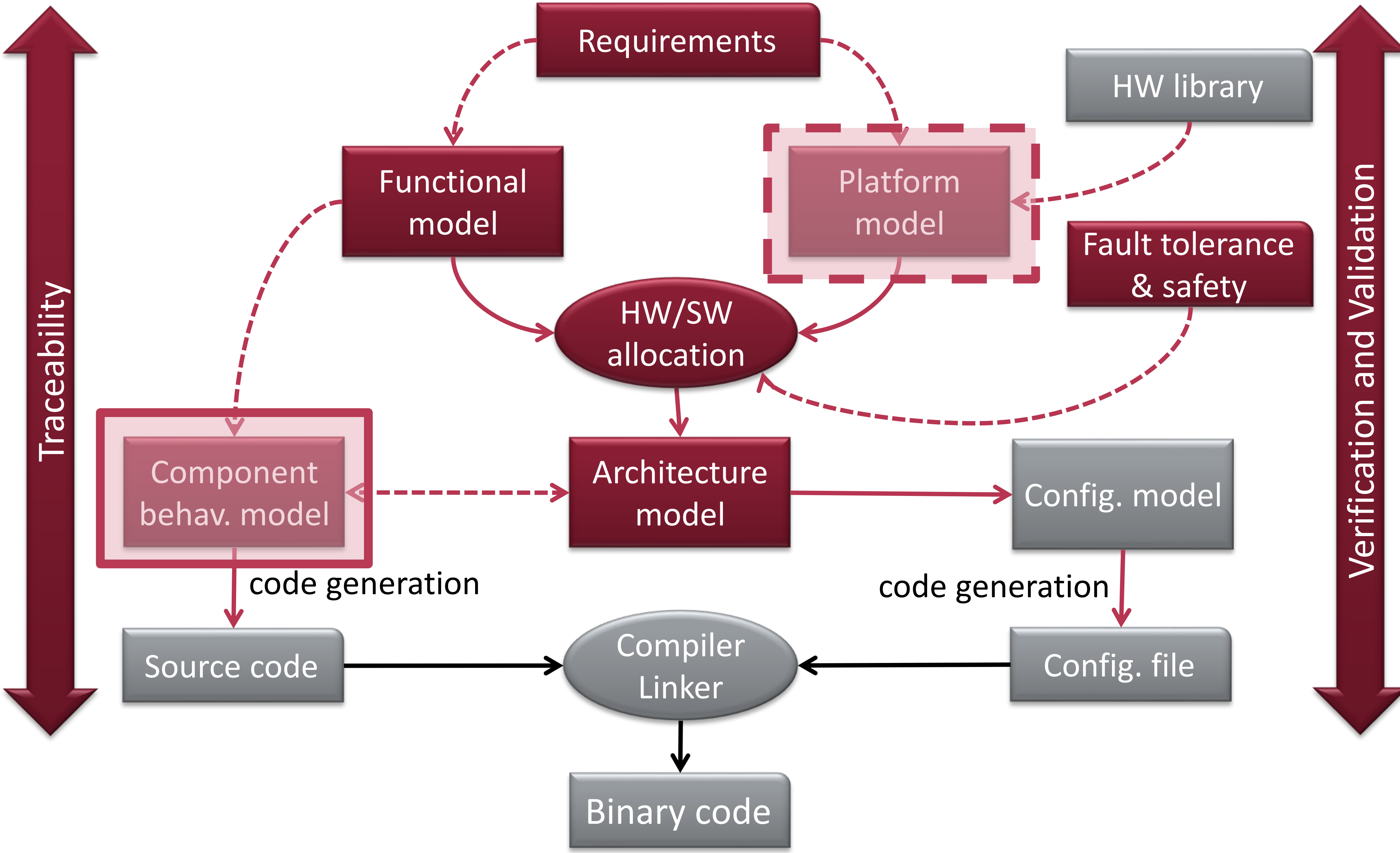
Reactive behavioral modeling

Systems Engineering BSc Course

Budapest University of Technology and Economics
Department of Measurement and Information Systems

Platform-based systems design



Learning Objectives

Reactive behavioral modeling

Understand the basic blocks of reactive component design

Identify the events, states and actions to describe component behavior

Understand the syntactic building blocks of UML State Machines

Understand the semantics of UML State Machines

Use hierarchy to structure the models and express abstraction-refinement of states

Build clean and expressive models by using best practices

Code generation

- Understand the main ideas of different approaches
- Understand the advantages and disadvantages of different approaches

PREVIOUSLY… (SYSTEM MODELING)

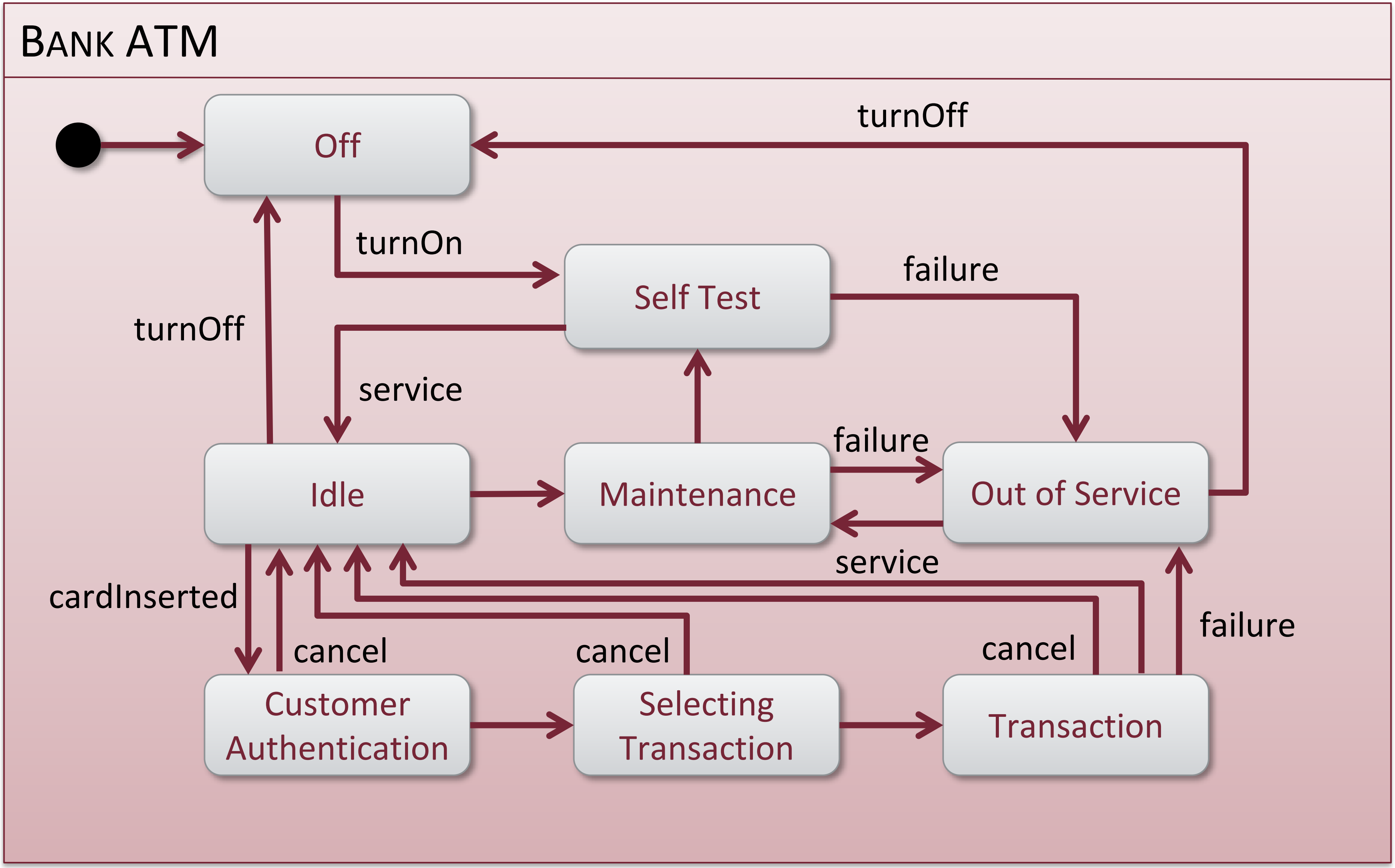
State machines
Hierarchical state refinement
Parallel/Orthogonal regions

