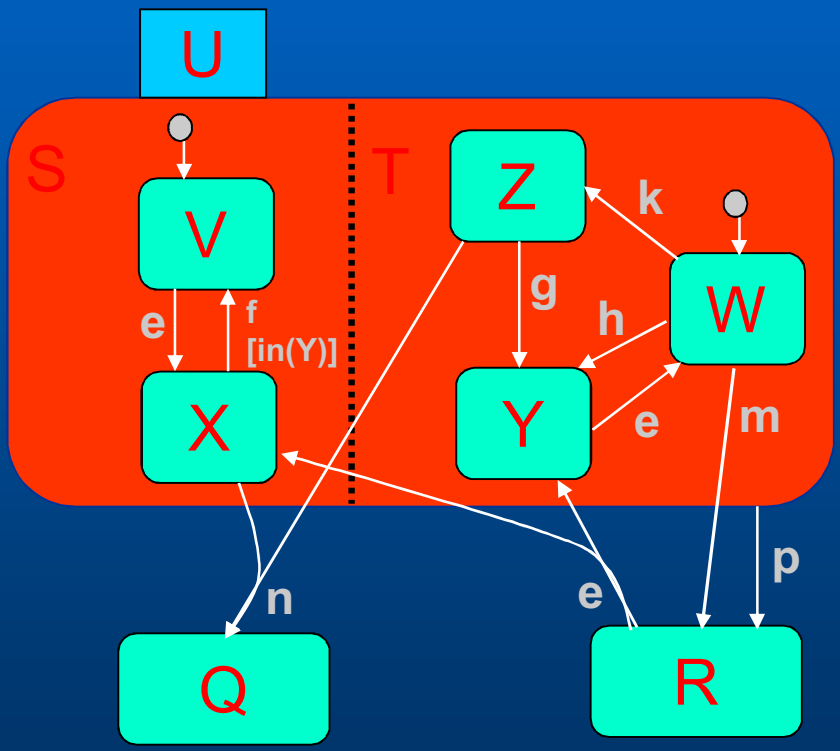
Entering and leaving AND-super-states



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

Benefits of AND-decomposition

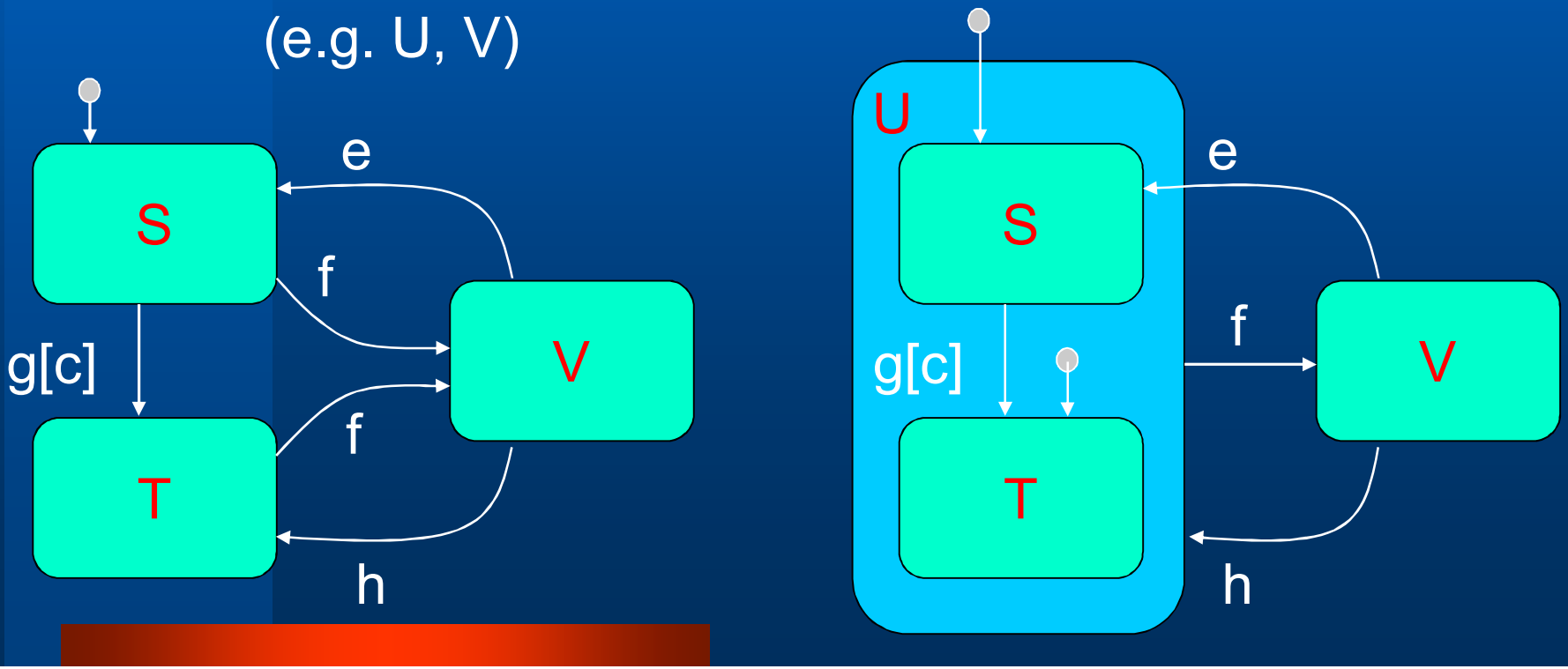




AND/OR State Comparison

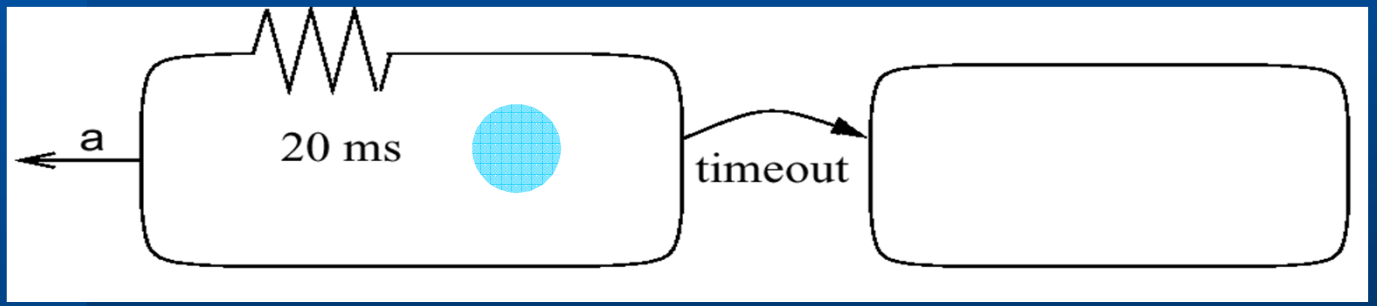
- AND-states have ***orthogonal*** state components
  - AND-decomposition can be carried out on any level of states
- OR-states have sub-states that are ***exclusive***

(e.g. U, V)



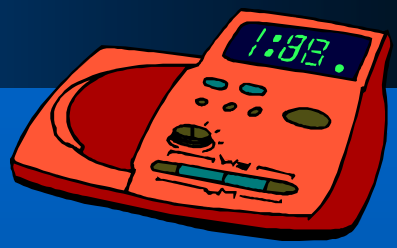
Timers

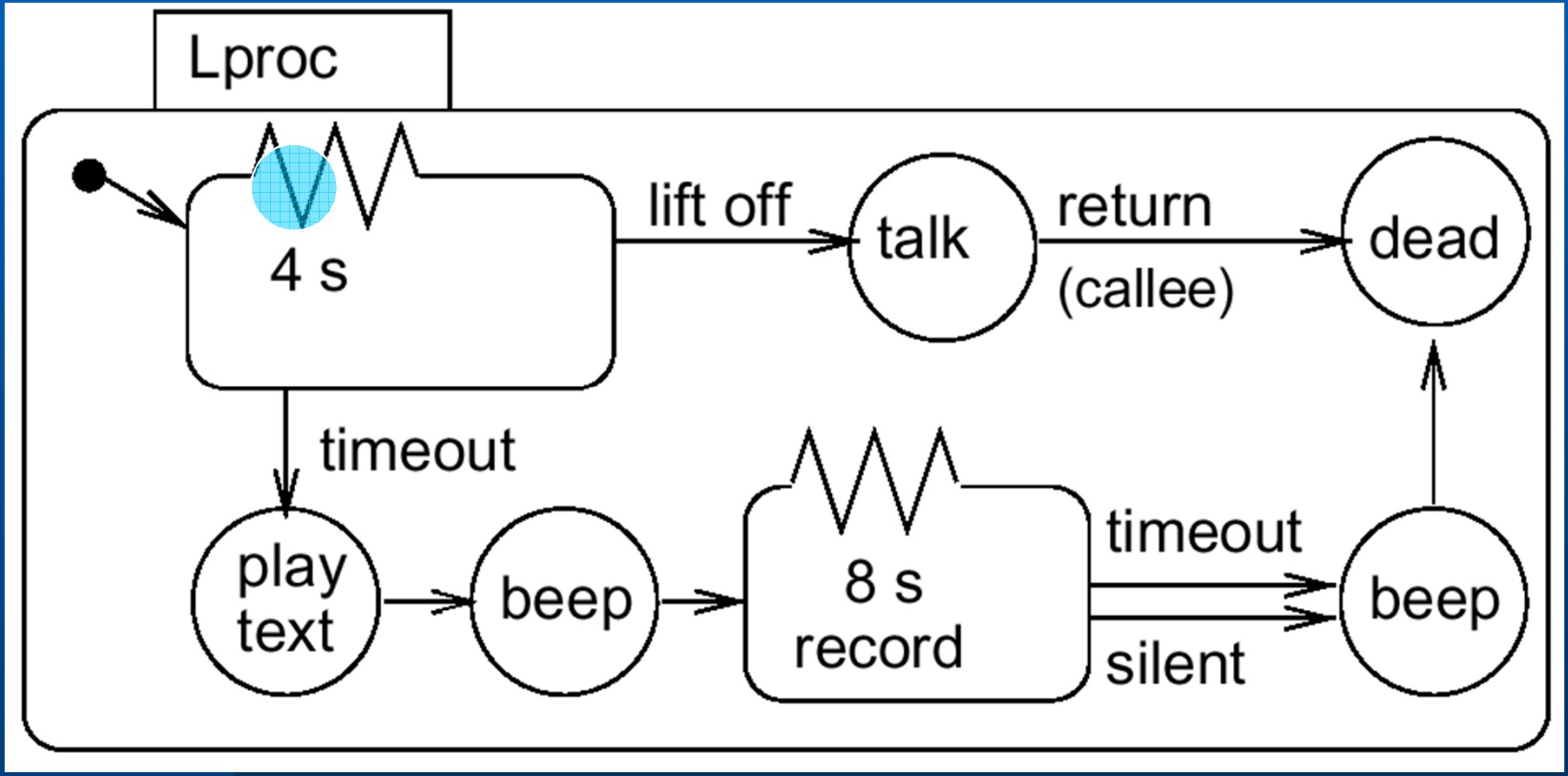
- Since time needs to be modeled in embedded systems, timers need to be modeled.
- In StateCharts, special edges can be used for timeouts.



