UML State Machines

HUSZERL Gábor

huszerl@mit.bme.hu

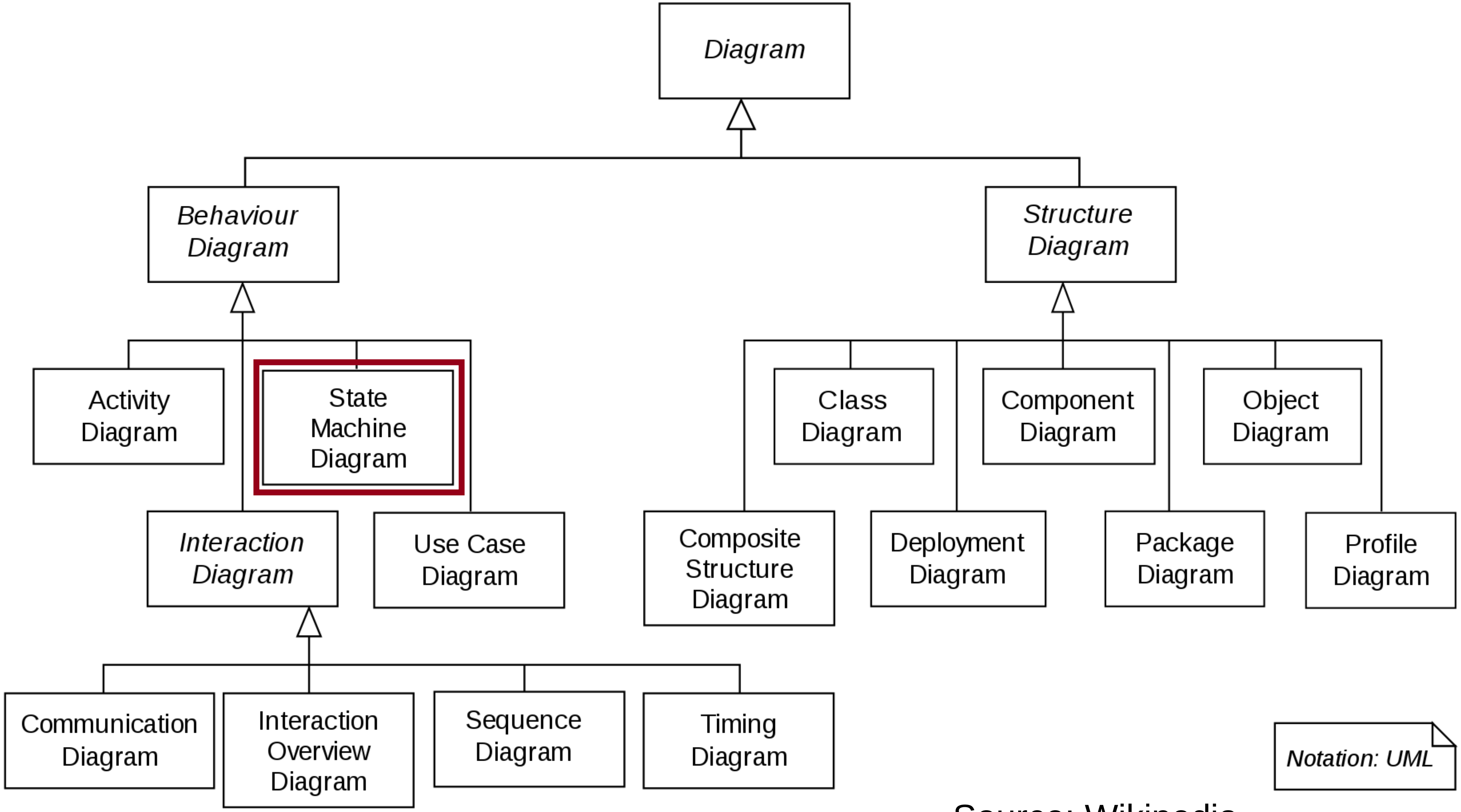
MŰEGYETEM 1782

BME VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR

Méréstechnika és Információs Rendszerek Tanszék

ftsrg Critical Systems Research Group

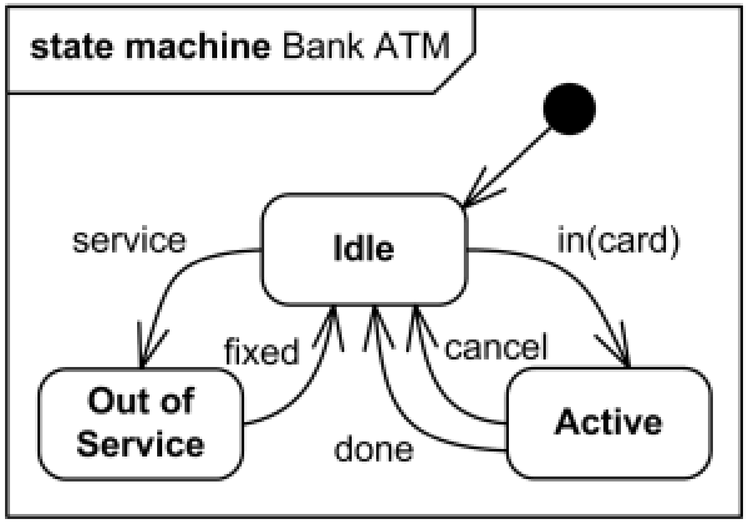
UML Diagram Types



Source: Wikipedia

State Based Modelling

- State
  - Distinguishable from each other
  - Different behaviour in different states
  - Always exactly one active state at a time

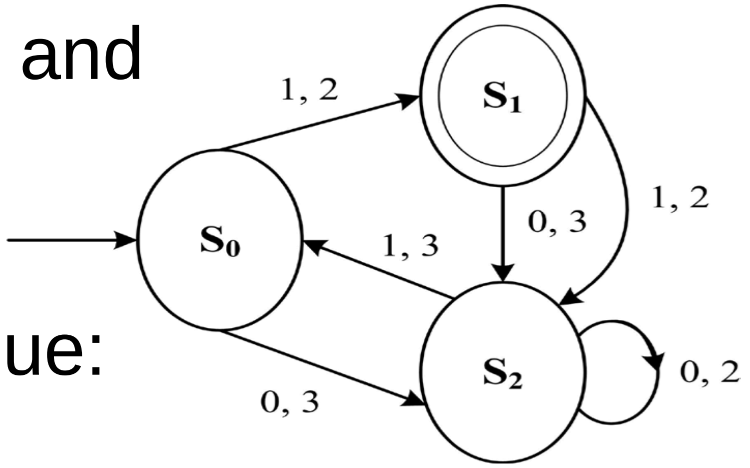


- Reactive systems
  - Event-driven: reacting to external events
  - Reaction depends on the event itself and on the history of the system
  - {Waiting for events – Processing an event} loop

Background of State Machines

$$M = (S, I, O, f, g, s_0)$$

„**Finite State Machine (FSM)**: mathematical abstraction composed of a finite number of states and transitions between those states.”

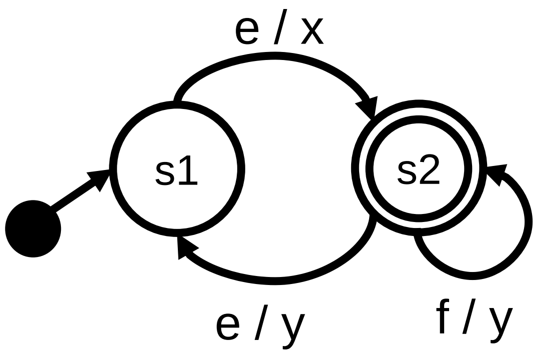


- Long-standing, multipurpose description technique:
  - Hardware elements (→ Digital Technique)
  - Operation of programs (→ Programming 1,2,3)
  - Protocols (→ Communication Networks)
  - Grammars (→ Languages and Automata, MSc)
  - Formal verification (→ Formal Methods, MSc)
- **In common**: states and transitions

Current State	Output		State	
	Input		Input	
	0	1	0	1
S_0	3	2	S_2	S_1
S_1	3	2	S_2	S_2
S_2	2	3	S_2	S_0

Source: SWEBOK

Behaviour Defined by the State Machine



- Set of states:
  - `s1, s2`
- Set of events:
  - `e, f`
- Set of output actions:
  - `x, y`

- Behaviour `:=` set of possible traces
  - trace: series of (input, output) steps

- Example: traces of the state machine on the left
  - `(e, x)`
  - `(e, x), (f, y)`
  - `(e, x), (f, y), (f, y)`
  - `(e, x), (e, y), (e, x)`
  - `...`

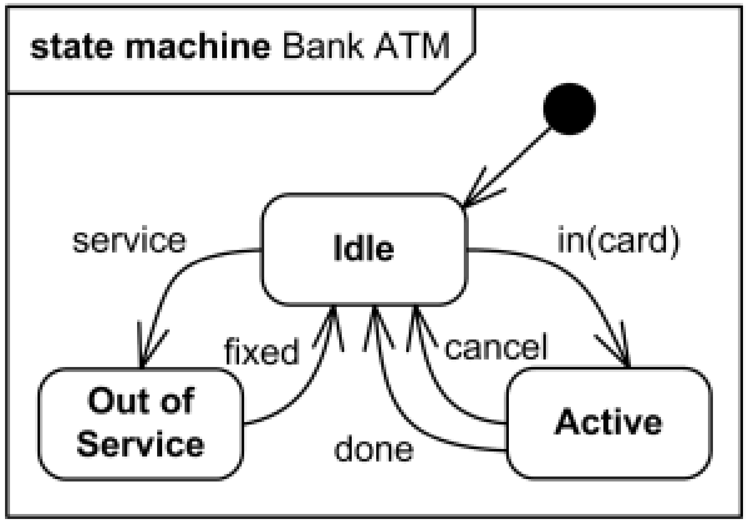
What happens if in state `s1` an event `f` occurs?

How They Can Be Used in Software Systems?