State Space

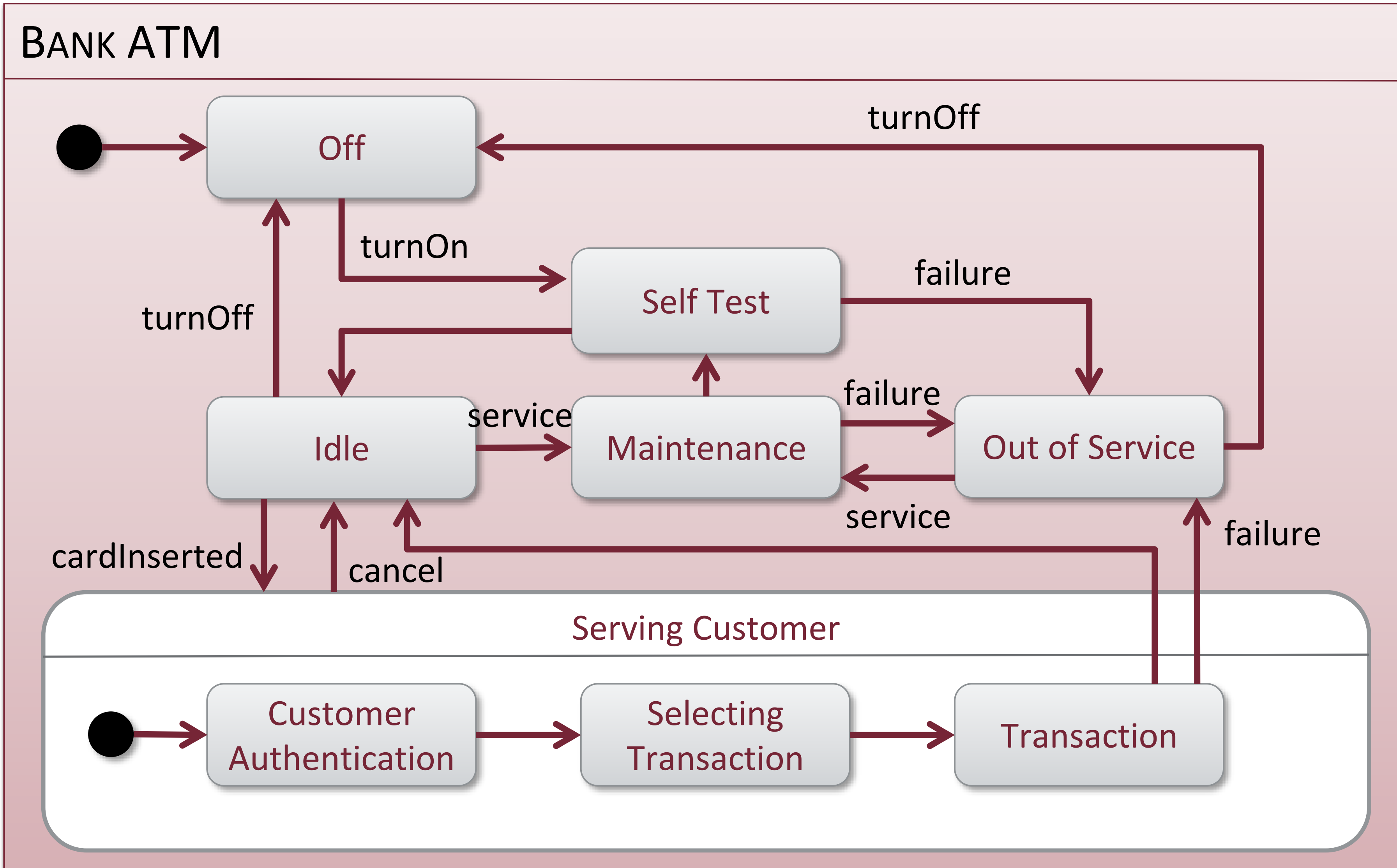
- **State space**
  - A set of distinct **system** states
  - **DEF**: The state space is a set such that in every moment, the system can be described by exactly one element.

- **Current state**
  - **DEF**: At a given moment, the **current state** of the system is the single element of the state space that describes the system in that moment.

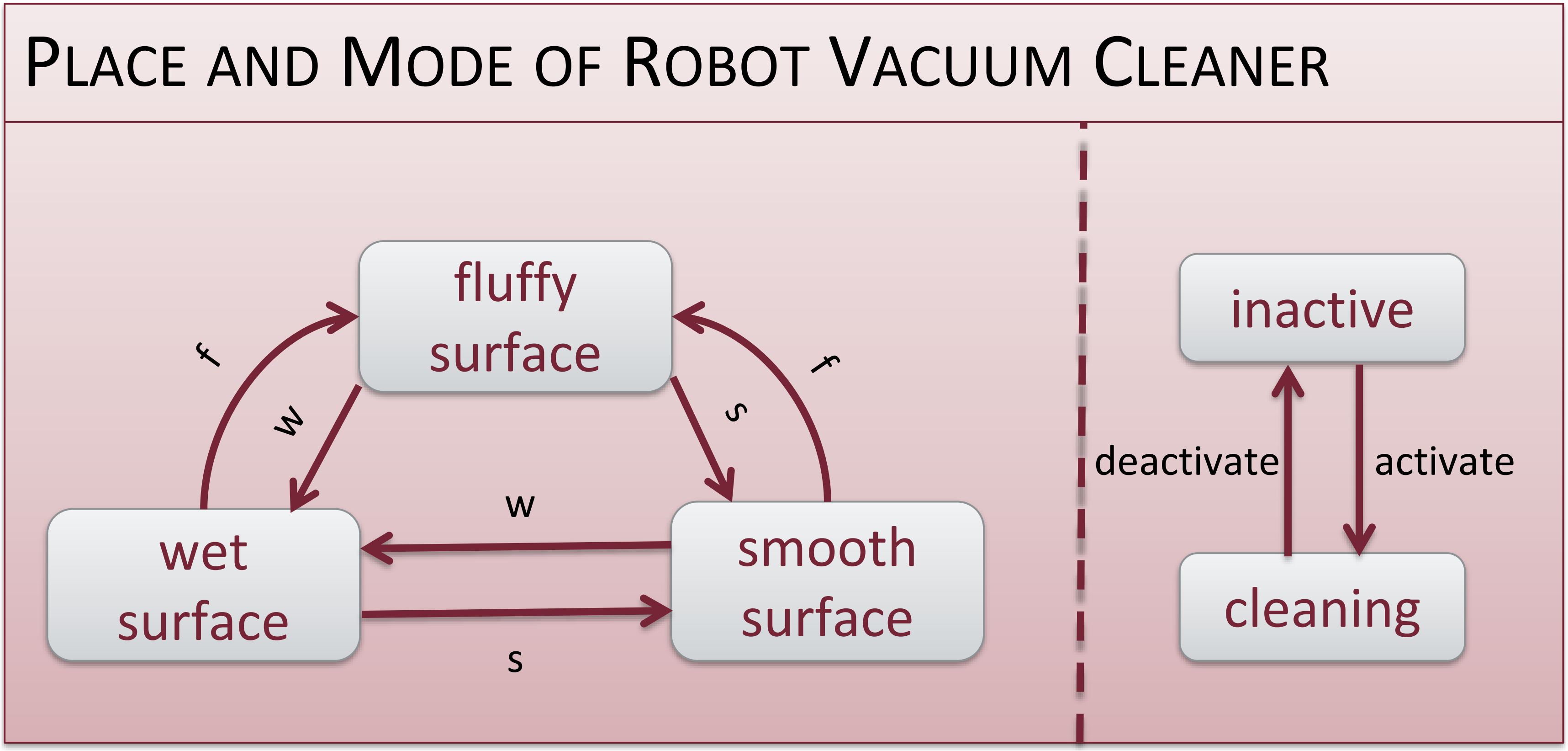
State machine



Hierarchical state machine



Parallel/orthogonal regions



REACTIVE COMPONENTS

Event, Event queue

State, State variable

Transition, Action

Event-oriented approach

- Classic programs:
  - Input parameters, processing, output
  - See: Activity diagram
- **Reactive systems:**
  - Behavior is triggered by *events*
  - The system reacts to its environment
  - Continuous operation
    - Idle state: waiting for events
- Examples:
  - Most GUIs, *Active Object* pattern, Web services