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

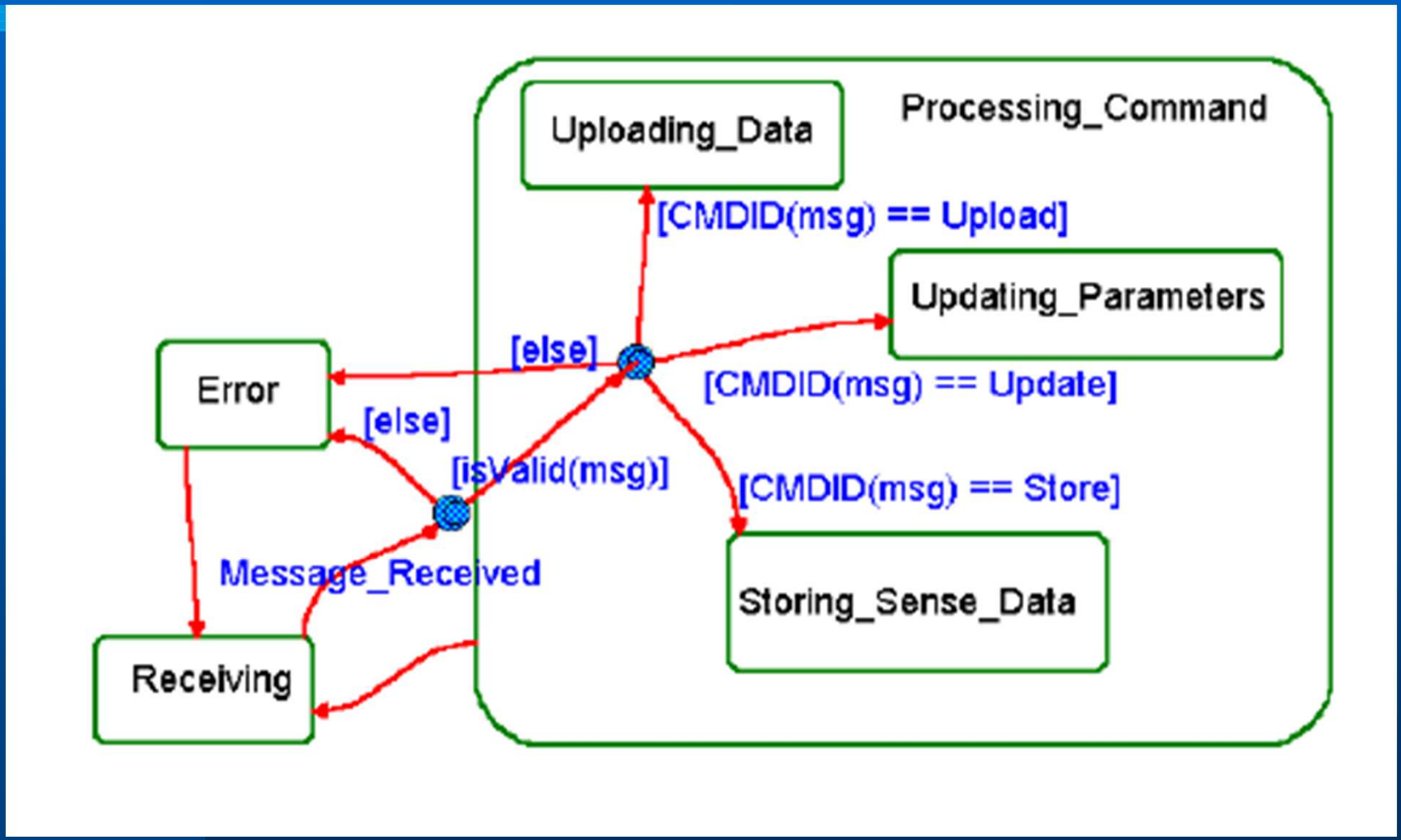




General edge labels in StateCharts

- 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
- Alternative: name(params)[guards]^event_list/action_list
  - Event list, aka propagated transitions, is a list of transitions that occur in other concurrent state machines because of this transitions
- 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



Source: B. P. Douglass & iLogix

StateCharts Actions and Events

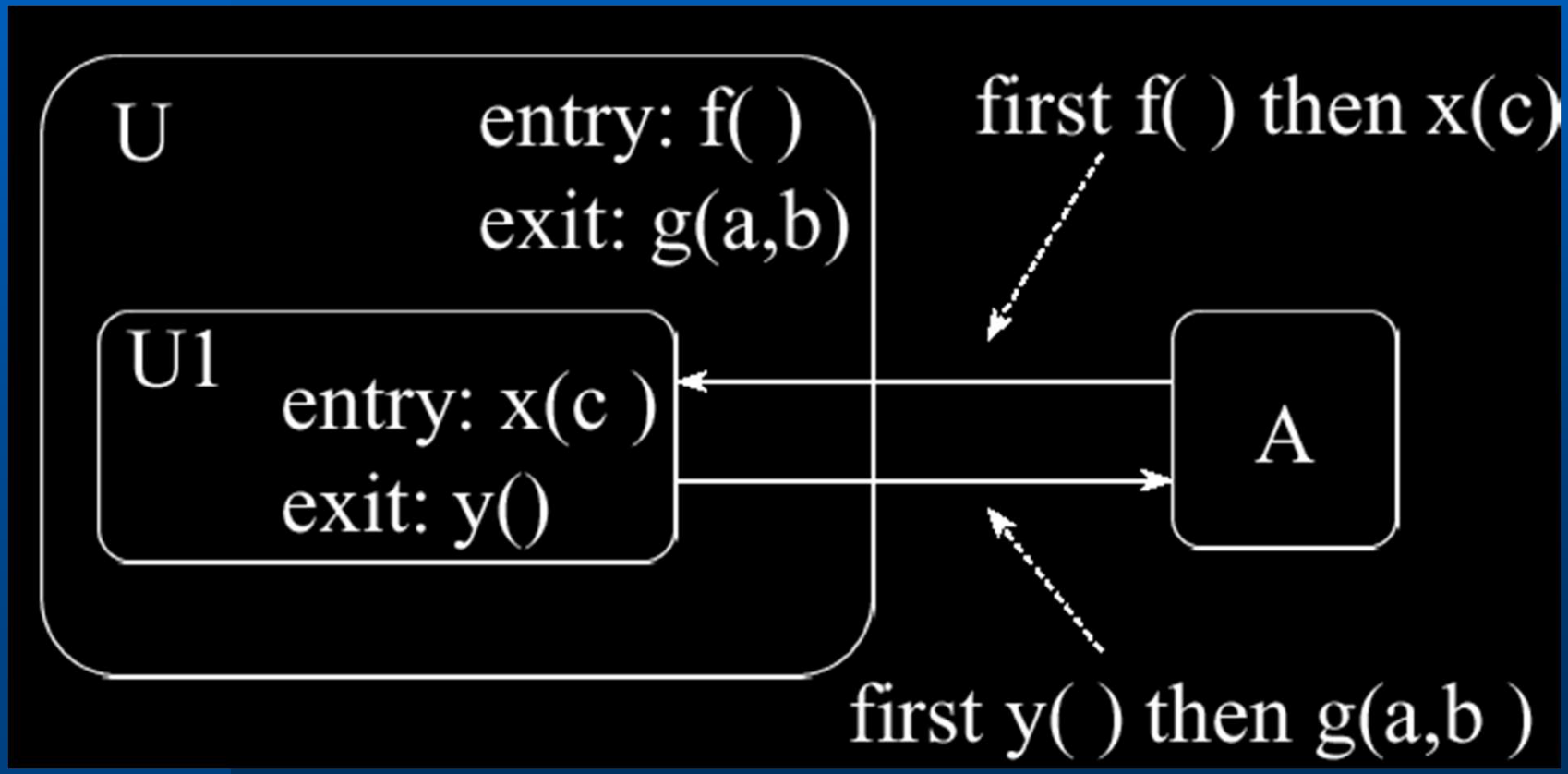
- An action A on the edge leaving a state may also appear as an event triggering a transition going into an orthogonal state
  - Executing the first transition will immediately cause the second transition to be taken simultaneously
- Actions and events may be associated to the execution of orthogonal components:
  - action start(A) causes activity A to start
  - event stopped(B) occurs when activity B stops
  - entered(S), exited(S), in(S) etc.