- High-level descriptions of behaviours
  - Description of the whole system or its main components
  - From outside visible behaviour (protocol)
  - Often abstract events and actions

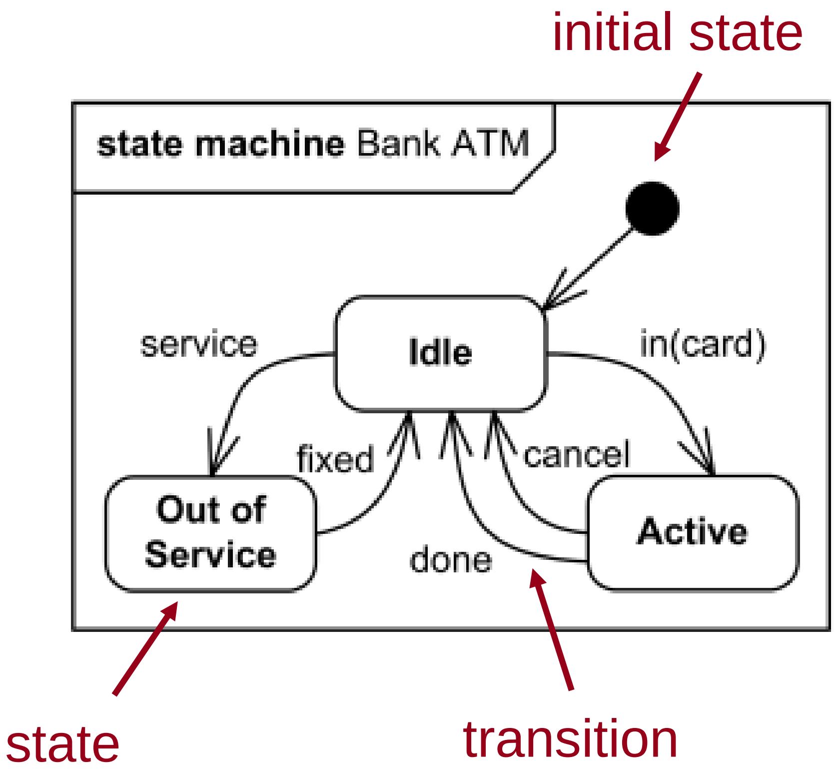
- Low-level descriptions of behaviours
  - Specification of a complex class
  - Describing the main behaviour (classifierBehavior)
  - Events: Reception of a *Signal*, invocation of an *Operation*

Elements



State, Transition

State and Transition

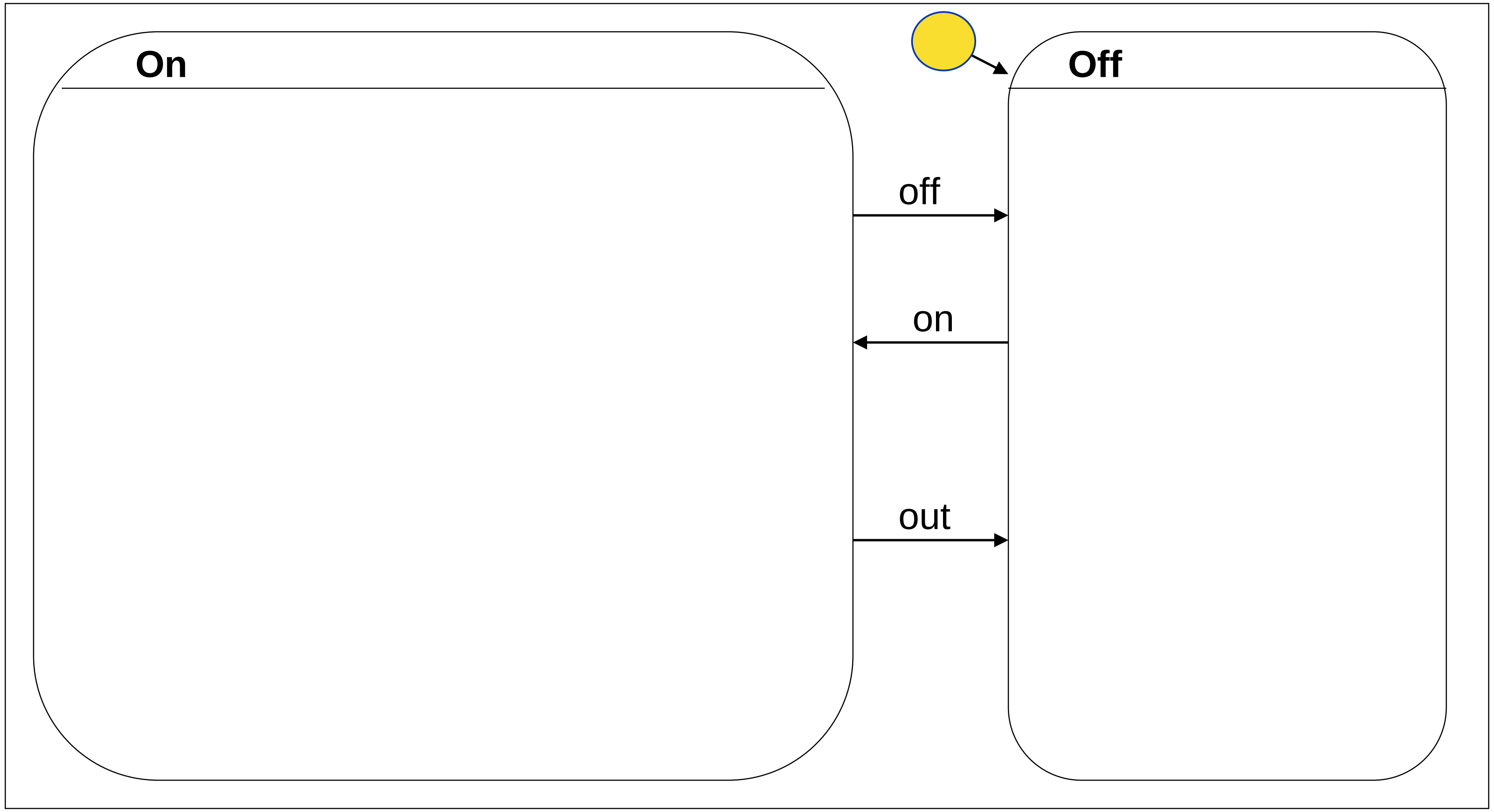


- State
  - (Implicit) invariant holds in it
- Pseudo state
  - Transient states (see later)
  - E.g. initial state
- Transition
  - Between source and target nodes
  - Source and target can be the same
  - `trigger [guard] / effect`

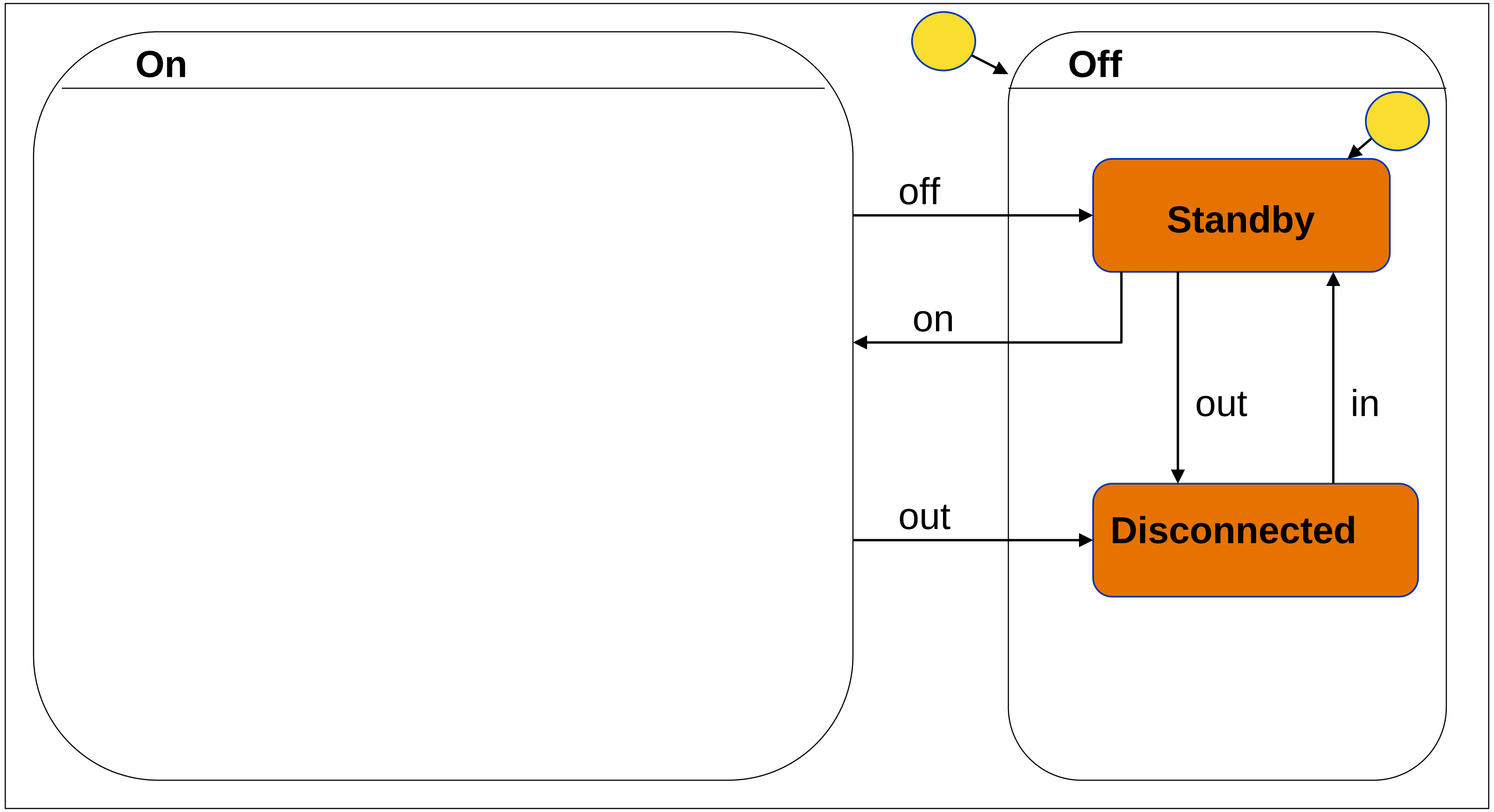
Refinement of States

- Simple state
  - Has no substates
- Composite state
  - Has at least one region (=container for nodes and transitions)
  - OR-type (hierarchic) refinement
    - Has 1 region, in it always exactly 1 state is active
  - AND- type (parallel) refinement
    - Has multiple orthogonal regions, in **each of them** always exactly 1 state is active
- Submachine refinement
  - Embedding a whole state machine

Refinement of States (Example)