Events

- **Event:**
  - *Asynchronous* occurrence with optional parameters
  - E.g. mouse click + coordinates
- **Event queue:**
  - Events are placed in an event queue in the order of occurrence
  - The reactive systems processes and reacts to them one-by-one
- *Quiz: Can two asynchronous events occur precisely at the same time?*

States

- Can reactions depend on previous events?
  - No → Stateless system (1 state!)
  - Yes → Internal states
- **State variables:**
  - Data that the systems stores/processes/uses
  - Keep their values between event occurrences
  - Special state variable: *control location*
- **State:**
  - The current values of the state variables of the system at a given moment (→ *state vector*)

State transitions

- **Transition:**
  - An event can trigger a change of system state
  - E.g. the value of a variable is changed, or from this point, the system will react differently to events
- **Action:**
  - Behavior executed due to occurrence of events
  - Can access: state variables, parameters of the event
- Actions may belong to transitions
  - Transition = (source state, *event*, *action*, target state)

Precondition: source state, *event* — Postcondition: *action*, target state

italic = optional

State transitions

- **Transition:**
  - An event can trigger a change of system state
  - E.g. the value of a variable is changed, or from this point, the system will react differently to events
- **Action:**
  - Behavior executed du
  - Can access: state vari
- Actions may belong to tran   ns
  - Transition = (source state, *event*, *action*, target state)

Precondition: (source state, *event*) — Postcondition: (*action*, target state)

italic = optional

Transitions without an event: *Implicit / spontaneous* transitions, not triggered by external events

UML/SYSML STATE MACHINE

States (hierarchical refinement, pseudostates)

Transitions (timers, complex transitions)

The UML State Machine

- **UML State Machine Diagram (Statechart):**
  - For the modeling of hierarchical and concurrent systems
  - For the description of the behavior of a *UML class* or *SysML block*
    - Attributes of the object or component may be (state) variables in the state machine
- Extensions compared to simple state machines:
  - Hierarchical states (state refinement)
  - Concurrent behavior (parallel threads)
  - Memory (stored state configurations)

Terminology

- *Concrete state:*
  - The current state vector (i.e. values of state variables)
  - Like defined so far
  - Can be infinitely many (e.g. when modeling *time*)
- *Abstract state:*
  - ≈Set of concrete states
  - ≈Predicates over concrete states
  - UML State Machine → „control location”
    - Along a distinguished state variable (state configuration, see later)
    - Other variables are not part of the state signature

State (UML State Machine)

Hierarchical state refinement:

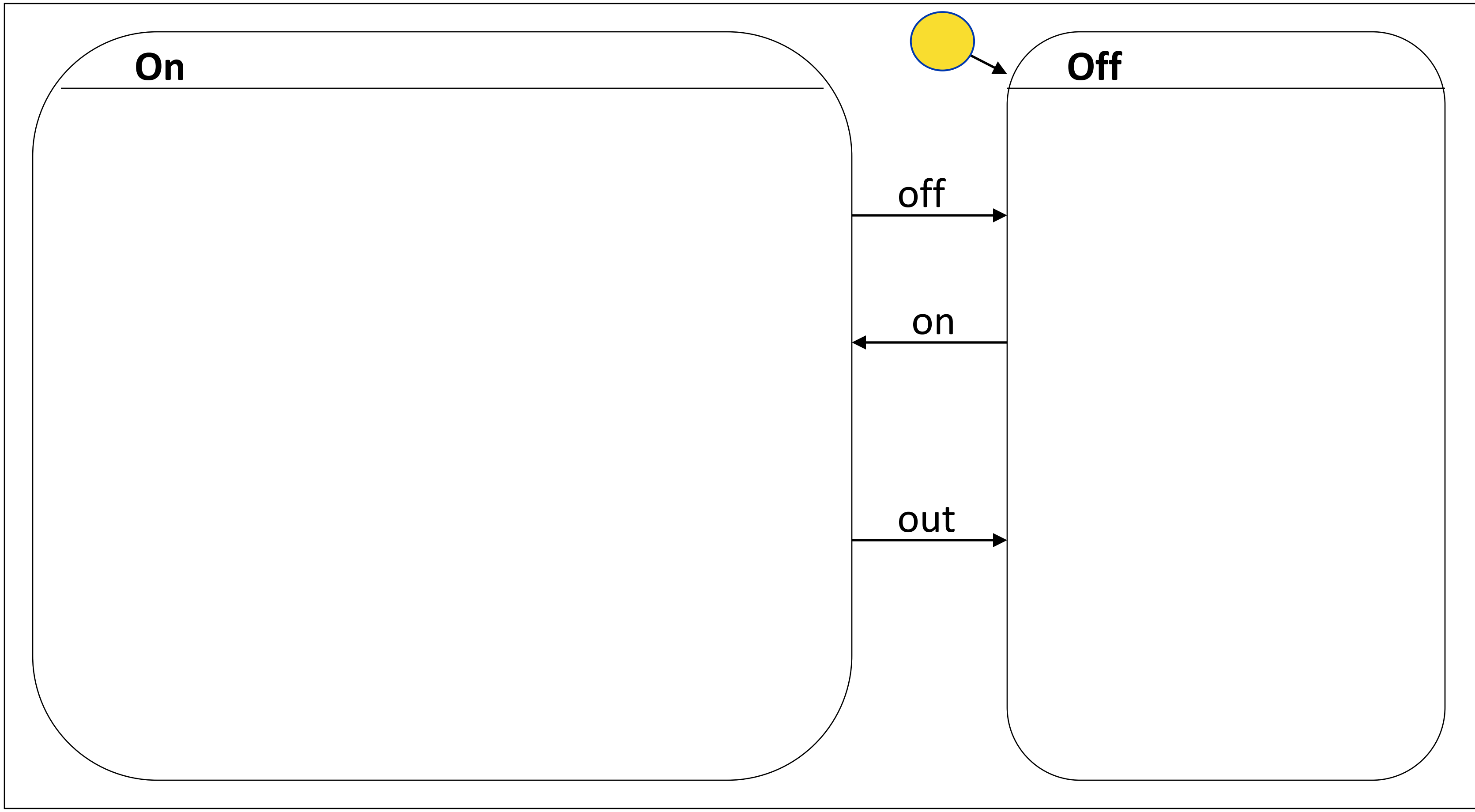
- **Simple state**
- **OR-refinement** (hierarchical refinement):
  - State is replaced by complete state machine
  - Refined state active $\leftrightarrow$ Exactly 1 child state active
- **AND-refinement** (parallel refinement):
  - State is replaced by parallel state machines (*parallel regions*)
  - Refined state active $\rightarrow$ Exactly 1 child state active *in each parallel region*