Communication in Concurrent FSMs

- Broadcast events
  - Events are received by more than one concurrent FSM
  - Results in transitions of the same name in different FSM
- Propagated transitions
  - Transitions which are generated as a result of transitions in other FSMs
- Issues:
  - Broadcast and propagated events and transitions can lead to very non-intuitive behaviors as they *violate locality* in the abstraction
  - Can result in very inefficient implementations

Order of Nested Actions

- Executed from outermost – in on entry

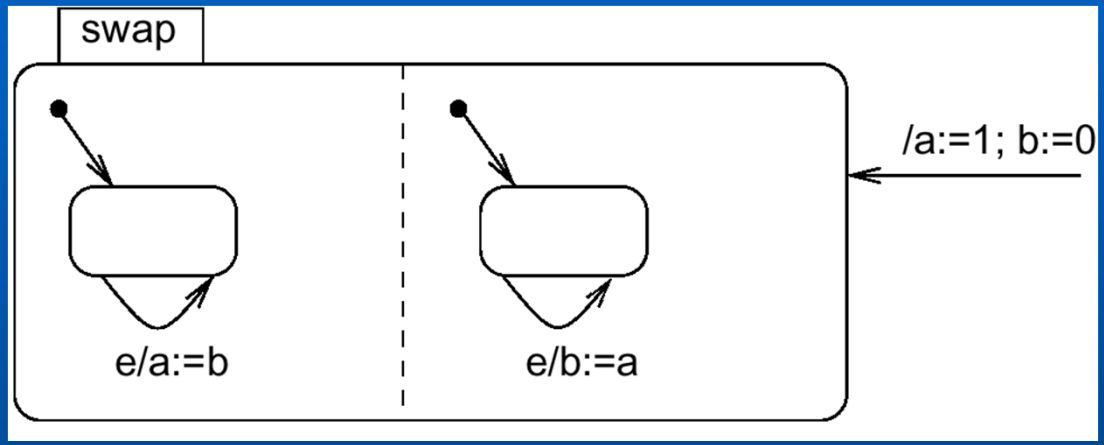


Source: B. P. Douglass & iLogix

StateCharts (StateMate Semantics)

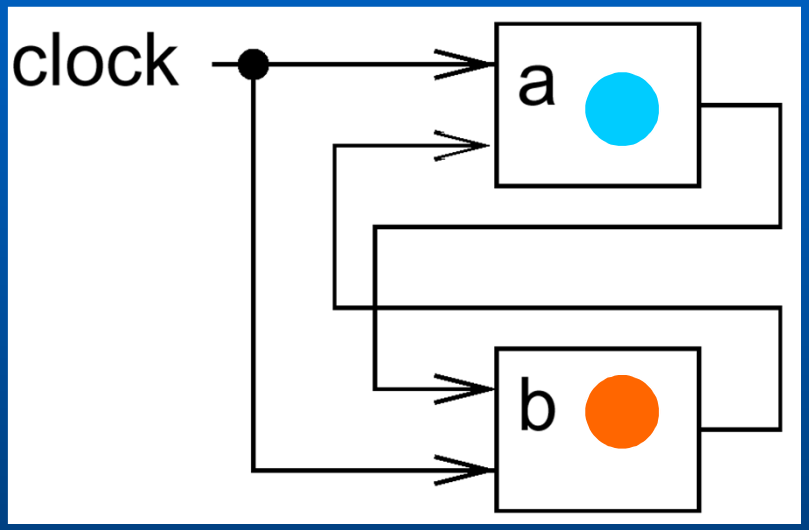
- How are edge labels evaluated?
- Three phases:
  - Effect of external changes on events and conditions is evaluated,
  - The set of transitions to be made in the current step and right hand sides of assignments are computed,
  - Transitions become effective, variables obtain new values.
- Separation into phases 2 and 3 guarantees deterministic and reproducible behavior.

Example



- In phase 2, a and b are assigned to temporary variables. In phase 3, these are assigned to a and b. As a result, variables a and b are swapped.
- In a single phase environment, executing the left state first could assign the old value of b (=0) to a and b. Executing the right state first would assign the old value of a (=1) to a and b. The execution would be nondeterministic.

Reflects model of clocked hardware



- In an clocked (synchronous) RTL system, both registers would be swapped as well.

Same separation into phases found in other languages as well, especially those that are intended to model hardware.

Evaluation of StateCharts

- Pros:
- Hierarchy allows arbitrary nesting of AND- and OR-super states.
- (StateMate-) Semantics defined in a follow-up paper to original paper.
- 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.

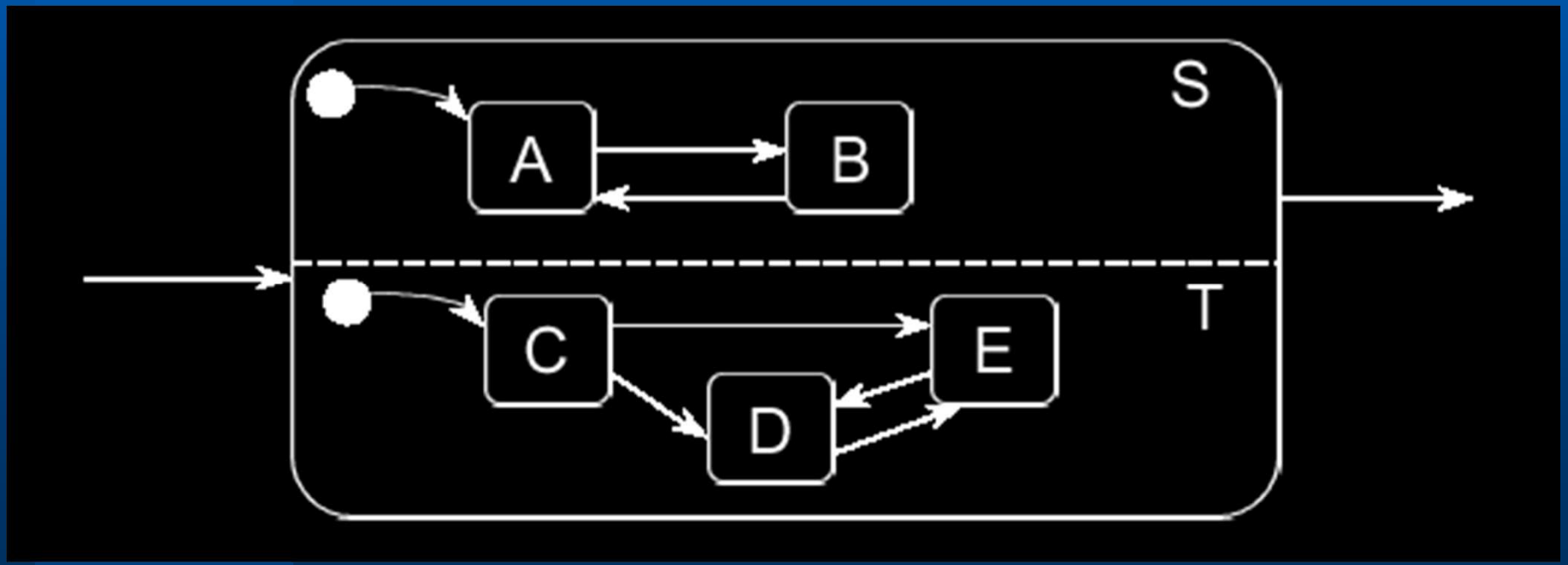
Evaluation of StateCharts

Cons:

- Generated C programs can be inefficient
- Generated Hardware can be worse
  - Both could be improved by restrictions on broadcast and transition propagation, but this requires fundamental semantic changes
  - But, quite useful as a paradigm for extended automata
    - States become durational events
    - Systematic, non-unit time event model
- Difficult to apply to distributed applications
- No description of structural hierarchy

Concurrent Statecharts

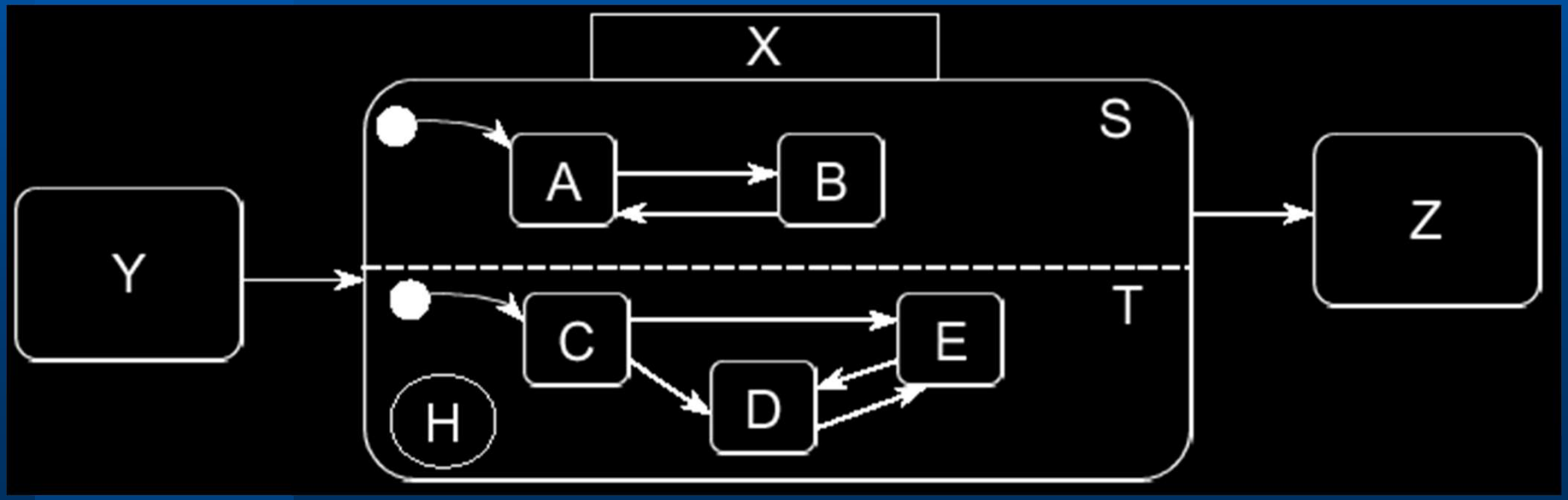
- Many embedded systems consist of multiple threads, each running an FSM
- State charts allow the modeling of these parallel threads