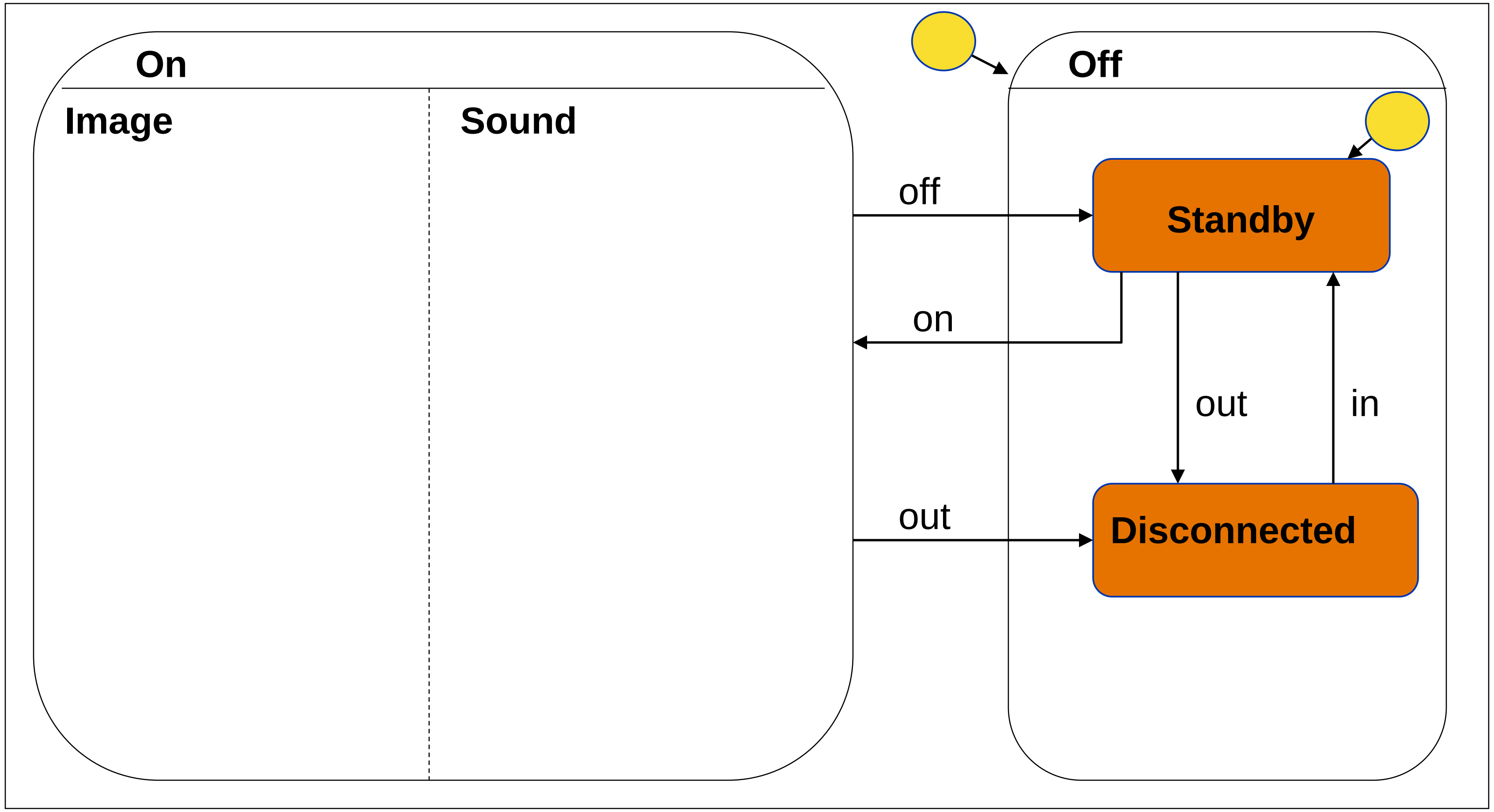


Refinement of States (Example)



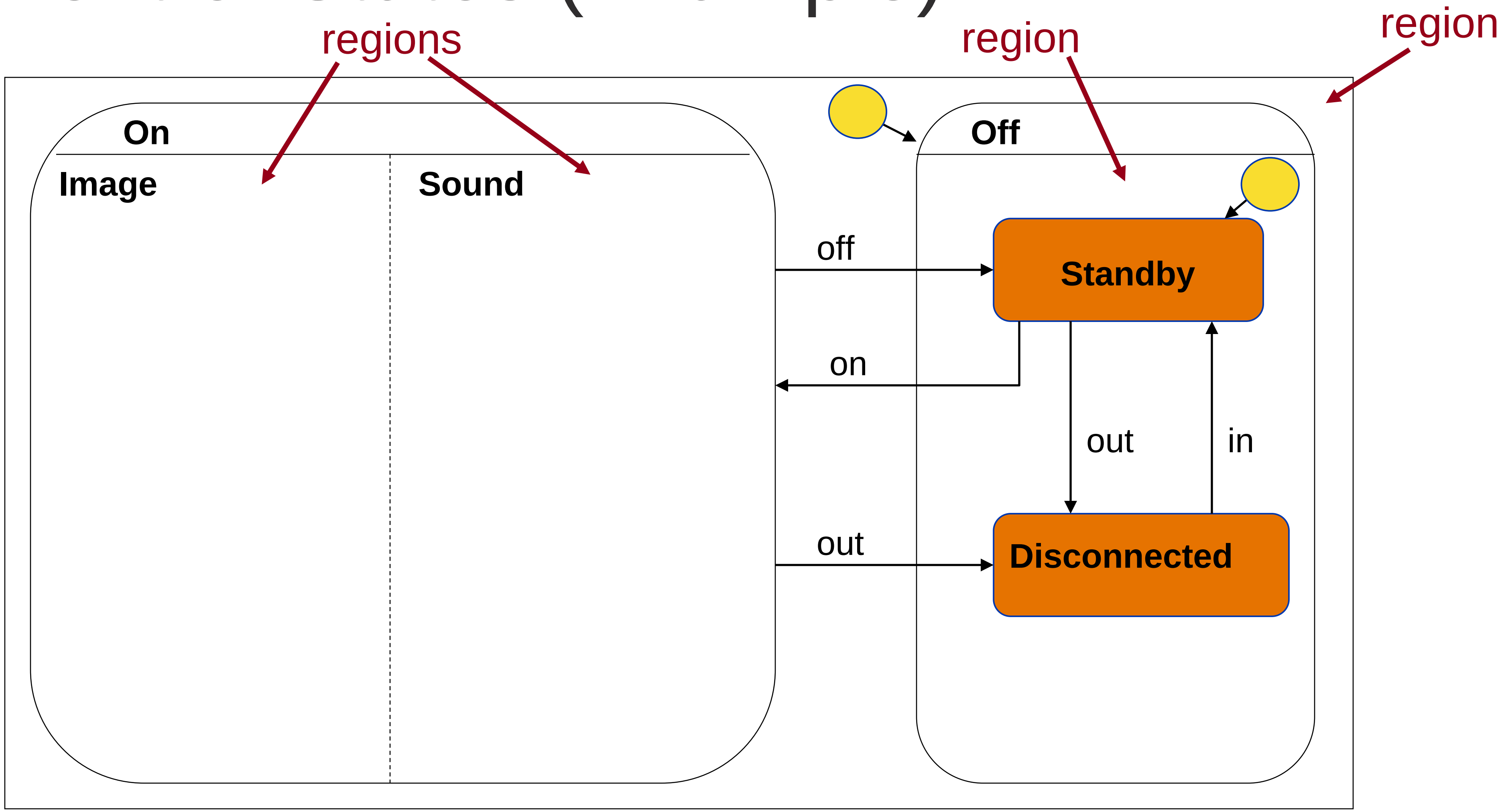
OR-refinement

Refinement of States (Example)



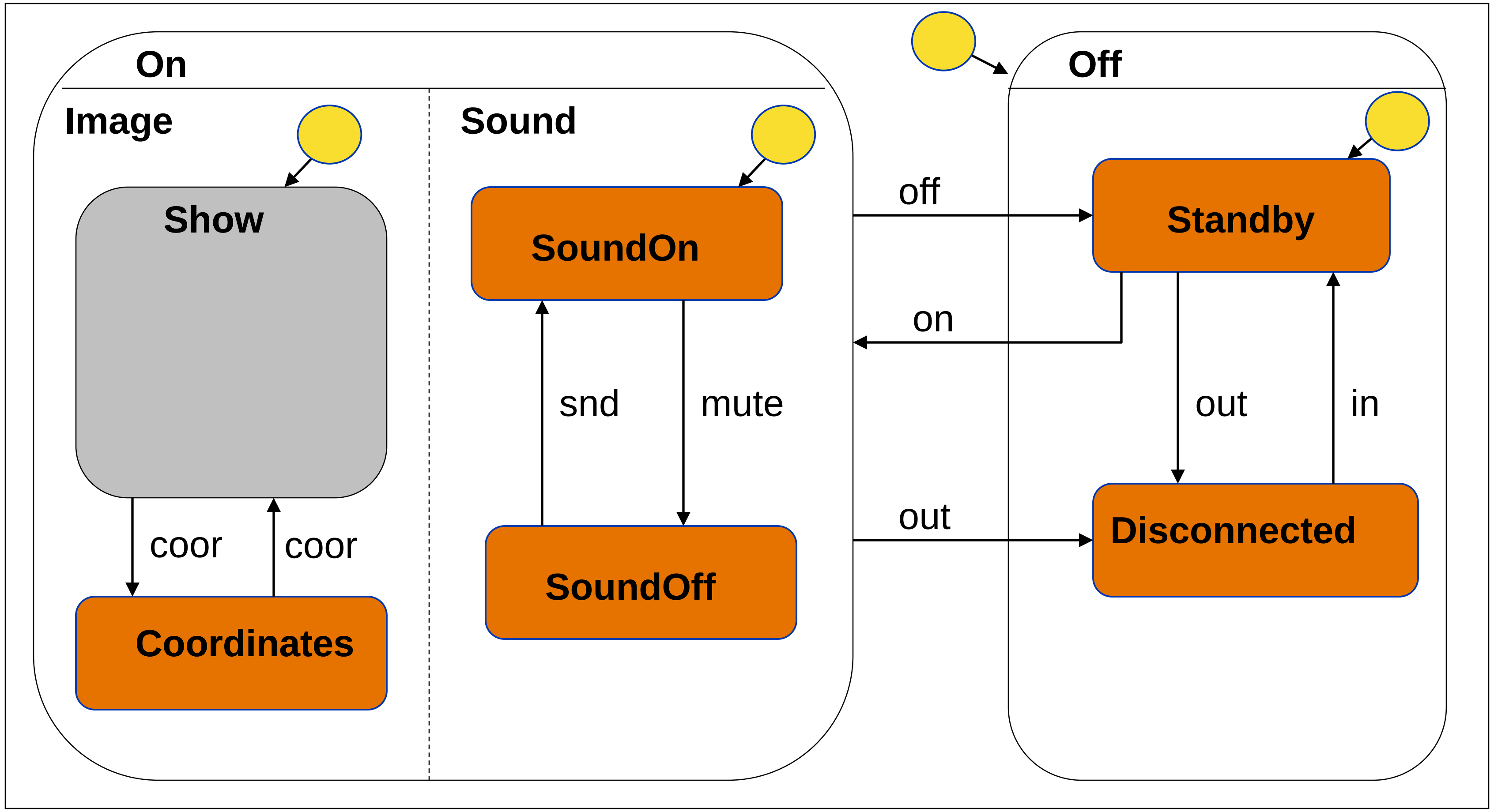
AND-refinement **OR-refinement**

Refinement of States (Example)