Complex state

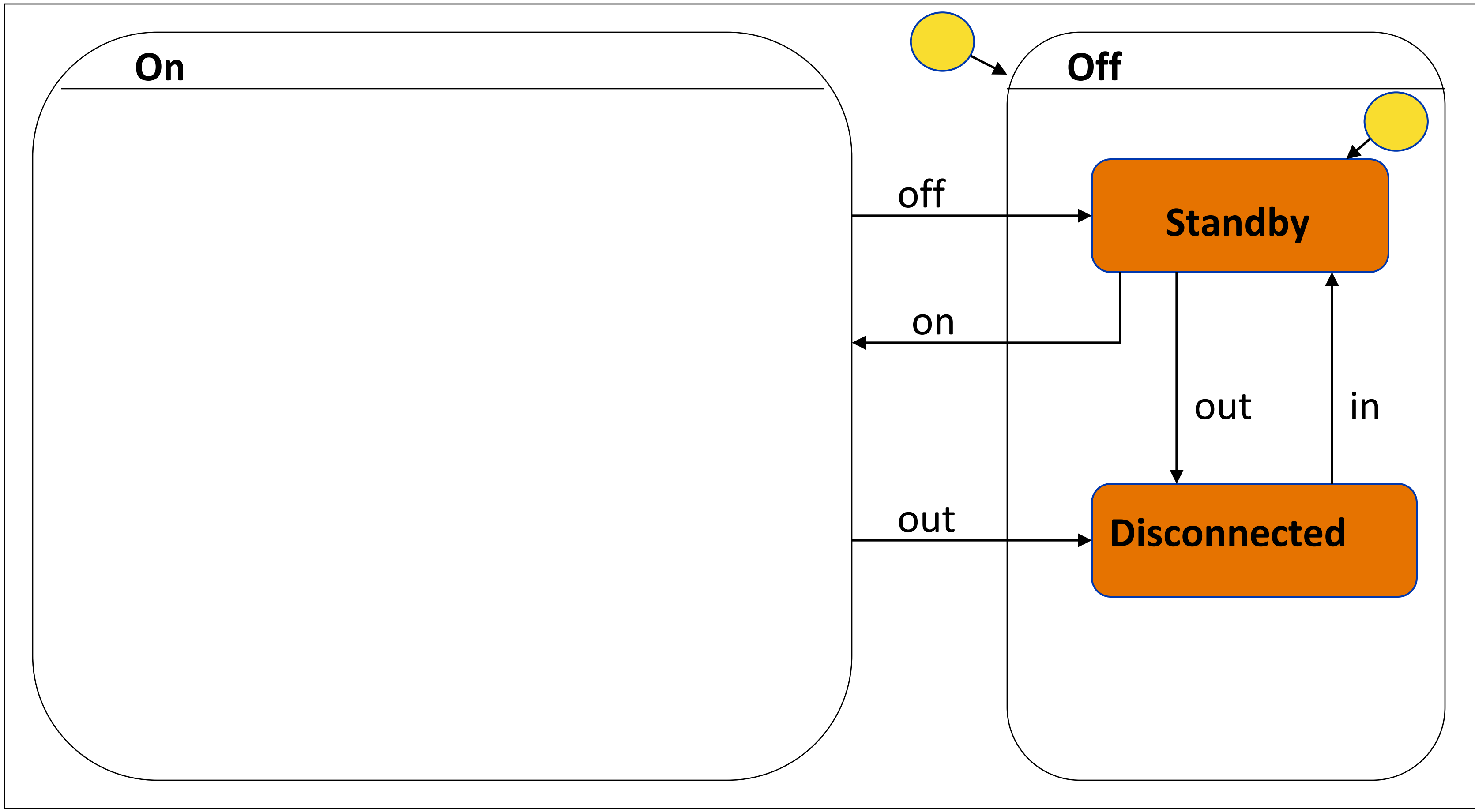
MŰEGYETEM 1782

ftsrg

State refinement (example)

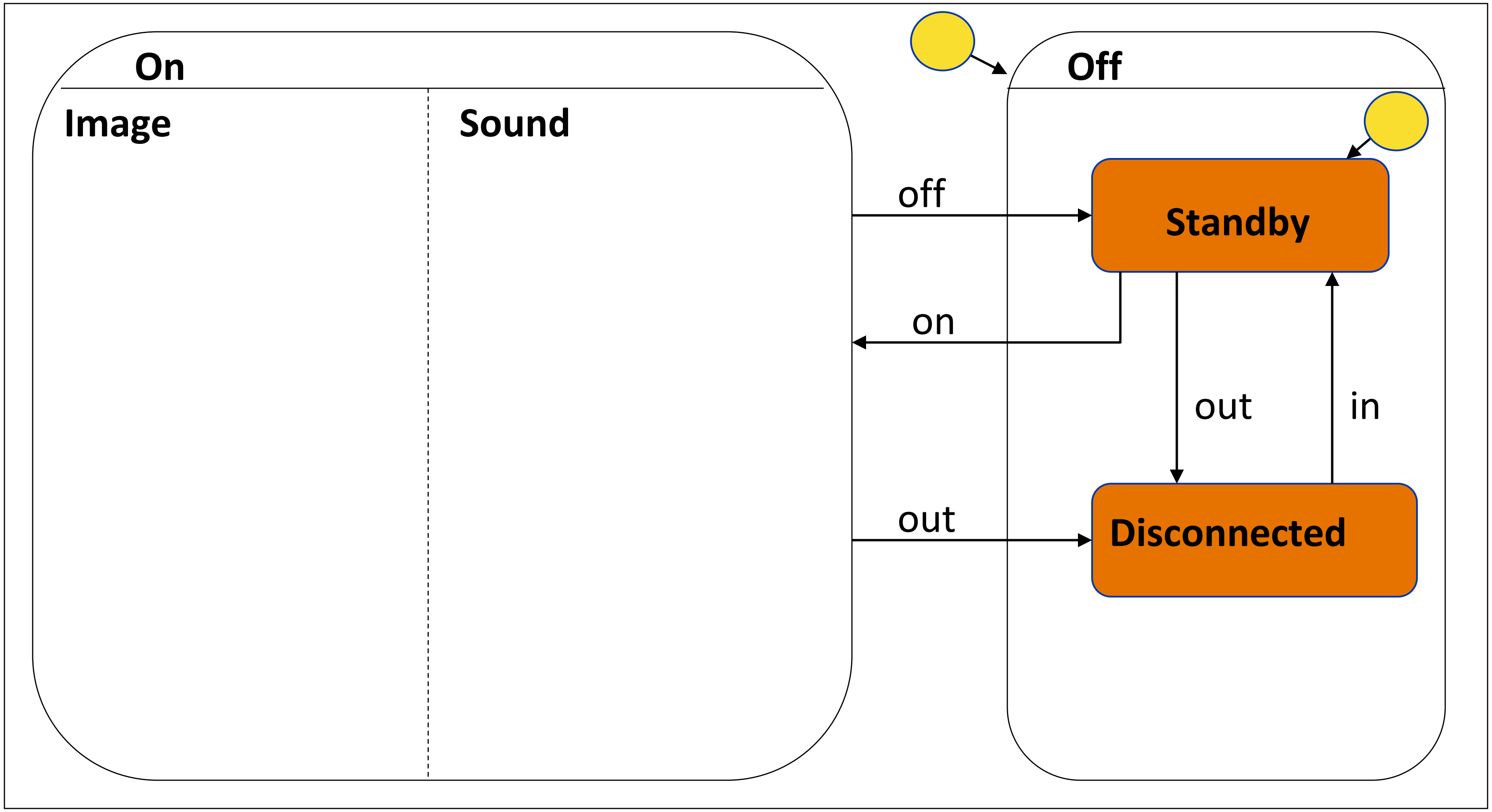


State refinement (example)