Source: B. P. Douglass & iLogix

Concurrent Statecharts

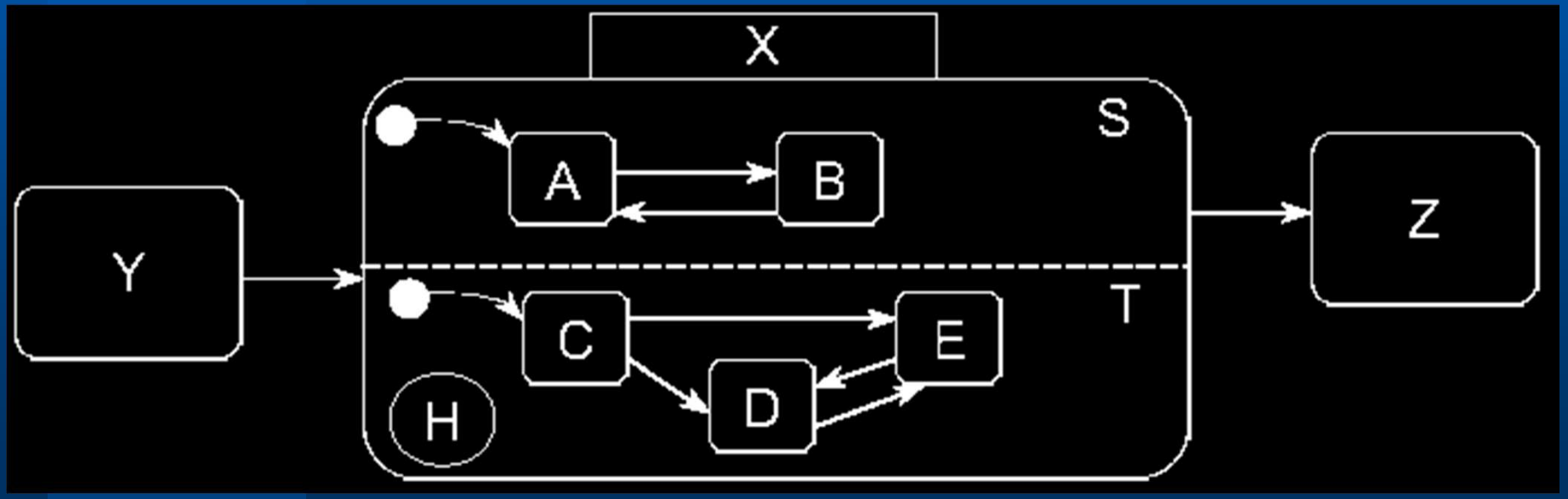
- States S and T are active at the same time as long as X is active
  - Either S.A or S.B must be active when S is active
  - Either T.C, T.D or T.E must be active when T is active



Source: B. P. Douglass & iLogix

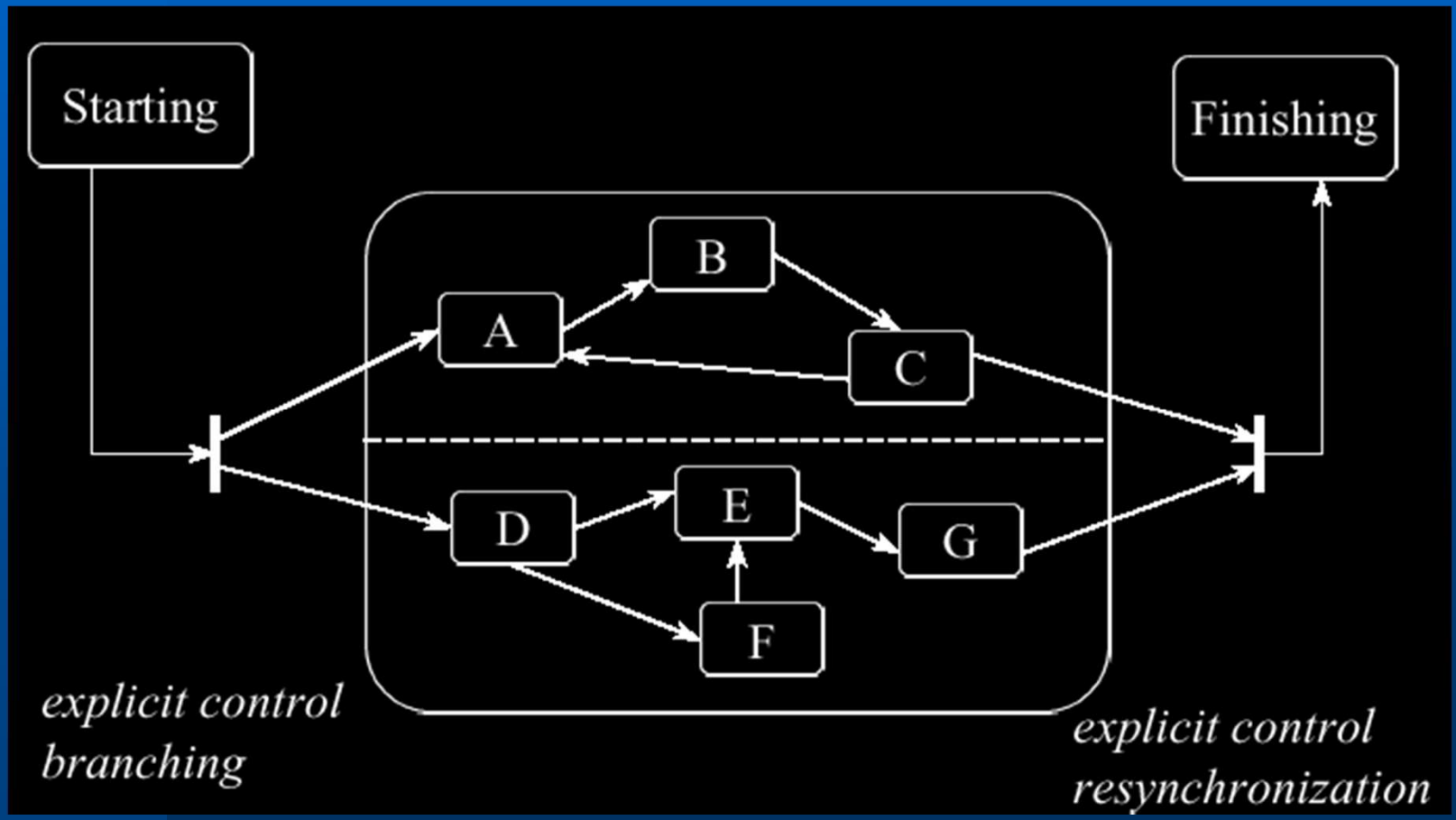
Concurrent Statecharts

- When X exits, both S and T exit
  - If S exits first, the FSM containing X must wait until T exits
  - If the two FSMs are always independent, then they must be enclosed at the highest scope