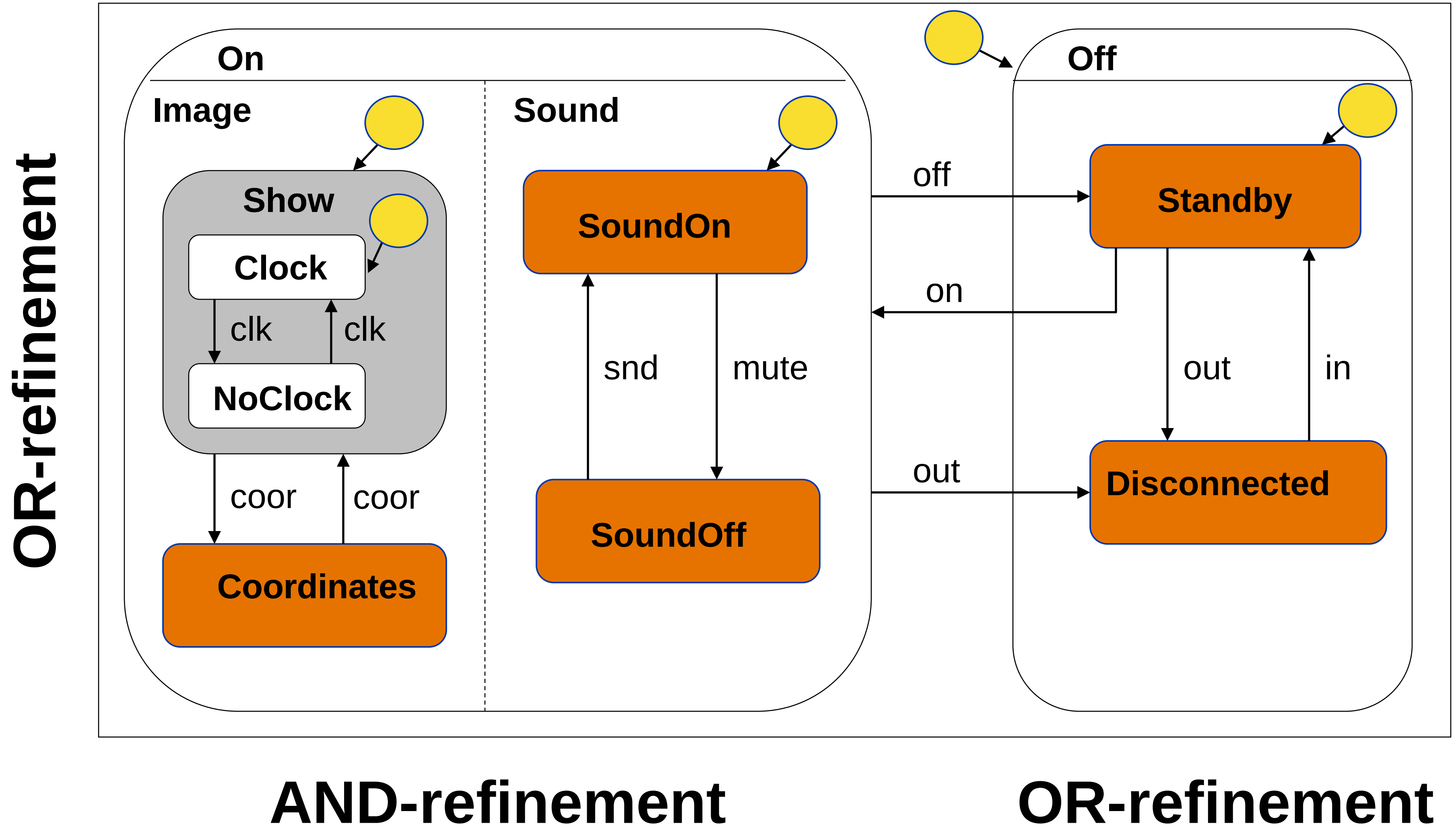
AND-refinement **OR-refinement**

Refinement of States (Example)



AND-refinement **OR-refinement**

Refinement of States (Example)

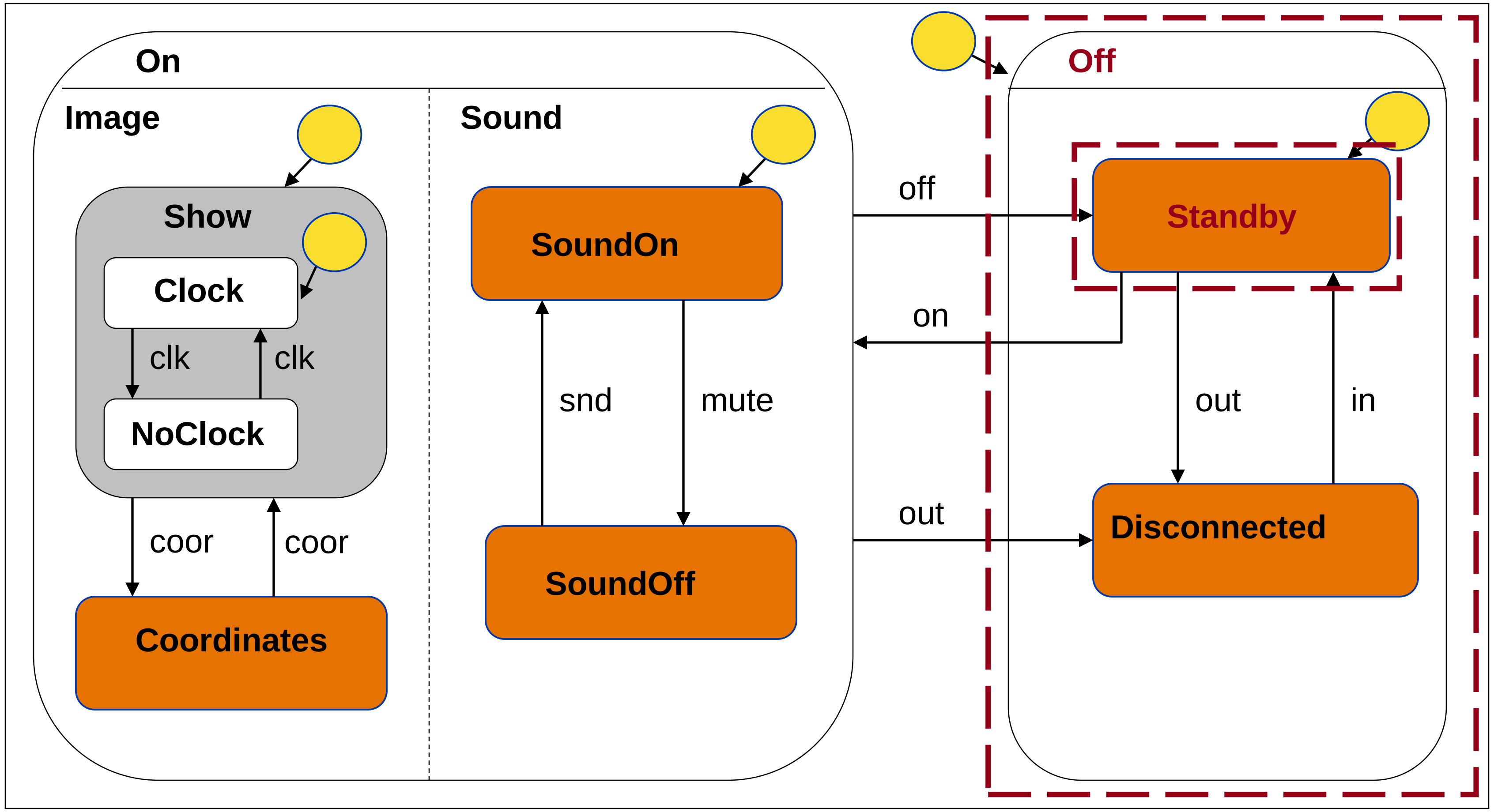


States Configuration

- In composite states: active state configurations (instead of active states)

- Valid configuration:
  - Top-level state machine has always exactly one active state
  - In each active OR-refined state: always exactly one active (sub)state
  - In each active AND-refined state: always exactly one active (sub)state in each region

Valid State Configuration (Example 1)