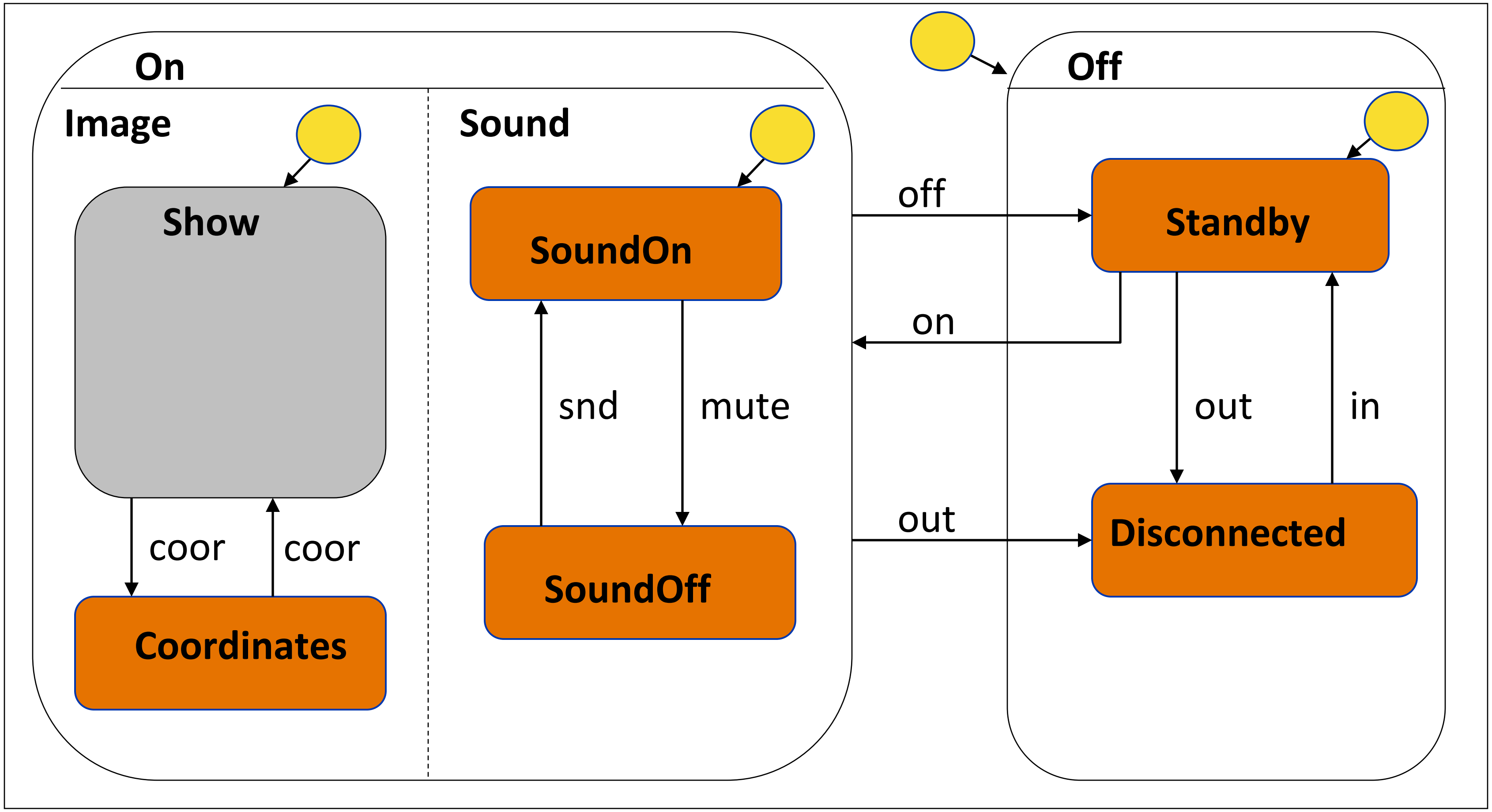
OR-refinement

State refinement (example)



AND-refinement **OR-refinement**

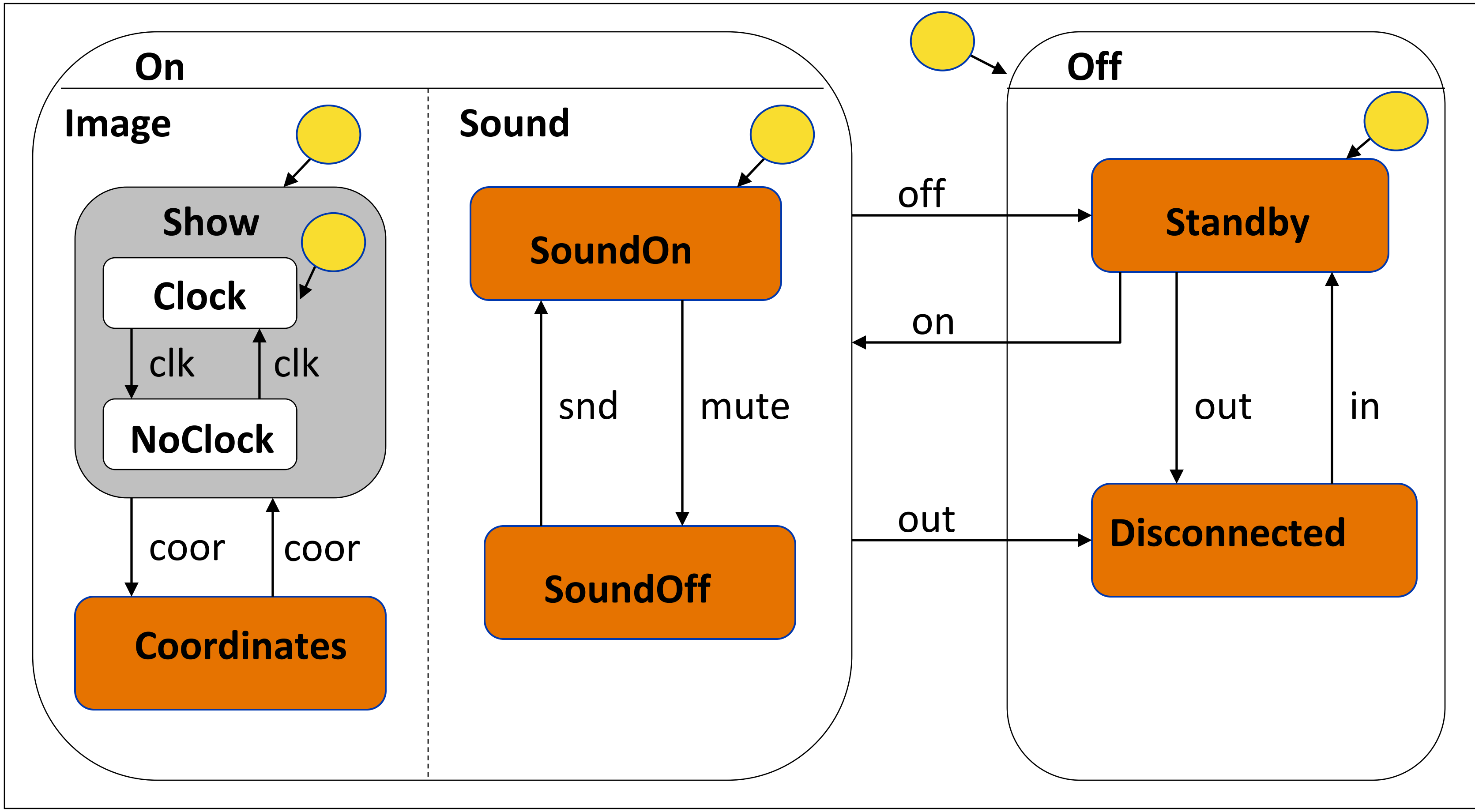
State refinement (example)



AND+OR-refinement **OR-refinement**

State refinement (example)

OR-refinement



AND+OR-refinement

OR-refinement

State configuration

- In a UML State Machine, there can be multiple active „states" (abstract states / control locations)
- A state configuration is the set of active states (state space = set of valid state configurations)
- Valid **state configuration:**
  - The top-level state machine has *exactly one* active state
  - In every active OR-refinement there is *exactly one* active state
  - In every region of an active AND-refinement there is *exactly one* active

State (UML State Machine)

<u>Actions related to states:</u>

- ***Entry/Exit action***:
  - Executed when entering/exiting a state
- ***Do action:***
  - Starts after the *Entry action* has finished
  - Runs in parallel with *Do actions* of other active states
  - May produce a *completion* event when finished
  - Is terminated when the state is left
  - Example: waiting for connections, blinking light, etc.
  - *Note: mixture of flow- and state-based modeling!*