Source: B. P. Douglass & iLogix

Explicit Synchronization

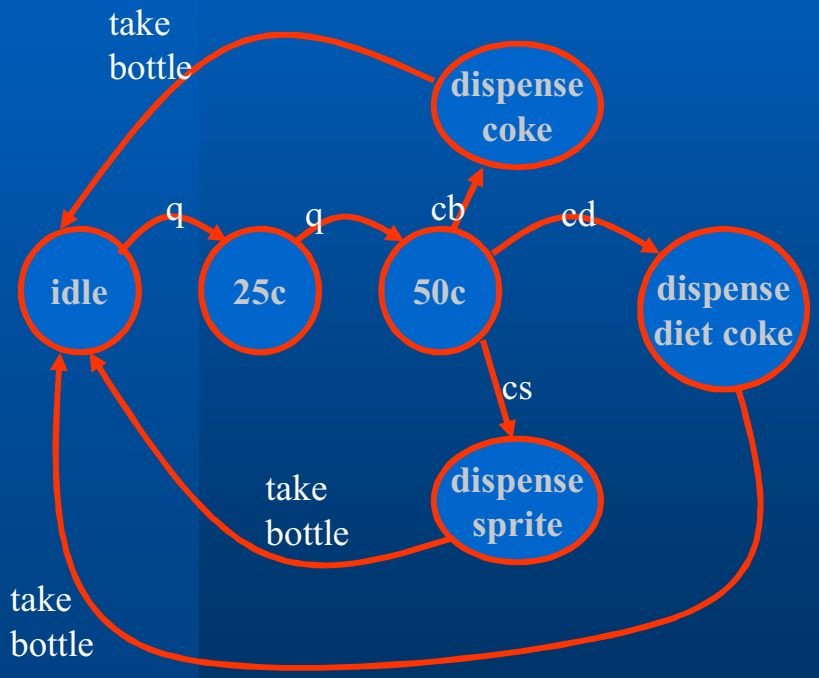


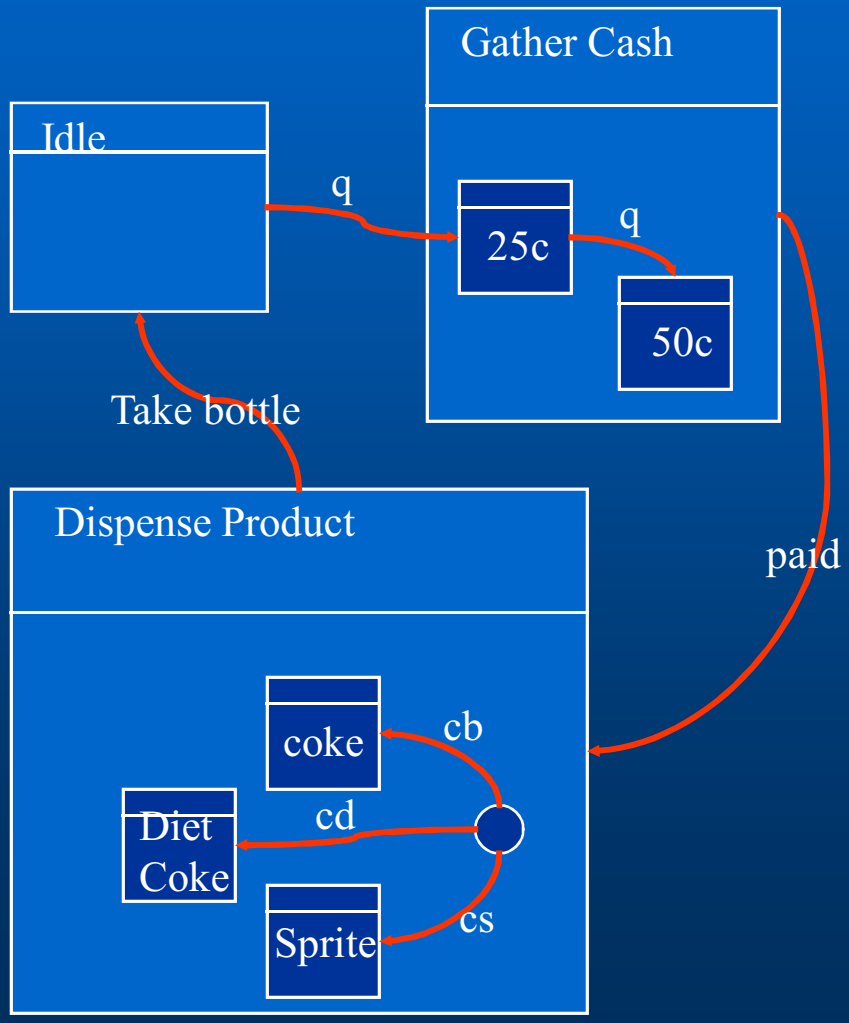
Source: B. P. Douglass & iLogix

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

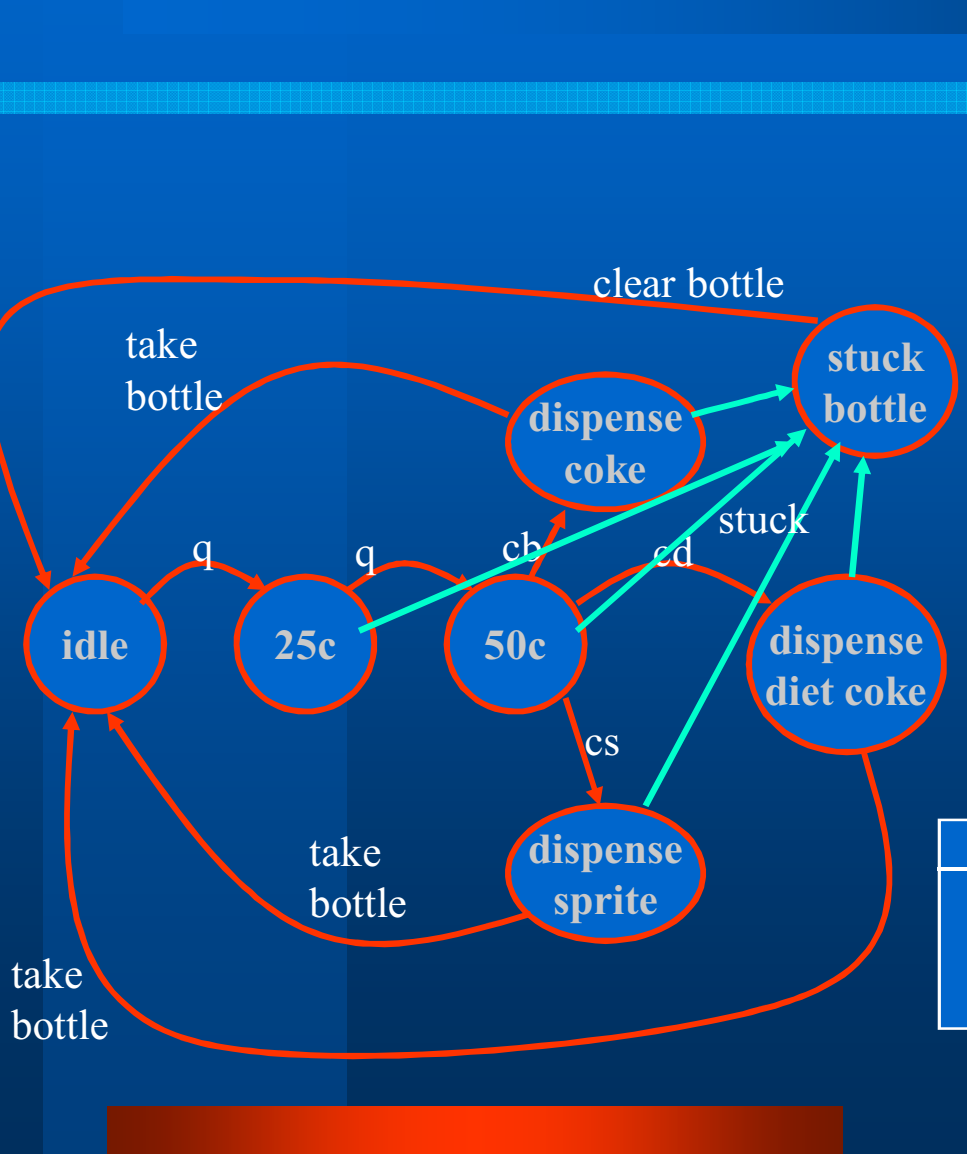


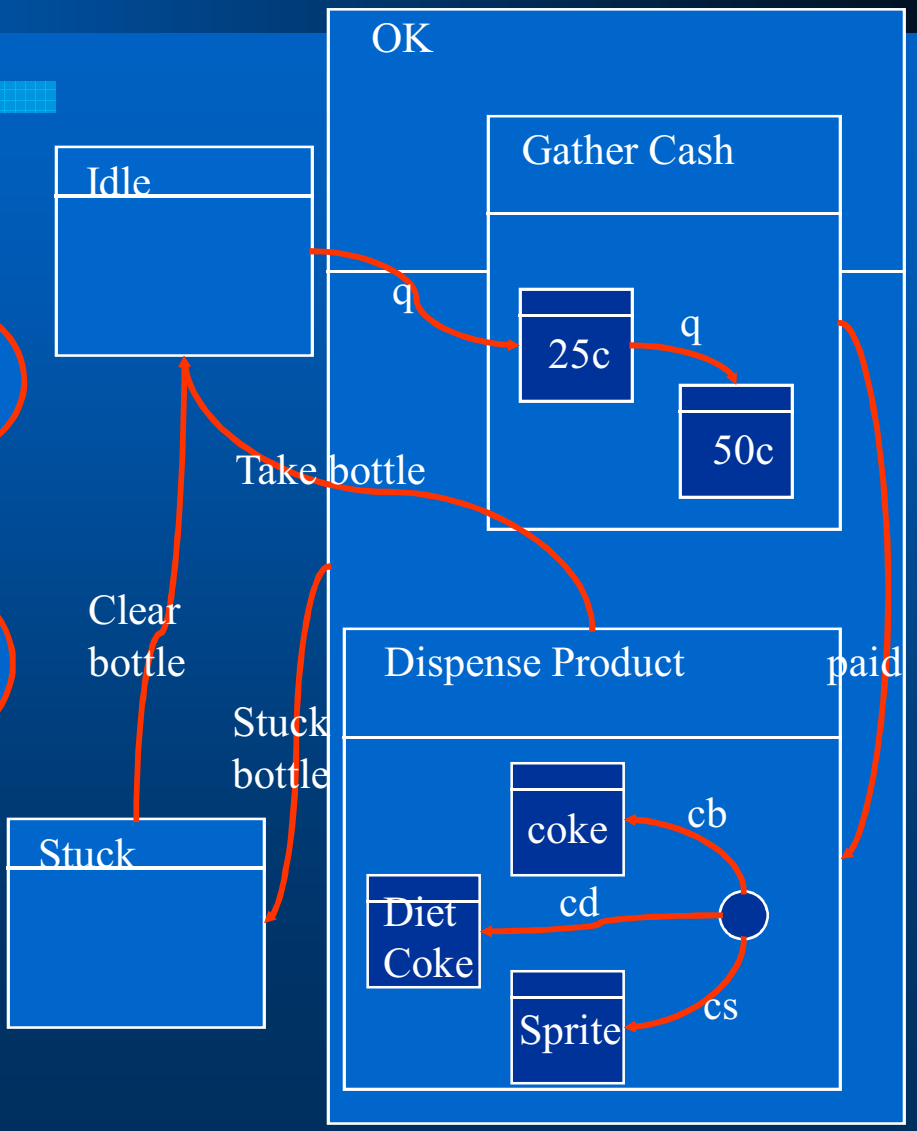


Coke Machine, Version 1.1

- 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

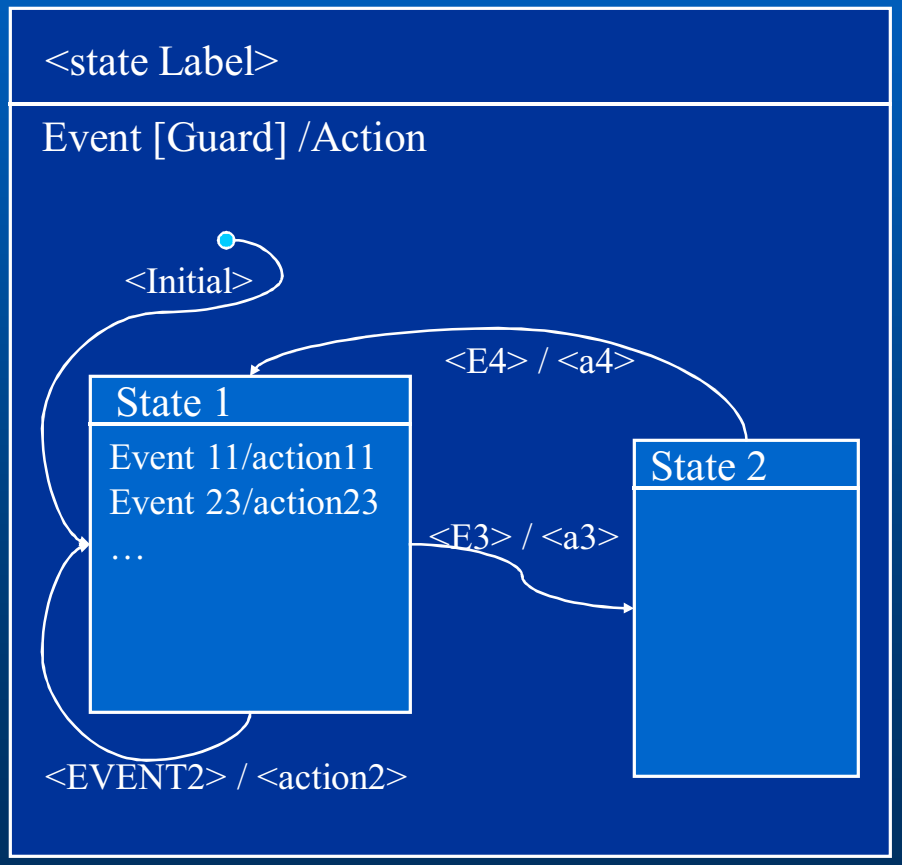
- Hierarchy allows for
  - Sensible default activity
  - Multiple places to augment behavior
    - Consider behavior on stuck recovery– eats your money…
  - Large reduction in number of states required
  - Easy to add extended state semantics
    - Augmented state and guarded actions
- How can we make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