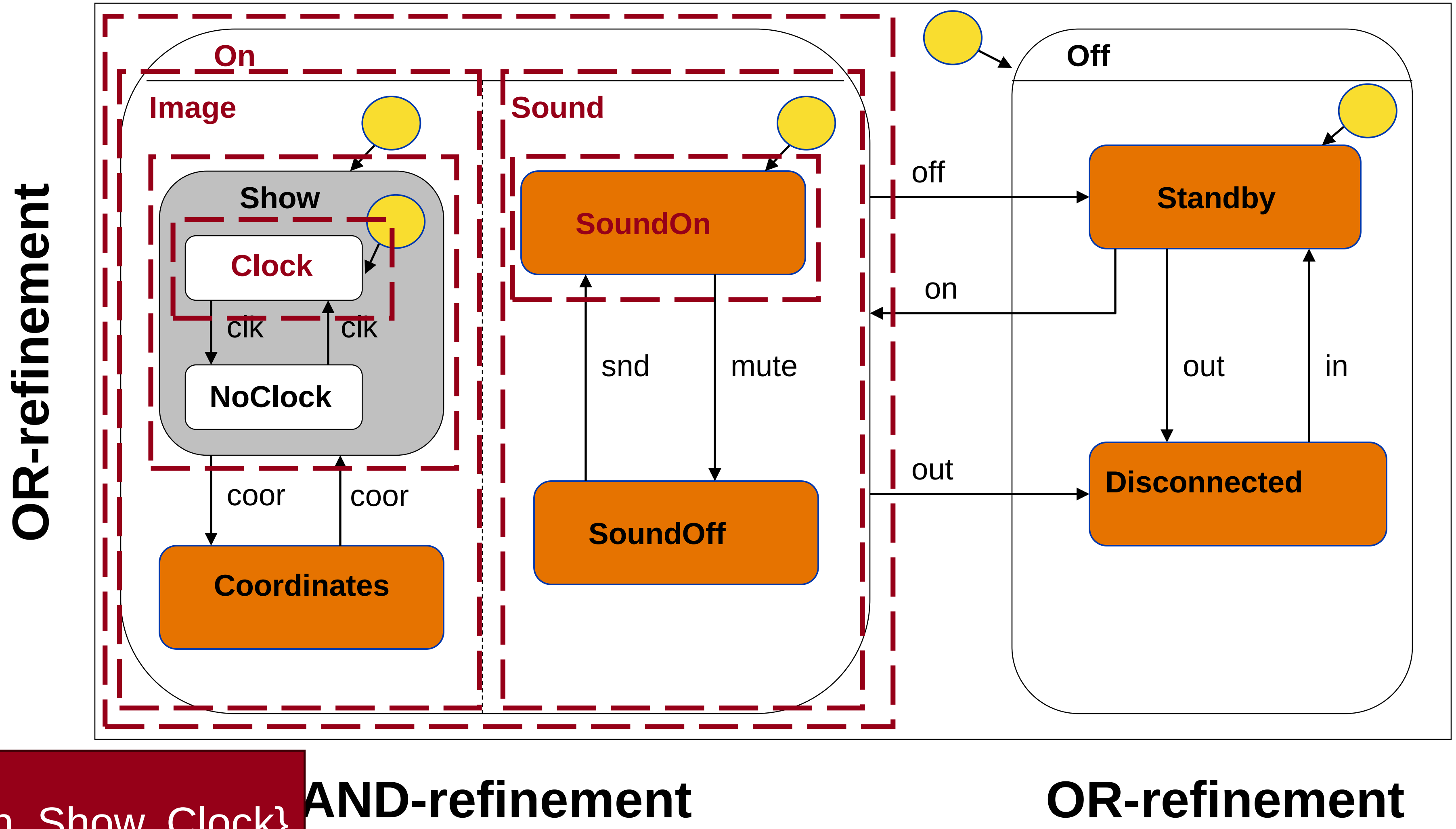
OR-refinement

{Off, Standby}

AND-refinement

OR-refinement

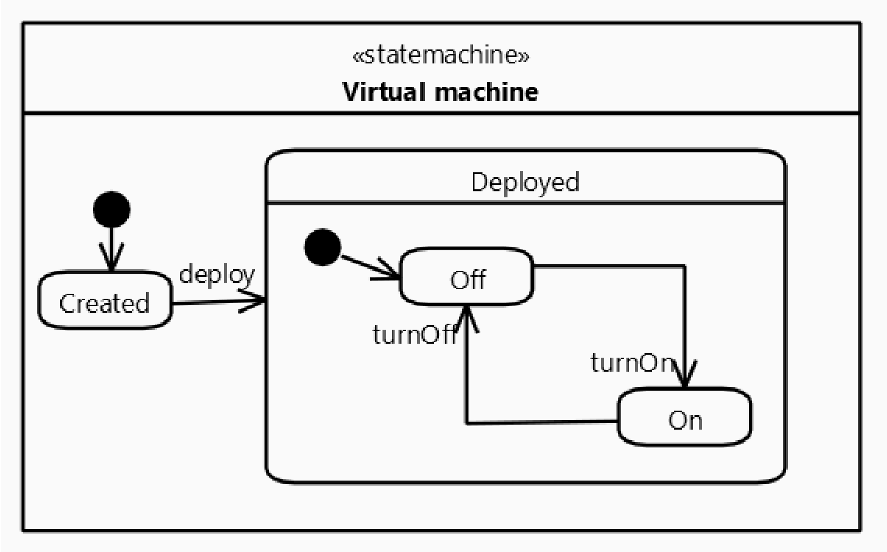
Valid State Configuration (Example 2)



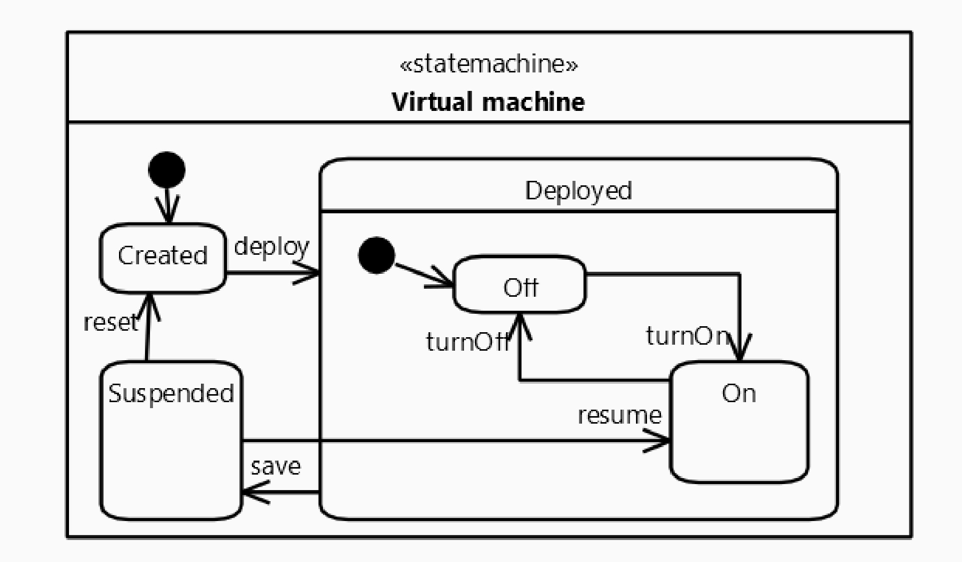
{On, SoundOn, Show, Clock}

Entering a Region

- Default activation
  - Implicit entering: the target of the transition is the containing state
  - The (contained) initial substate will be entered
  - Have a initial state in each region!

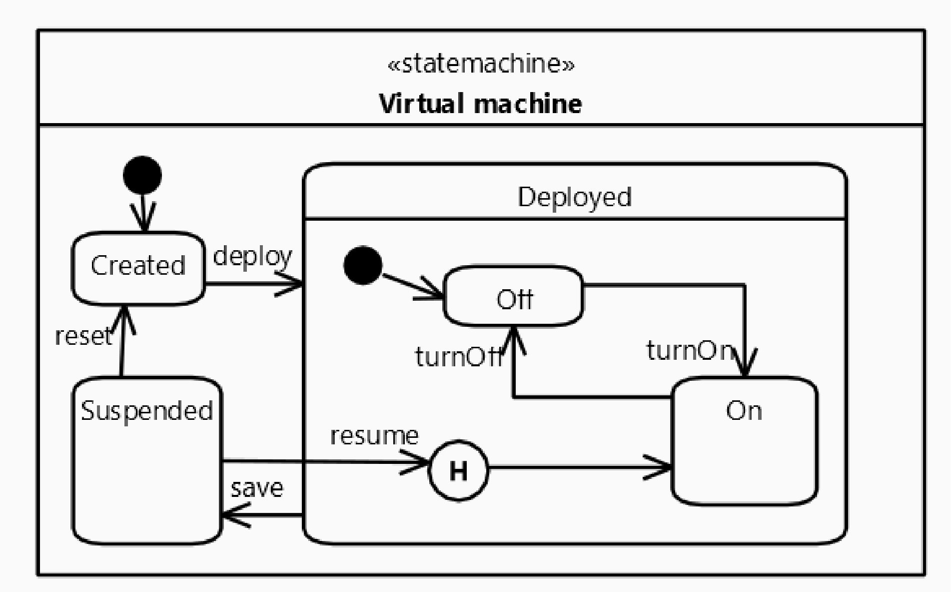


- Explicit activation
  - The target of the transition is a substate
  - The active state is directly specified
  - Think twice before using it!



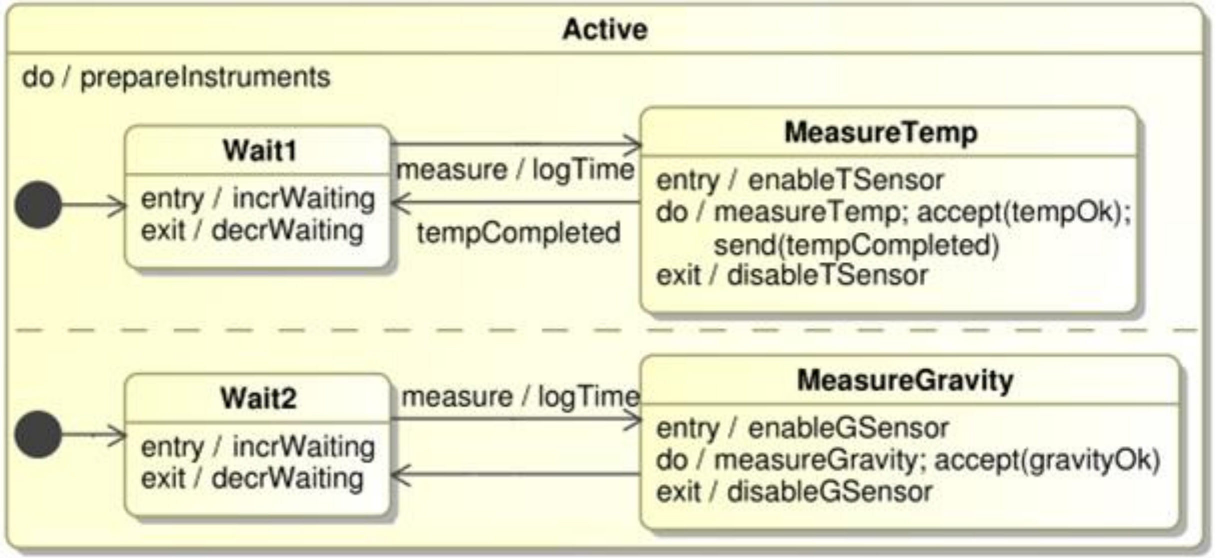
Entering a Region (2)

- History
  - Remembers, from which substate it was left the last time
  - The system can return to that (continue)
  - Kinds:
    - Shallow (H): remembers the first level only
    - Deep (H*): remembers the whole hierarchy of states
  - Transition from the History (optional): default, if there was no “last time”, no previous active state

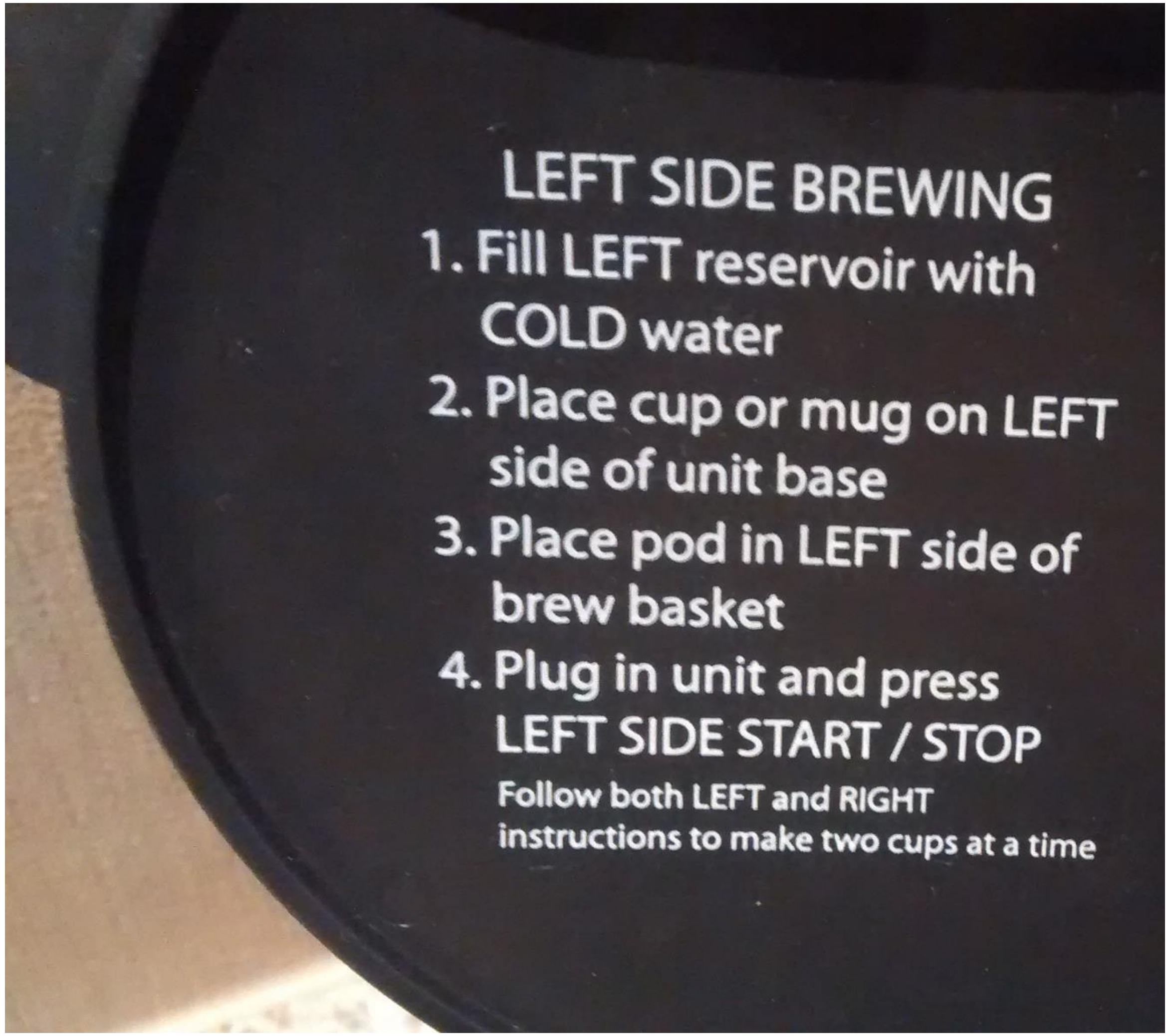


Defining Behaviours Inside the States

- entry behavior
  - when entering the state (from outside)
- exit behavior
  - when leaving the state
- doActivity behavior
  - starts after entering the state
  - Runs until it finishes or until the state is left
  - Parallel to everything else
  - CAUTION: complicated semantics!