Transitions (UML State Machine)

- **Transition:**
  - Modeling of state changes
  - Can be triggered by events or completion
  - Can depend on current values of variables
  - An action may be executed when the transition *fires*

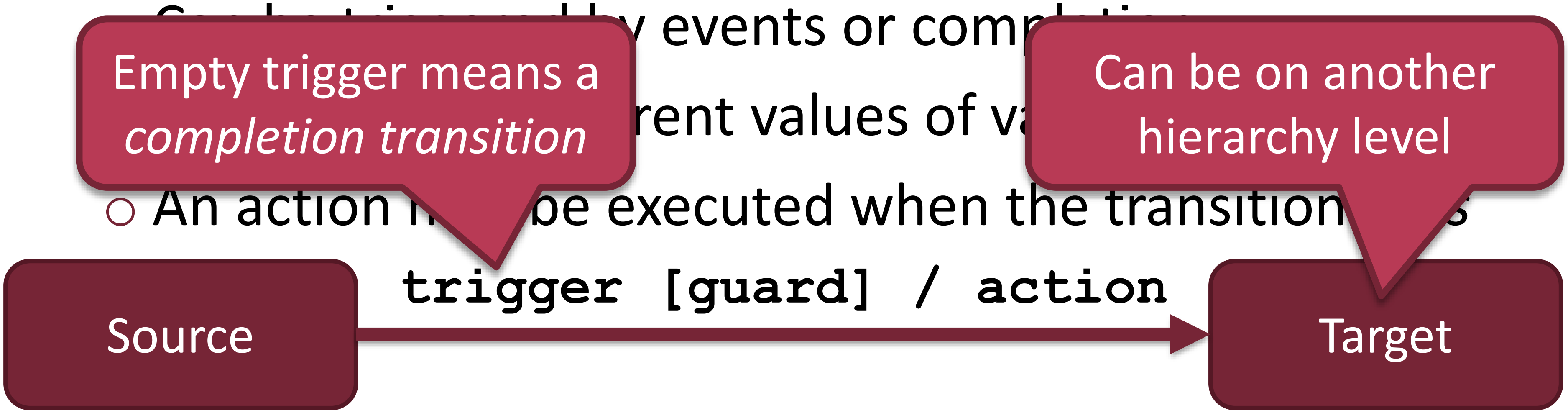
Transitions (UML State Machine)

- **Transition:**
  - Modeling of state changes
  - Can be triggered by events or completion
  - Can depend on current values of variables
  - An action may be executed when the transition *fires*

- Trigger: event that causes the reaction
- Guard: logical formula, must be true to fire
- Action: the action to execute

Transitions (UML State Machine)

- **Transition:**
  - Modeling of state changes
  - … y events or comp …
  - … rent values of va …
  - An action … be executed when the transition … s

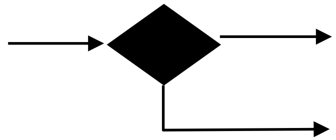


- Trigger: event that causes the reaction
- Guard: logical formula, must be true to fire
- Action: the action to execute

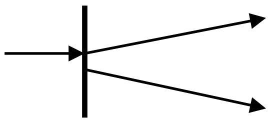
Transitions (UML State Machine)

<u>Complex transitions:</u>

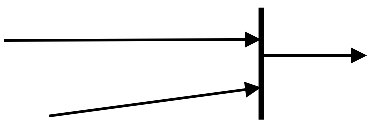
- **Condition:** Different reactions to an event based on certain conditions



- **Fork:** To denote target states in multiple parallel regions



- **Join:** To synchronize parallel regions and denote a common target state

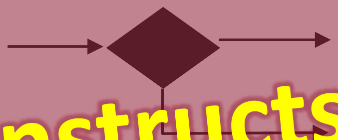


- **Internal transitions:** like a self-loop, but its firing does not leave and enter the source state
  - Notation: along with the attributes of the state

Transitions (UML State Machine)

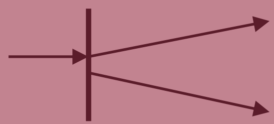
<u>Complex transitions:</u>

- **Condition:** Different reactions to an event based on certain conditions

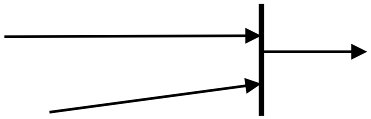


Often replaceable by other constructs

- **Fork:** To denote target states in multiple parallel regions



- **Join:** To synchronize parallel regions and denote a common target state



- **Internal transitions:** like a self-loop, but its firing does not leave and enter the source state
  - Notation: along with the attributes of the state

Events and Actions (UML)

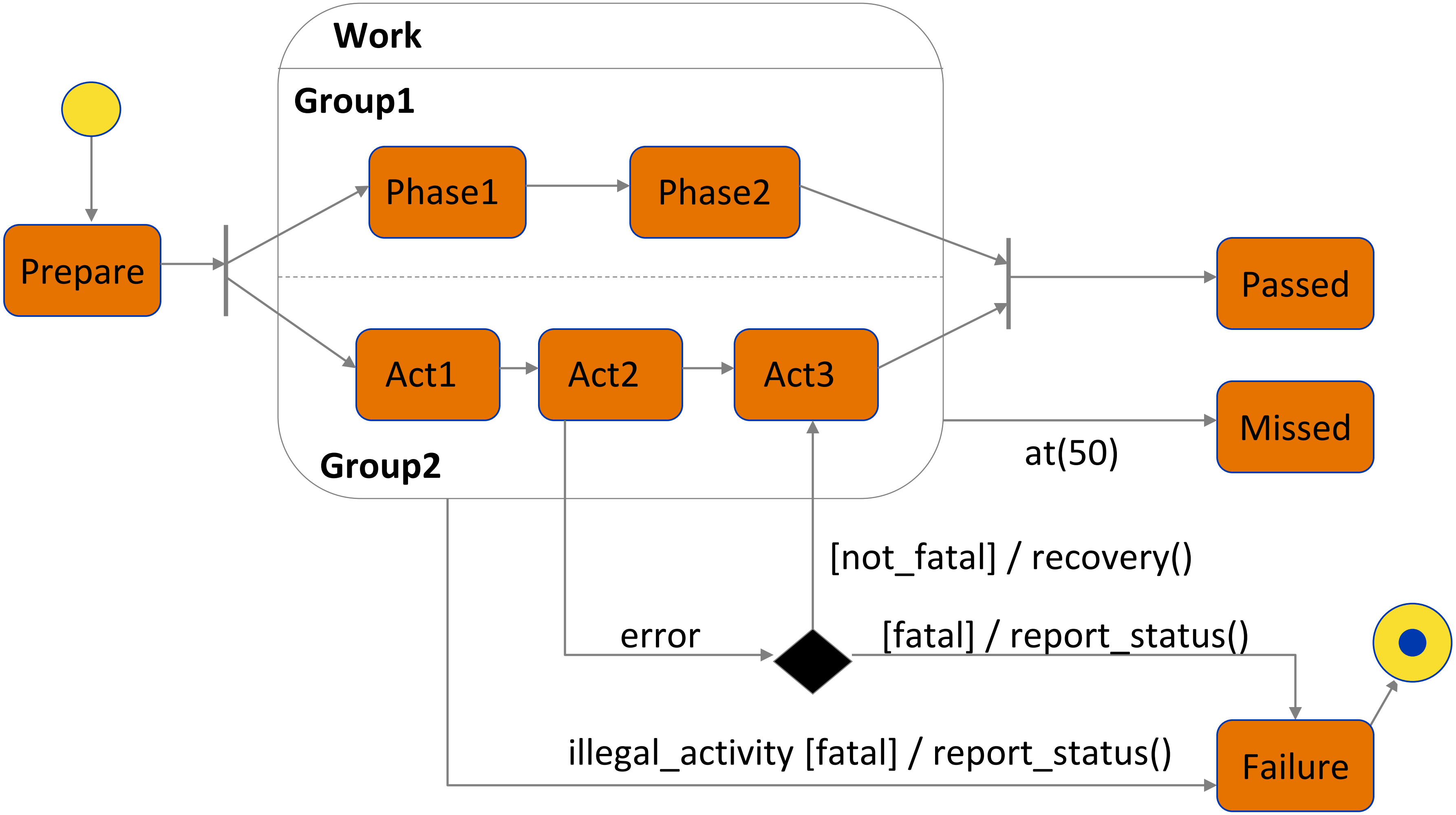
Events:

- Instances of the Event class (and its subclasses)
  - Asynchronous reception of a message
  - Invocation/completion of a method or behavior
  - Timer events
    - at(t): the value of the global clock is *t*
    - after(t): the source state has been active for time *t*

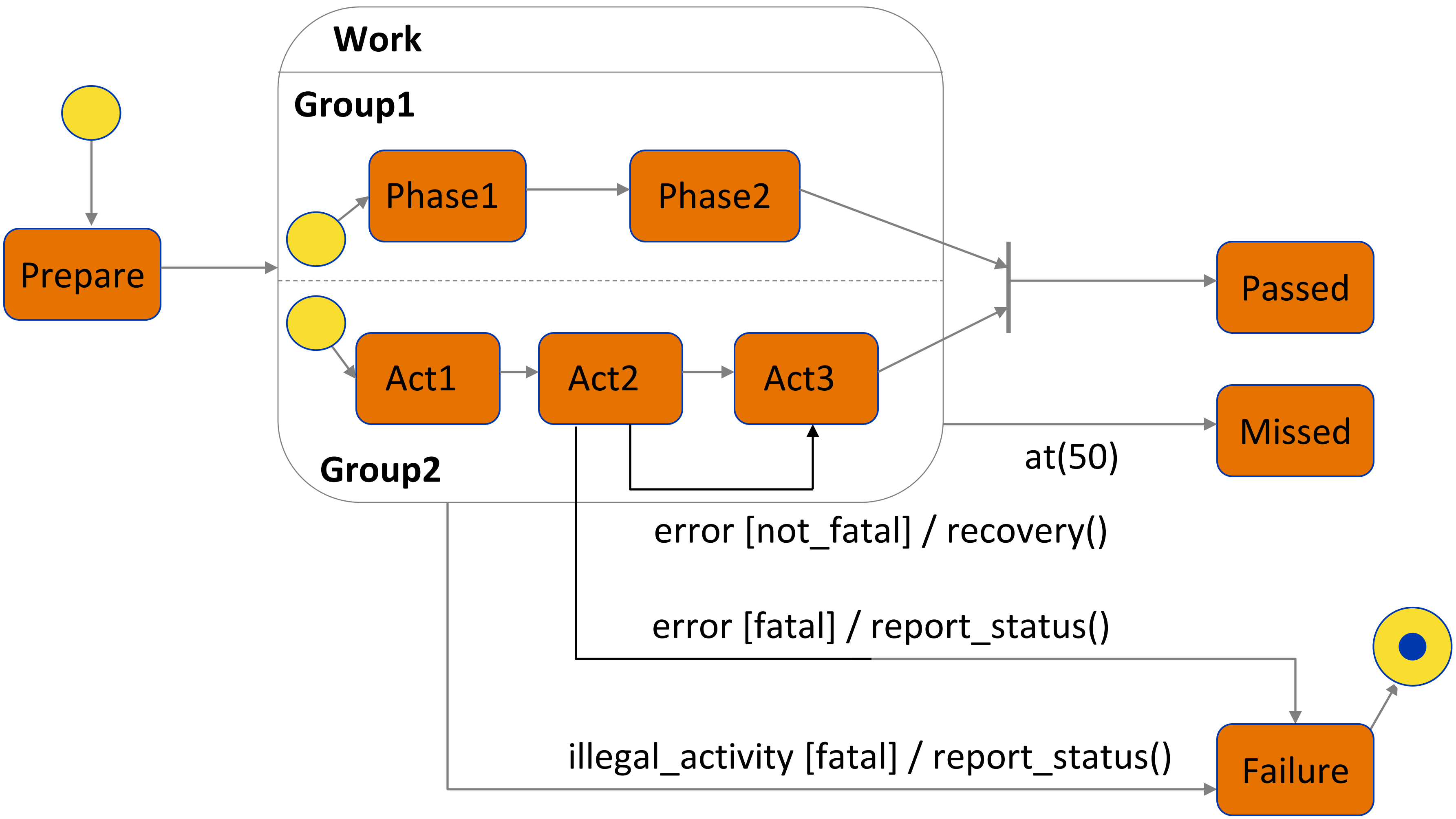
Actions:

- Instances of the Behavior class (and its subclasses)
  - Mostly Activities

Transitions (example)



Without Forks and Conditions (example)



Pseudostates

Initial state:

- Shall be one in every OR-refinement and every region of AND-refinements*
- Denotes the state to activate when entering a complex state

Final state:

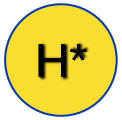
- The execution of the State Machine is terminated
  - May generate a completion event
- Rarely used („*reactive systems do not terminate*")

* It is considered bad practice, but omittable if transitions directly lead to child states of the complex state

Pseudostates

History State:

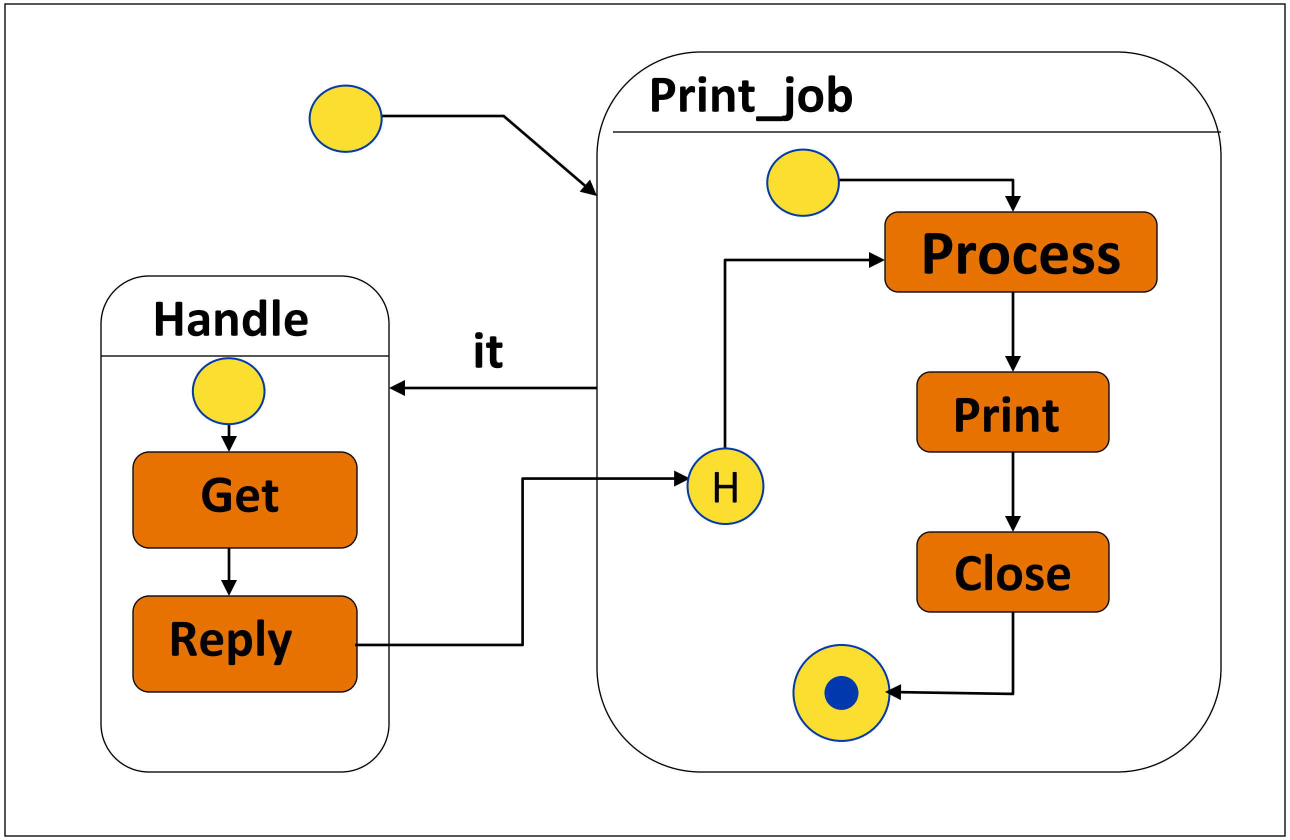
- Extension of the Initial State
- Denotes initial state when entering for the first time
- Stores the current state before exiting
- Restores last state on consecutive entries