UML State Machines

- Syntax and supported semantics for QP nano and related software dispatch schemes.
  - Want to support most important parts of State Charts
  - Want implementation to be directly in C
    - Easier to debug and understand
    - Can be reasonably fast dispatch despite direct implementation
  - UML model allows use of UML tools for code coverage, validation and (sometimes) verification.
  - UML model adopted Harel's work so this is the natural representation for UML…

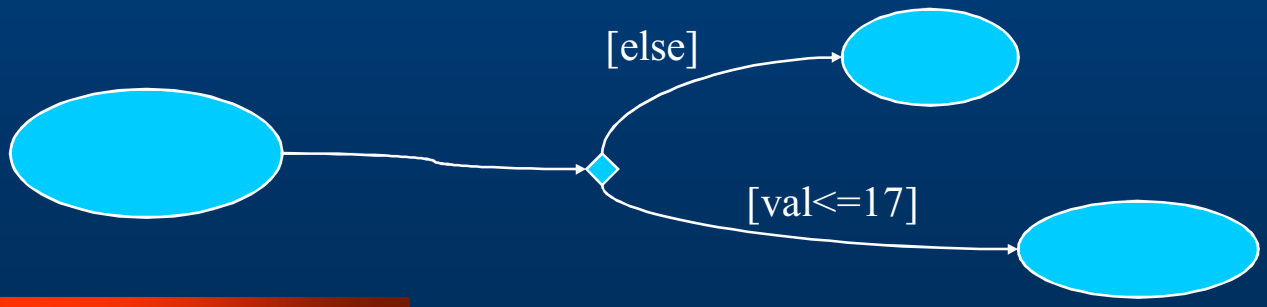
States

- Formalize notion of context (modality) and events (inputs)
- Transitions move the machine to new states
  - Provide for ***actions***
    - Actions could be C/C++ code fragments
    - Modify local state
    - Create new events
  - Explicit modality
- Compresses behavior since don't care what sequence of events got you to the state – just what state you are at…



Extended FSM

- Possible to represent every bit of context as state
  - But really inefficient: consider a 32 bit parameter in a sin(x) function:
    - Resolving each bit as ‘state’ leads to an explosion of states
    - Program size would explode too…
- Instead, extend FSM behavior with local context static parameters
  - Apply ***Guards*** as needed to force control changes:



Guards

- In general, guards could be thought of as a generalized event– but:
  - In both Software and Hardware guards are more complex than signals
    - Often signals can be simple enumerations leading to simple pointer based function dispatch
    - Guards are expressions that need to be dynamically evaluated so require if (expression) nesting in simplest case
- So – guards solve a real problem, but the cost is not free
- Common problem in FSM based systems upkeep is explosion of guards and static ‘flags’ which ought to be encoded as new state behavior.
- Think of Guards as part of the Data-Flow, Events as control tokens

Events

- An activity at a time which needs to be accounted by the system behavior
- Events often carry parameters
  - E.g. an interrupt might have an identity and time stamp
  - A button depression should tell which button
- Frequently, Events are queued
  - Allow for sensitivity to happenings at unplanned times
  - Enable Simpler Execution Models

Transitions

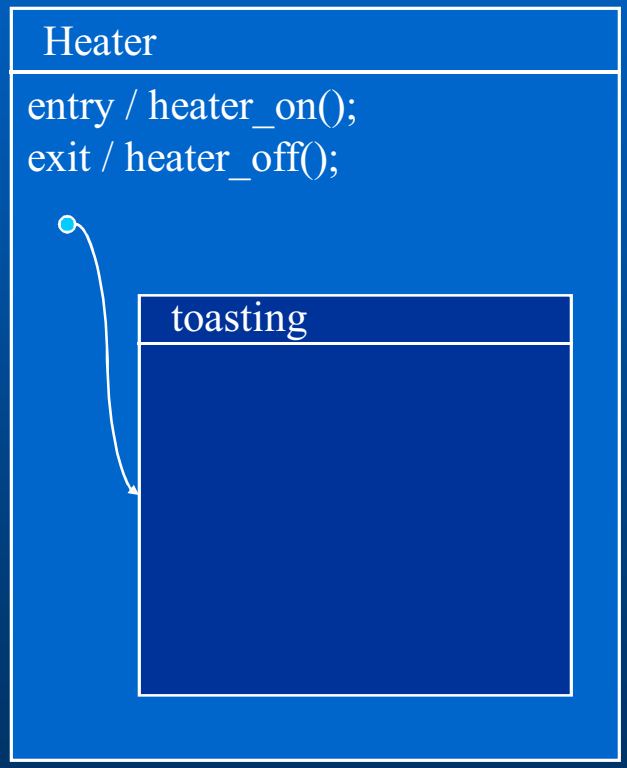
- In general, a transition needs:
  - A parent state
  - A triggering Event
  - An optional guard expression
  - A child state (to transition to)
  - An Action (modification of the local state within the context scope
- Actions and guards may take real-time
- A single event might sponsor multiple transitions from a state
  - Differentiate by guards (must be deterministic in UML)
  - No defined order of testing the possible transitions
  - Bad idea for guards to have side effects!

Run to Completion

- It is assumed that all sponsored behaviors run to completion
  - New events are queued, but cannot modify transitions in progress
- This does *not* mean that the FSM itself can't be preempted!
  - But the state and resources of the FSM cannot be shared with potentially preempting tasks
- Could have several active contexts, each with its own FSM as long as events are queued and dispatched to all machines.
- Issue: latency of this model is bounded by worst case step
  - Keep actions and guards short!

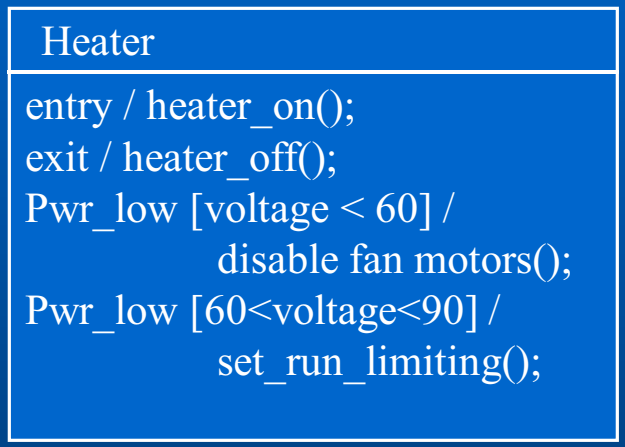
Entry and Exit Actions

- Provide for activity on entry and exit of state
- *entry* and *exit* are UML keywords
- Avoids adding actions to all entering/exiting transitions

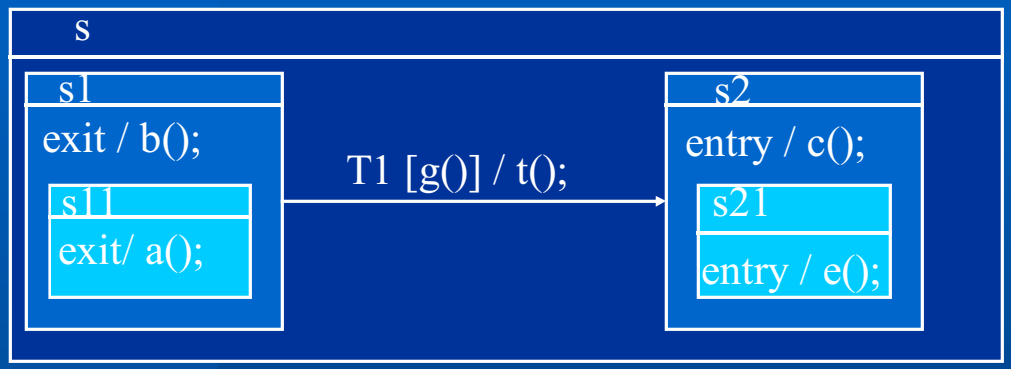


Internal Transitions

- Sometimes events should cause actions, but not change of state
  - Simply add (unordered) list of transitions to state body



Execution Sequence



1. Evaluate Guard, if TRUE
2. Exit Source State
3. Execute Actions of transition
4. Enter Target State

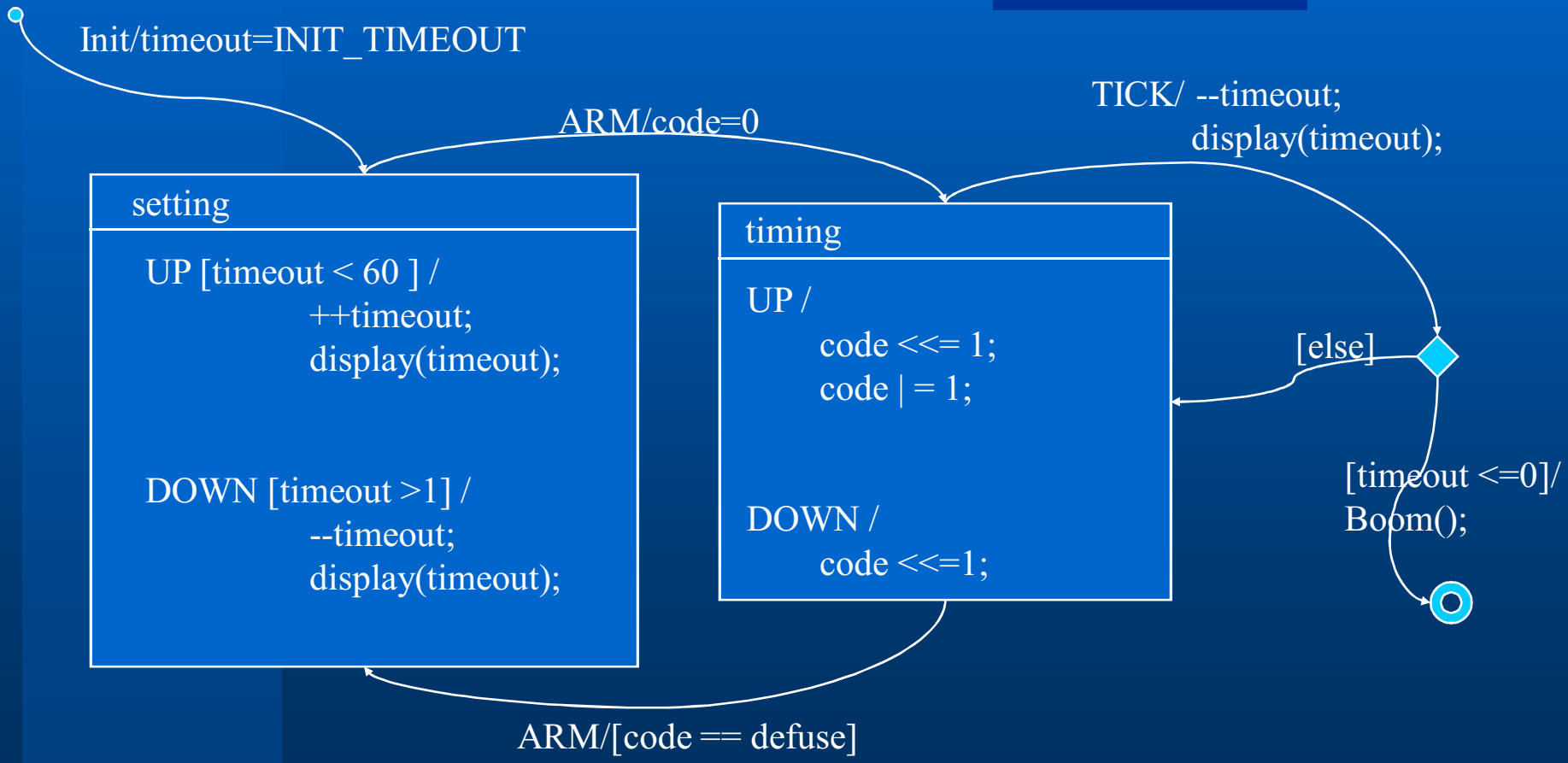
UML: g(), if true then a(), b(), t(), c(), e()