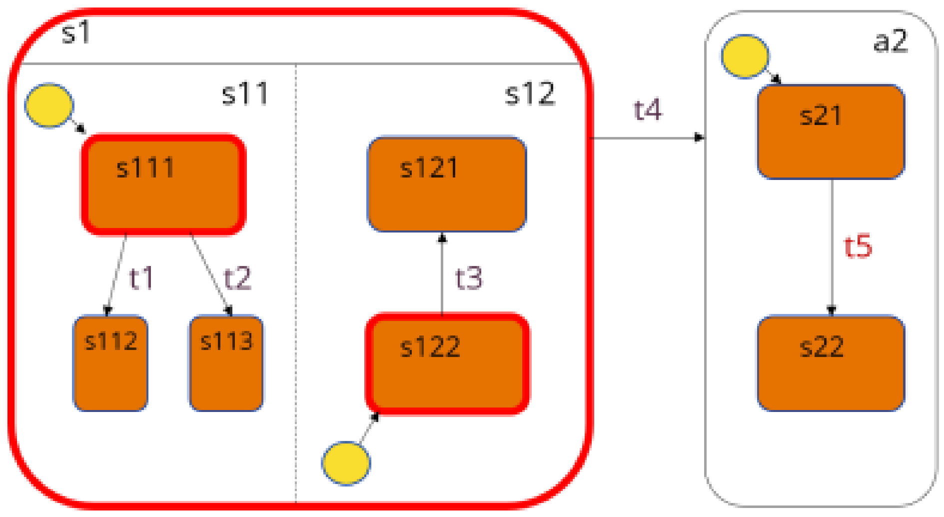
EXERCISE: Coffee Machine



Further Elements (not further specified here)

- Completion transition:
  - implicit trigger: Completion event (activity in source state is finished)
- Deferred events: event is "put off" for later
- Transition kinds: external / local / internal
- Compound transition: "chain" of transitions
- Final state: given region finishes
- Further pseudo states:
  - fork and join, choice and junction
  - entryPoint and exitPoint
  - terminate

(Simplified) Semantics

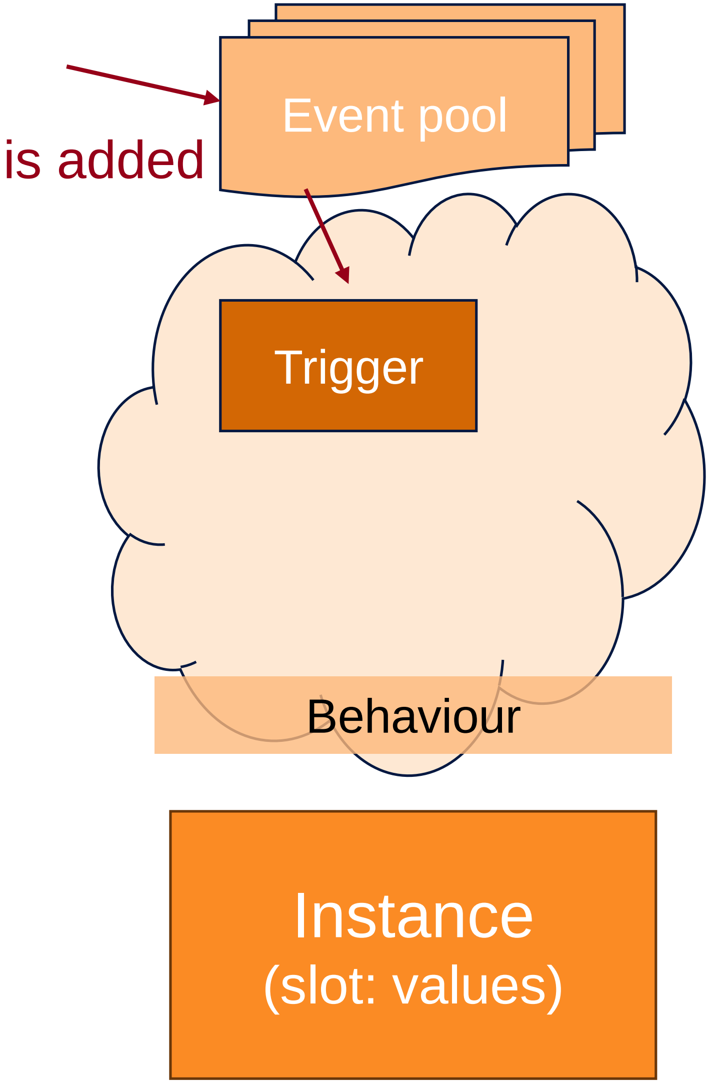


Run-to-completion (RTC)

The Execution of State Machines

- Asynchronously arriving event occurrences
  - Added to the Event pool
  - Processing not necessarily FIFO

- Cyclical operation of the execution:
  - Stable state configuration: waiting
  - Event processing step:
    - Dispatch: selection from pool, processing
    - **„run-to-completion” (RTC)**: processing only one event at a time, it cannot take another event before it was completed
    - Reaching a new stable state configuration



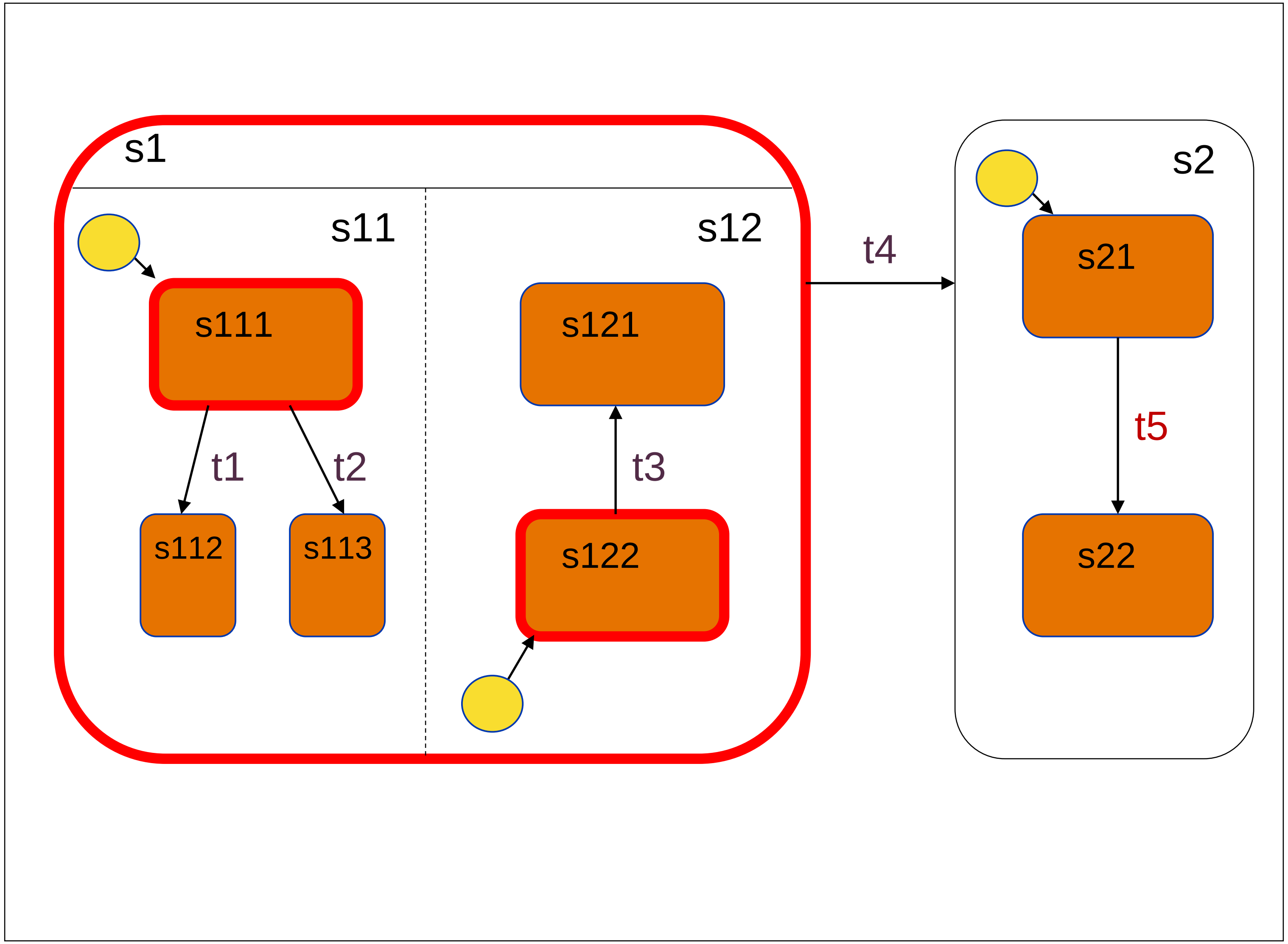
Event Processing (1)

1. Selecting an event occurance
2. Collecting the enabled transitions
   - Each source of it is active AND it is triggered by the selected event AND its guard condition evaluates to true
3. Depending on the number of enabled transitions:
   - If 0: event is dropped, the RTC is completed
   - If 1: that transition is fired (executed)
   - If >1:
     - Identifying the conflicting transitions, resolving, choosing from them
       - Conflict: the intersection of the sets of states they would exit is not empty
     - The ones in different orthogonal regions may fire in the same step

Event Processing (2)

4. Conflict resolution: based on priorities
   - Transition originating from a substate has a higher priority than one originating from its superstate
   - It is a partial solution only, it does not resolve all conflicts
5. Selecting the transitions to be fired
   - Conflict-free sets with no higher priority transitions left out
   - If there are several such sets, then non-deterministic choice
6. Executing firing
   - Exiting the source states, executing the exit behaviours (from inside out)
   - Executing the effects of the transition
   - Entering the target states, executing the entry behaviours (from outside in)

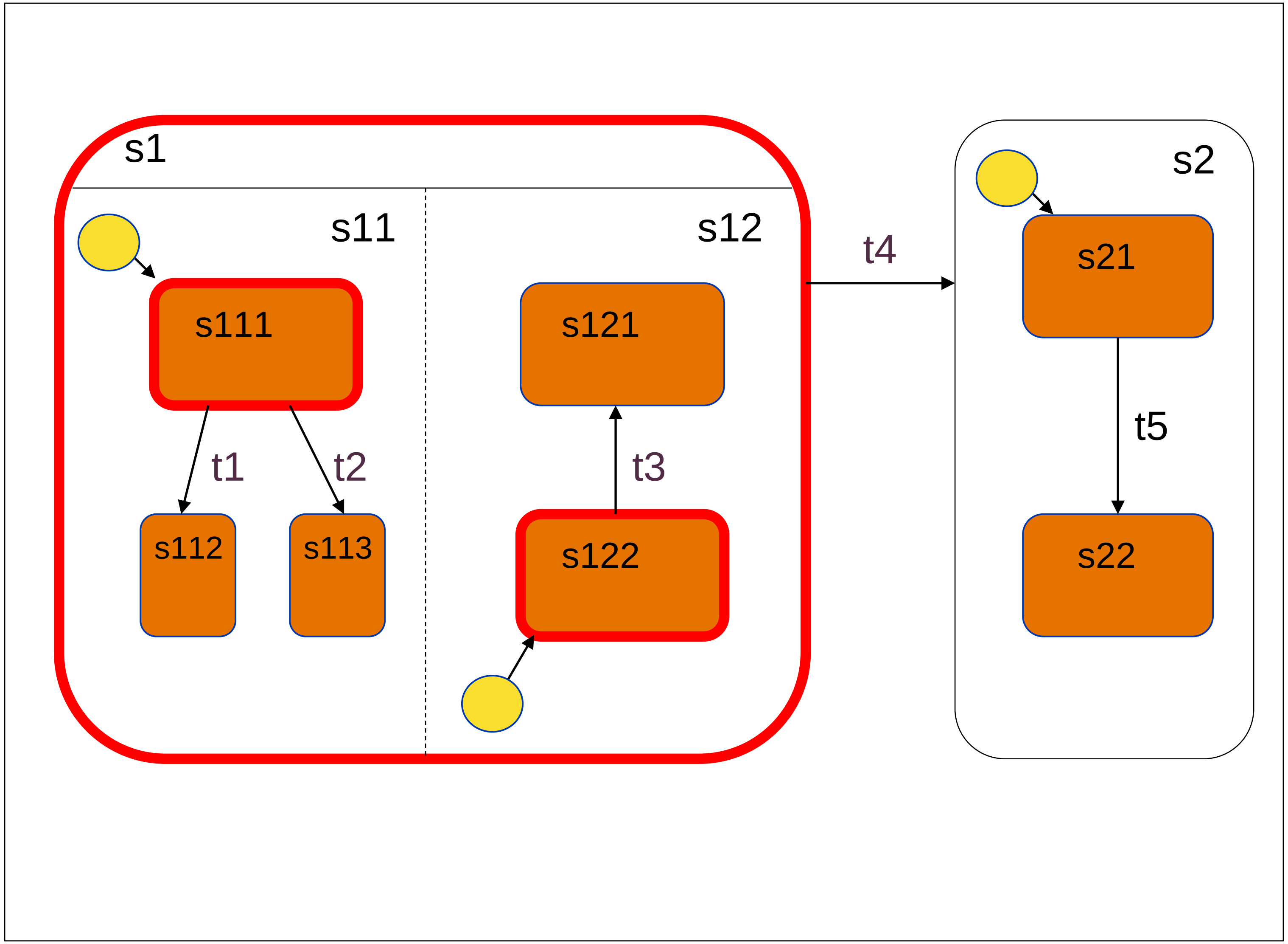
Enabling, Priority, Selection (Example)



- Each `t*` transition is triggered by event e
- State configuration:
  - {s1, s111, s122}
- e event dispatch

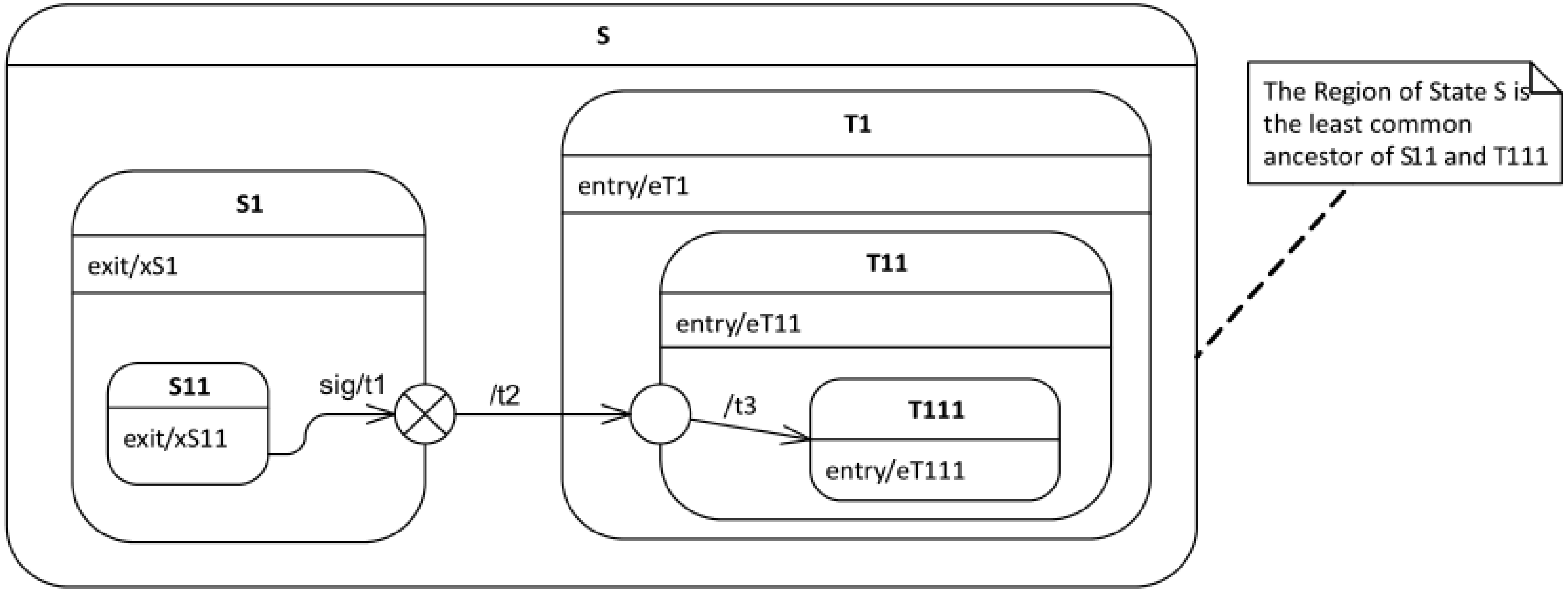
- Enabled transitions:
  - t1, t2, t3, t4
- Conflicting one:
  - {t1, t2}; {t1, t4}; {t2, t4}; {t3,t4}
- Not in conflict:
  - {t1, t3}, {t2, t3}

Enabling, Priority, Selection (Example)



- Priorities:
  - t1 > t4, t2 > t4, t3 > t4
- Fireable sets:
  - {t1, t3}
  - {t2, t3}
- One of them will be chosen

Execution of Firing (Example)



- State configuration:
  - {S, S1, S11}
- Event: sig
- Elements of the execution of firing:
  - xS11, t1, xS1, t2, eT1, eT11, t3, eT111

Good Practices

	fuel	abort	launch	land
Ready	Fueled/-	-/-	-/-	-/-
Fueled	-/-	Ready/-	Flying/ ignite engine	-/-
Flying	-/-	-/-	-/-	Landed/ shut down engine
Landed	-/-	-/-	-/-	-/-

Advices to state machines

How to Model with State Machines?

- Collect the input events and output actions
- Start with simple states
  - What are the different operational modes?
- Refine gradually (OR- or AND-refinement)
  - Have an initial state in each region
  - Move the common behaviour to the containing superstate
- Extend the transitions
  - Add effects and other behaviours
  - Resolve the initial non-determinisms, if possible
  - Completeness: are there any state+event pairs without specified behaviour?

Completeness and Unambiguity

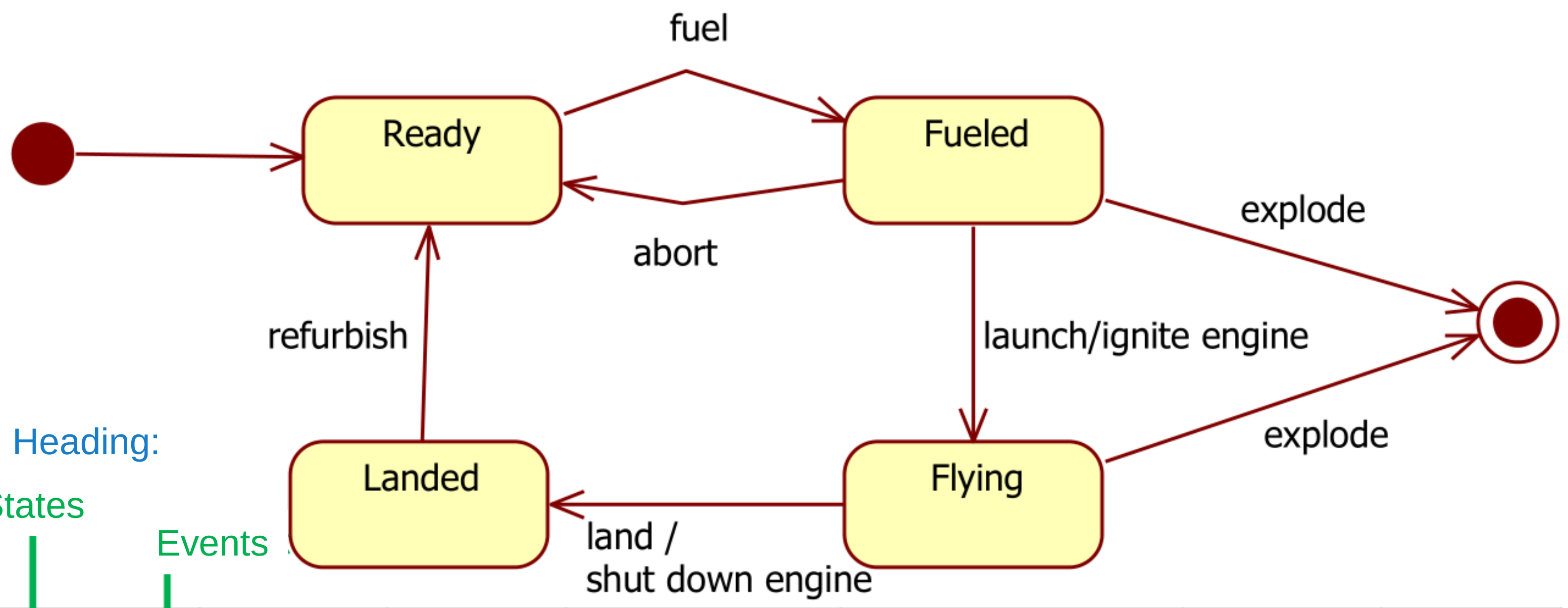
There is no non-specified behaviour

- Completeness
  - For each <state configuration, guard evaluation, event> triplet …
  - … there is at least 1 behaviour defined