In general, must exit states to least common State containing both source and target, then Enter states until target is reached

Note– some FSM models evaluate t() in same context as g() (the source context) to get: g(), t(), a(), b(), c(), e()

Time-Bomb Example

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



State Machine Coding Patterns

Nested Switch FSM

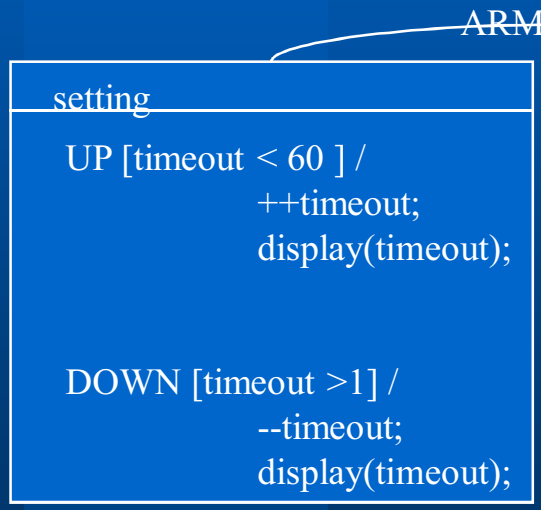
- “State” is an enumeration of a static type
- “event” is a typed signal (Also an enumeration)
- “init” sets up the timers and initializes the variables
- “dispatch” is called on each event to pass control to FSM

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;
```

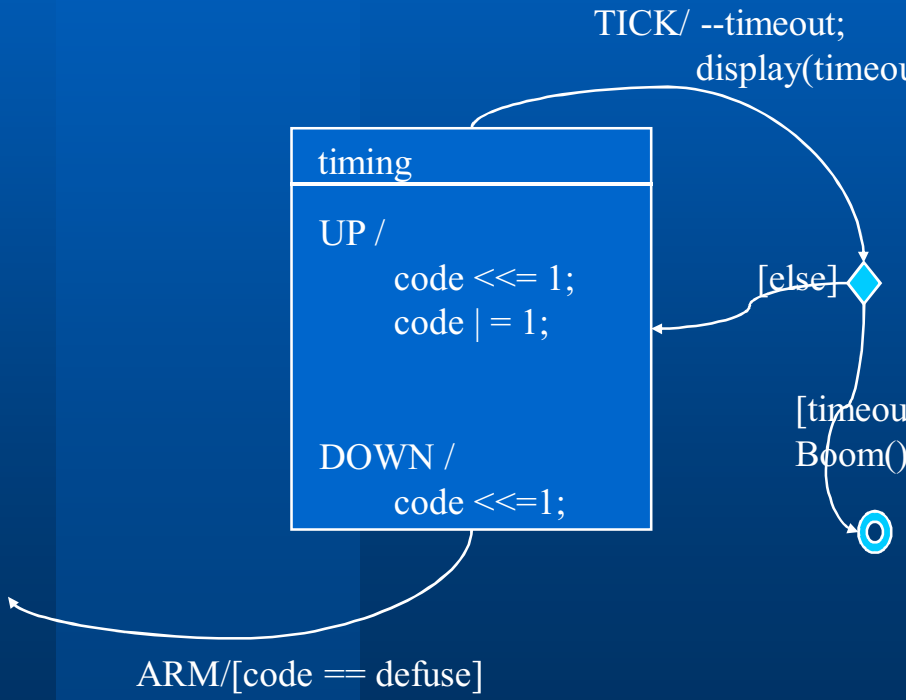
```
typedef struct Bomb1Tag {
    uint8_t state;
    uint8_t timeout;
    uint8_t code;
    uint8_t defuse;
} Bomb1;

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

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*

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();
    }
}
```

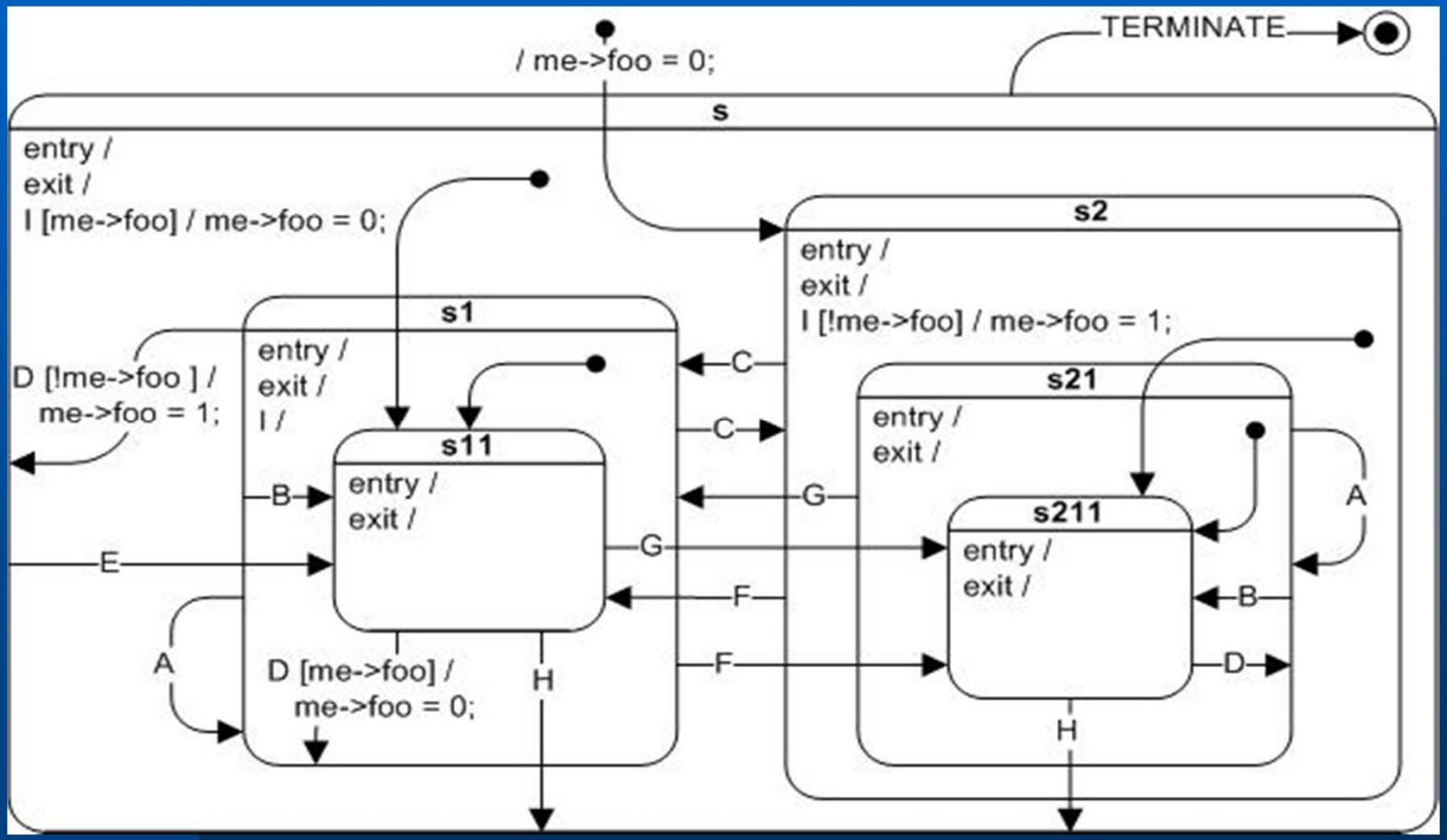
Table Event Processor: Issues

- Regular Structure for FSM
  - Enumerations used as indicies
  - 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

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…

Hierarchical FSM Example



Sample State Handler: Declarations

- Enumerate all Signals (events)
- Note Single inheritance of foo as extended state
  - QHsm must be first in structure!
- Static Pointers to State Handler Functions

```
enum QHsmTstSignals {
    A_SIG = Q_USER_SIG, B_SIG, C_SIG,
    D_SIG, E_SIG, F_SIG, G_SIG, H_SIG,
    I_SIG, TERMINATE_SIG, IGNORE_SIG,
    MAX_SIG };

typedef struct QHsmTstTag {
    QHsm super;
    uint8_t foo;
} QHsmTst;

static QState QHsmTst_initial(QHsmTst *me);
static QState QHsmTst_s    (QHsmTst *me);
static QState QHsmTst_s1   (QHsmTst *me);
static QState QHsmTst_s11  (QHsmTst *me);
static QState QHsmTst_s2   (QHsmTst *me);
static QState QHsmTst_s21  (QHsmTst *me);
static QState QHsmTst_s211 (QHsmTst *me);