- Unambiguity
  - For each <state configuration, guard evaluation, event> triplet …
  - … there is at most 1 behaviour defined

There is no non-deterministic behaviour

Specifying by State Table



Heading:

States

Events

	fuel	abort	launch	land	refurbish	explode
Ready	Fueled/-	-/-	-/-	-/-	-/-	-/-
Fueled	-/-	Ready/-	Flying/ ignite engine	-/-	-/-	-/-
Flying	-/-	-/-	-/-	Landed/ shut down engine	-/-	-/-
Landed	-/-	-/-	-/-	-/-	Ready/-	-/-

Cells: Next State / Action

In case of simple states, it helps checking completeness and unambiguity

Implementing State Machines

```
public class SomeThing {

    int s = 0;

    public void process(E e) {
        if (s==0) {
            if (e == E.N) s = 1;
        } else if (s==1) {
            if (e == E.S) s = 2;
            else if (e == E.M) s = 3;
        } else if (s==2) {
            if (e == E.I) s = 0;
            else if (e == E.F) s = 4;
        } else if (s==3) {
            if (e == E.I) s = 0;
        }
    }
}
```

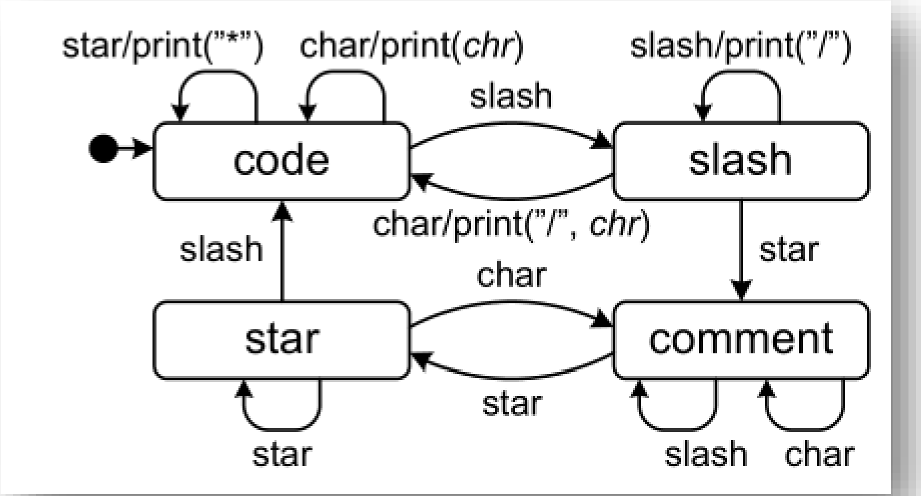
Implementing State Machines

- Patterns depending on the target programming language
  - Embedded code: no hierarchy, simple control, limited storage capacity
  - High-level languages: even on OO basis

- Typical patterns:
  - Mapping a simple state machine to an embedded switch
  - State table: two-dimensional array (states x events)
  - “State” design pattern

- It may be worth generating the code!

Example: Nested Switch Patterns

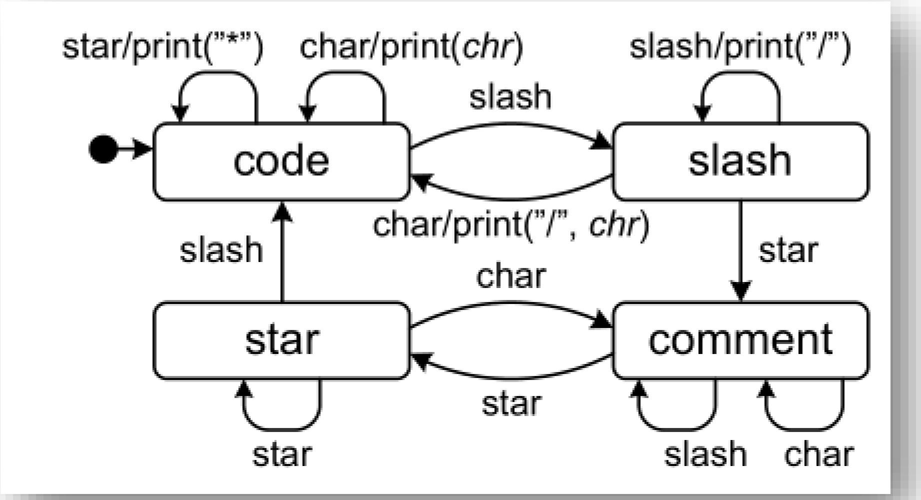


Source: Pintér Gergely. *Model based program synthesis and runtime error detection for dependable embedded systems.* PhD thesis, 2007.

```
void
nested_switch_implementation(const char* input) {

    const char *input_ptr;
    state_t state = S_CODE;

    for (input_ptr = input; *input_ptr != '\0'; input_ptr += 1) {
        trigger_t trigger = translate_trigger(*input_ptr);

        switch (state) {
        case S_CODE:
            switch (trigger) {
            case T_STAR:           /* Output the character. */
            case T_CHAR:
                printf("%c", *input_ptr);
                break;

            case T_SLASH:          /* Transition to state 'SLASH'. */
                state = S_SLASH;
                break;
            }
            break;

        case S_SLASH:
            switch (trigger) {
            case T_STAR:           /* Transition to state 'COMMENT'. */
                state = S_COMMENT;
                break

            case T_SLASH:          /* Output a slash character. */
                printf("/");
                break;

            case T_CHAR:           /* Output and transition to state 'CODE'. */
                printf("/%c", *input_ptr);
                state = S_CODE;
                break;
            }
            break;
[...]
```

Listing B.2: Implementation by the Nested Switch Pattern

Example: State Table Patterns



Source: Pintér Gergely. *Model based program synthesis and runtime error detection for dependable embedded systems.* [PhD thesis](), 2007.

```c
/** Pointer to activity function type. */
typedef void (*activity_t)(const char input_char);

/** Type of a cell in a state table. */
typedef struct {
    state_t next_state;          /* The next state to be stepped to. */
    activity_t activity;         /* The activity to be preformed. */
} cell_t;

void
state_table_implementation(const char* input) {

    const char *input_ptr;
    state_t state = S_CODE;

    const cell_t table[NUM_STATE][NUM_TRIGGER] =  {
        /* State 'CODE' (triggers: star, slash, char) */
        {{S_CODE, act_out_chr}, {S_SLASH, NULL}, {S_CODE, act_out_chr}},
        /* State 'SLASH' (triggers: star, slash, char) */
        {{S_COMMENT, NULL}, {S_SLASH, act_out_slash}, {S_CODE, act_out_slash_chr}},
        /* State 'COMMENT' (triggers: star, slash, char) */
        {{S_STAR, NULL}, {S_COMMENT, NULL}, {S_COMMENT, NULL}},
        /* State 'STAR' (triggers: star, slash, char) */
        {{S_STAR, NULL}, {S_CODE, NULL}, {S_COMMENT, NULL}}
    };

    for (input_ptr = input; *input_ptr != '\0'; input_ptr += 1) {
        const cell_t *cell;
        trigger_t trigger = translate_trigger(*input_ptr);

        cell = &(table[state][trigger]);
        if (NULL != cell -> activity)
            (cell -> activity)(*input_ptr);

        state = cell -> next_state;
    }
}
```

Listing B.3: Implementation by the State Table Pattern

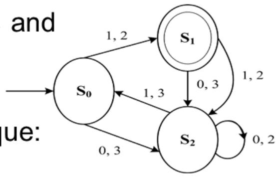
Summary

Summary

Background of State Machines

$M = (S, I, O, f, g, s_0)$

„**Finite State Machine (FSM)**: mathematical abstraction composed of a finite number of states and transitions between those states.”

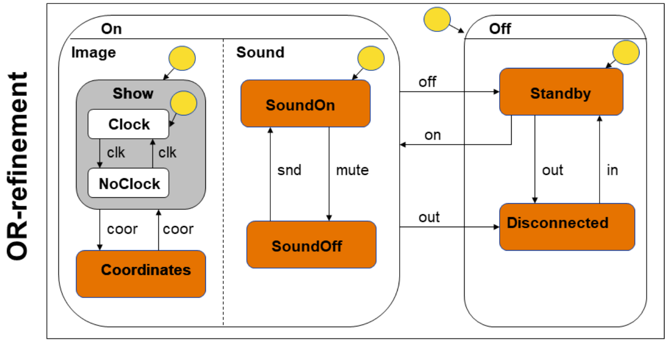


- Long-standing, multipurpose description technique:
  - Hardware elements (→ Digital Technique)
  - Operation of programs (→ Programming 1,2,3)
  - Protocols (→ Communication Networks)
  - Grammars (→ Languages and Automata, MSc)
  - Formal verification (→ Formal Methods, MSc)
- **In common**: states and transitions

Current State	Output Input		State Input	
	0	1	0	1
S_0	3	2	S_2	S_1
S_1	3	2	S_2	S_2
S_2	2	3	S_2	S_0

Source: SWEBOK

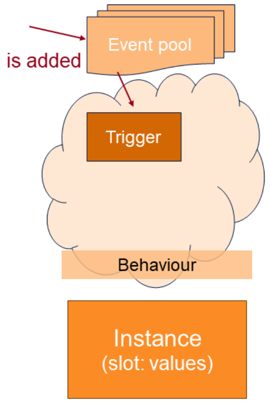
Refinement of States (Example)



AND-refinement **OR-refinement**

The Execution of State Machines

- Asynchronously arriving event occurrences
  - Added to the Event pool
  - Processing not necessarily FIFO
- Cyclical operation of the execution:
  - Stable state configuration: waiting
  - Event processing step:
    - Dispatch: selection from pool, processing
    - **„run-to-completion” (RTC)**: processing only one event at a time, it cannot take another event before it was completed
    - Reaching a new stable state configuration



How to Model with State Machines?

- Collect the input events and output actions
- Start with simple states
  - What are the different operational modes?
- Refine gradually (OR- or AND-refinement)
  - Have an initial state in each region
  - Move the common behaviour to the containing superstate
- Extend the transitions
  - Add effects and other behaviours
  - Resolve the initial non-determinisms, if possible
  - Completeness: are there any state+event pairs without specified behaviour?