QHsmTst HSM_QHsmTst;
```

Sample State Handler: Code

- Q_SIG() Is a macro reading the event signature
- Q_HANDLED() is a stub that indicates no further event handling is needed
- Note that Q_SUPER (here the top state) is the default if all the switch cases miss
- Transitions update return addresses

```
QState QHsmTst_s(QHsmTst *me) {
    switch (Q_SIG(me)) {
        case Q_ENTRY_SIG: {
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            return Q_HANDLED();
        }
        case Q_INIT_SIG: {
            return Q_TRAN(&QHsmTst_s11);
        }
        case E_SIG: {
            return Q_TRAN(&QHsmTst_s11);
        }
        case I_SIG: {
            if (me->foo) {
                me->foo = 0;
                return Q_HANDLED();
            } break;
        }
        case TERMINATE_SIG: {
            return Q_HANDLED();
        }
    }
    return Q_SUPER(&QHsm_top);
}
```

Hierarchical State Handling

- In this merged model, event handling and deferral are by switched state handler functions, but transitions must be managed via the state list
- Added issues:
  - Support for Hierarchical Semantics:
  - Top State and Initial transition
  - Nested Transitions
  - Entry and Exit Transitions
  - Action order and precedence
  - Finding Least Common state to support general transitions

Hierarchical Initialization

- Initial Semantic order:

1. Execute actions of topmost transition
2. For each level of nesting to target state
   - Execute entry actions
3. Execute actions of initial transition of target state

```
void QHsm_init(QHsm *me) {
    QStateHandler t;

    t = (QStateHandler)&QHsm_top;
    do {           /* descend into hierarchy */
        QStateHandler path[QEP_MAX_NEST_DEPTH_];
        int8_t ip = (int8_t)0;

        path[0] = me->state;
        Q_SIG(me) = (QSignal)QEP_EMPTY_SIG_;
        (void)(*me->state)(me);
        while (me->state != t) {
            path[++ip] = me->state;
            (void)(*me->state)(me);
        }
        me->state = path[0];
        Q_SIG(me) = (QSignal)Q_ENTRY_SIG;
        do {        /* retrace in reverse */
            (void)(*path[ip])(me);
        } while ((--ip) >= (int8_t)0);

        t = path[0];
        Q_SIG(me) = (QSignal)Q_INIT_SIG;
    } while ((*t)(me) == Q_RET_TRAN);
    me->state = t;  /* current state update */
}
```

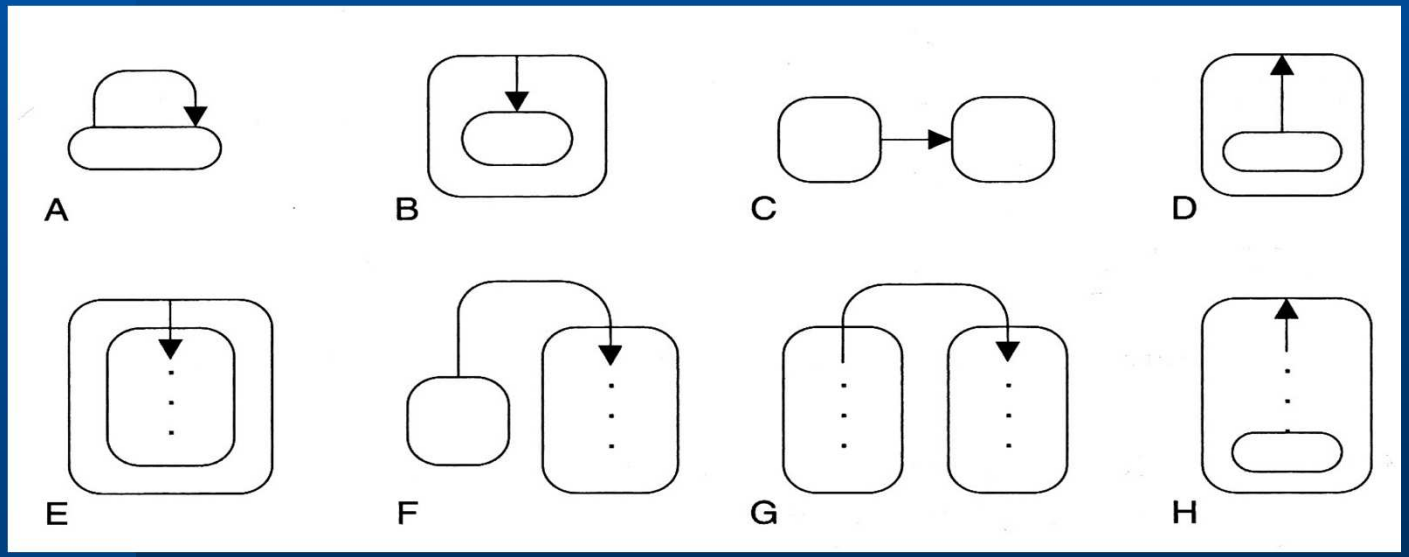
Hierarchical Dispatch Part 1

- Do-loop pushes events to super states if not handled
- Follow required action semantics in transition order
- While-loop allows for super-state transition handling

```
void QHsm_dispatch(QHsm *me) {
 QStateHandler path[QEP_MAX_NEST_DEPTH_];
    QStateHandler s;
    QStateHandler t;
    QState r;

    t = me->state;
    do {       /* recursively invoke handlers */
        s = me->state; r = (*s)(me);
    } while (r == Q_RET_SUPER);

    if (r == Q_RET_TRAN) { /* transition taken */
        int8_t ip = (int8_t)(-1);
        int8_t iq;
        path[0] = me->state; /* target state */
        path[1] = t;

        while (t != s) { /* ascend to trans state */
            Q_SIG(me) = (QSignal)Q_EXIT_SIG;
            if ((*t)(me) == Q_RET_HANDLED) {
                Q_SIG(me) = (QSignal)QEP_EMPTY_SIG_;
                (void)(*t)(me); /* handle actions */
            }
            t = me->state;
        }
```

Hierarchical Dispatch Part 2

- Many possible transition scenarios
  - Must quickly find least common ancestor state of source and target states
  - Choose execution order to heuristically minimize number of invoked state handlers returning EMPTY_SIG : A, B, C, D, E, F, G, H



Hierarchical Dispatch Part 3

```
t = path[0];    /* target of the transition */

if (s == t) {       /* (A) source==target */
    Q_SIG(me) = (QSignal)Q_EXIT_SIG;
    (void)(*s)(me); /* exit the source */
    ip = (int8_t)0; /* enter the target */
}
else {
    Q_SIG(me) = (QSignal)QEP_EMPTY_SIG_;
    (void)(*t)(me); /* find superstate of target */
    t = me->state;
    if (s == t) {   /* (B) source==target->super */
        ip = (int8_t)0;      /* enter the target */
    }
    else {
        Q_SIG(me) = (QSignal)QEP_EMPTY_SIG_;
        (void)(*s)(me); /* find superstate of source */
                        /* (C) source->super==target->super */
        if (me->state == t) {
            Q_SIG(me) = (QSignal)Q_EXIT_SIG;
            (void)(*s)(me); /* exit the source */
            ip = (int8_t)0; /* enter the target */
        }
--more cases--
```

Hierarchical FSM Execution Notes

- Performance Issue:
  - Despite the effort taken to solve common cases quickly, the hierarchy does not change dynamically…
  - It is possible to symbolically traverse all the transitions during compile time so that the machine does not follow the structure of the hierarchy– but immediately goes to the target state
    - Compilers are really good at inlining the several fragments of code from the traversal
    - Resulting code is no longer human readable… but if the code is derived from a clean specification…
    - Could make tool to build very fast dispatch, keeping HSM spec. but lowering cost of implementation (time and space)

Notes on HFSM Design

- Can design HFSM by refinement/elaboration (top-down) or by generalization (bottom-up)
  - For top down design, identify major modes of the system
    - Give each mode a state, create transitions between modes
    - For each mode, consider the behavior sequences that do not lead to mode changes, but require state (i.e. flags)
      - *Preferentially create sub-states to realize these behaviors*
      - *Use guards on signals to realize actions which do not need state update*
    - Identify minimal set of extended state variables
      - *Good Candidates are highly multi-valued (data-like) variables*
  - Once you have designed a significant part of the behaviors
    - Consider common behaviors that can be realized by adding superstates
    - I.e. refactor the decomposition to simplify

More Notes

- For bottom-up design, build states that realize simplest subset of core behavior sequences
  - Look at missing behaviors– determine if they can be generalized by adding a super-state
  - One Key is to identify an event type that is usually handled in a common way
    - Could generalize and realize differences using guarded actions in a common superstate
  - Minimize extended state variables:
    - Good candidates for removal are variables with few values or ones that change behaviors with subsequent events

Quality of Hierarchy

- One quality metric is by analogy to object oriented programming ideas
  - Simplifications arise when objects carry exactly the information to allow correct functioning
    - Beware handling odd cases– want members of a superstate to handle similar behaviors
  - Refactoring structure is done for identical reasons as in OOP
- Another metric is to handle behaviors with similar time scales in orthogonal parts of the code
  - Widely varying time-scales might be good candidates for differing active objects (different concurrent FSMs)

Conclusions

- Hierarchical FSM provide safe way to build efficient reactive machines
  - Safety comes from simplicity of implementing *closed* automata
    - Idea – sensible fallbacks for defaults (unhandled events)
    - HFSM have no death states (uniform top-state)
      - *Usually bad idea for embedded system to terminate…*
  - FSM provide abstraction where you don't care how you reached a state – just what to do when you are there
  - Separating "actions" from "controls" allows for fast dispatch of events
    - Performance of FSM determined by slowest handler function

References

- Further Reading:
  - D. Harel, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, vol. 8 (1987) 231 - 274.
  - D. Harel et al., STATEMATE: A working environment for the development of complex reactive systems, IEEE Transactions on Software Eng.,vol. 16 (1990), no. 4, 403 - 414.
  - Aho, Hopcroft, Ullman “The Design and Analysis of Computer Algorithms”, Addison-Wesley 1974 (Essential FSM manipulation algs.)