Deep History State:

- Like the History State, but stores the last state configuration in the whole subhierarchy

Pseudostates (example)



Supplementary: Initial and History State

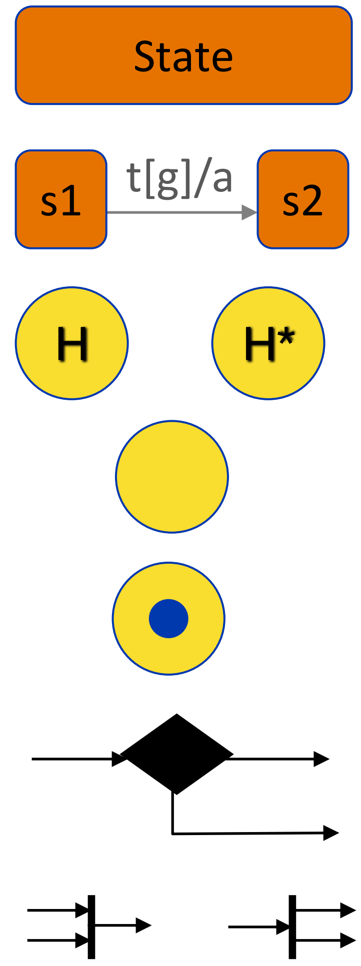
Combination of Initial State and History:

- If the transition leads to the complex state, the Initial State has priority
- The transition can lead directly to the History State to explicitly denote that the last state (configuration) is to be restored

- *Morals: be careful* ☺

Summary of syntax

- State
- Transition
- (Deep) History State
- Initial State
- Final State
- Condition
- Synchronization (Fork/Join)



SEMANTICS

Event queue

Scheduler

Priority

Conflict

Basics of semantics

1. Incoming events are put in an **event queue**
2. The **scheduler** takes a single event out of the event queue *in every step*
3. The event is processed by the State Machine
   - **„Run to completion”:** The event is completely processed until there is no more transition to fire
   - The State Machine can still be terminated externally
4. After the *complete* processing of the event, the scheduler starts the processing of the next event

The event queue **serializes** and **synchronizes**

Process of firing

1. Start from a **stable state configuration**
   - Nothing can be fired without an event occurrence

2. Collect **enabled transitions**:
   - Source state is active
     - Element of the current state configuration
   - The current event is the trigger of the transition
     - Completion transitions are triggered by a completion event
   - The guard of the transition evaluates to *true* over the current state and the current values of variables

Process of firing

3. Based on the **number of enabled transitions:**
   - *If only one:* Fire!
   - *If none:* **Nothing happens***
   - *If multiple:* Selection of transitions to fire

4. Detection of **conflicts**:
   - <u>Enabled</u> transitions *t1* and *t2* are in conflict *iff* the intersection of the sets of states <u>left</u> during firing is <u>not empty</u>

* Deferrable triggers may keep the event for later use

Process of firing

3. Based on the **number of enabled transitions:**
   - *If only one:* Fire!
   - *If none:* **Nothing happens***
   - *If multiple:* Selection of transitions to fire

→ The trigger of both transitions is the current event

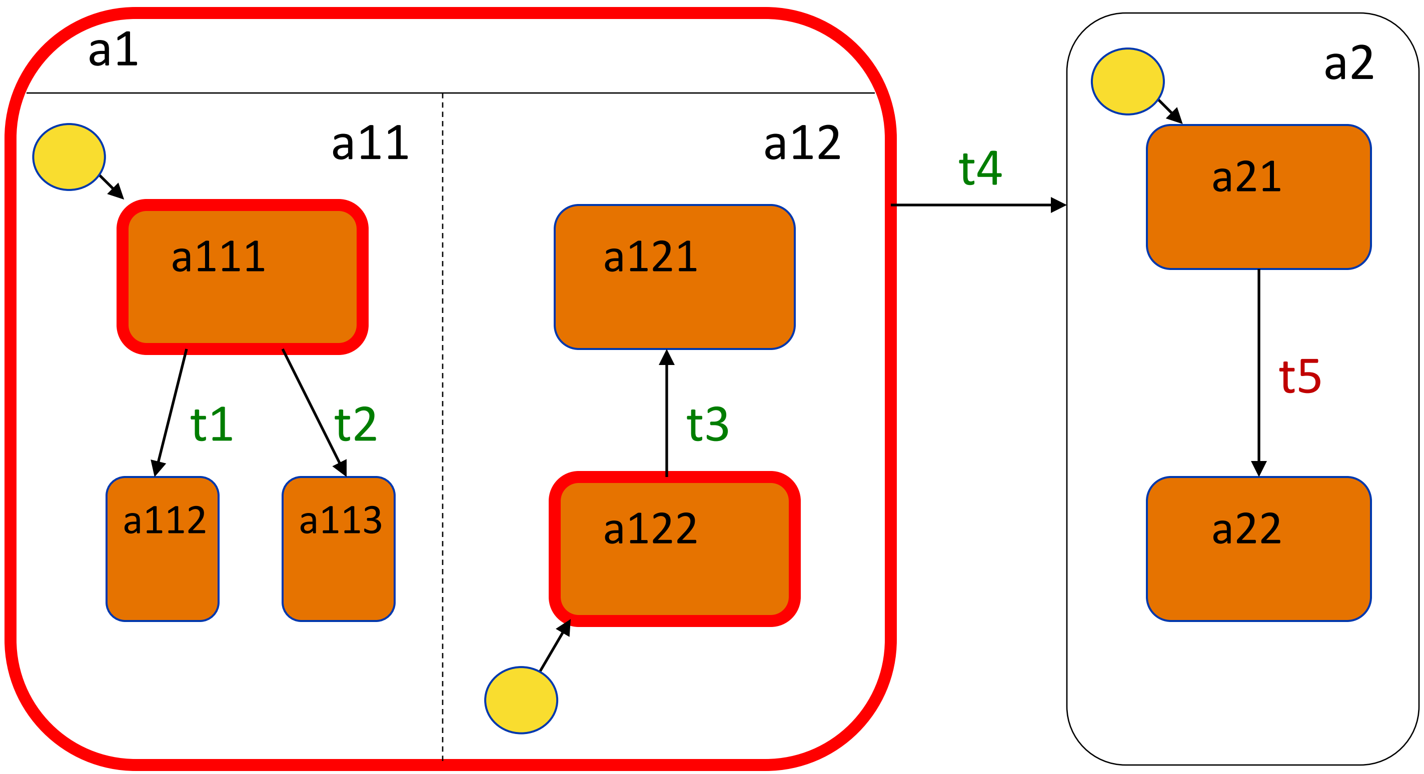
4. Detection
   - Enabled transitions *t1* and *t2* are in coflict *iff* the intersection of the sets of states left during firing is not empty

~{States of source config.} \ {States of target config.}

later use

Conflicts (example)

Every transition is triggered by the same event **e**: which sould fire?



Enabled transitions: t1, t2, t3, t4

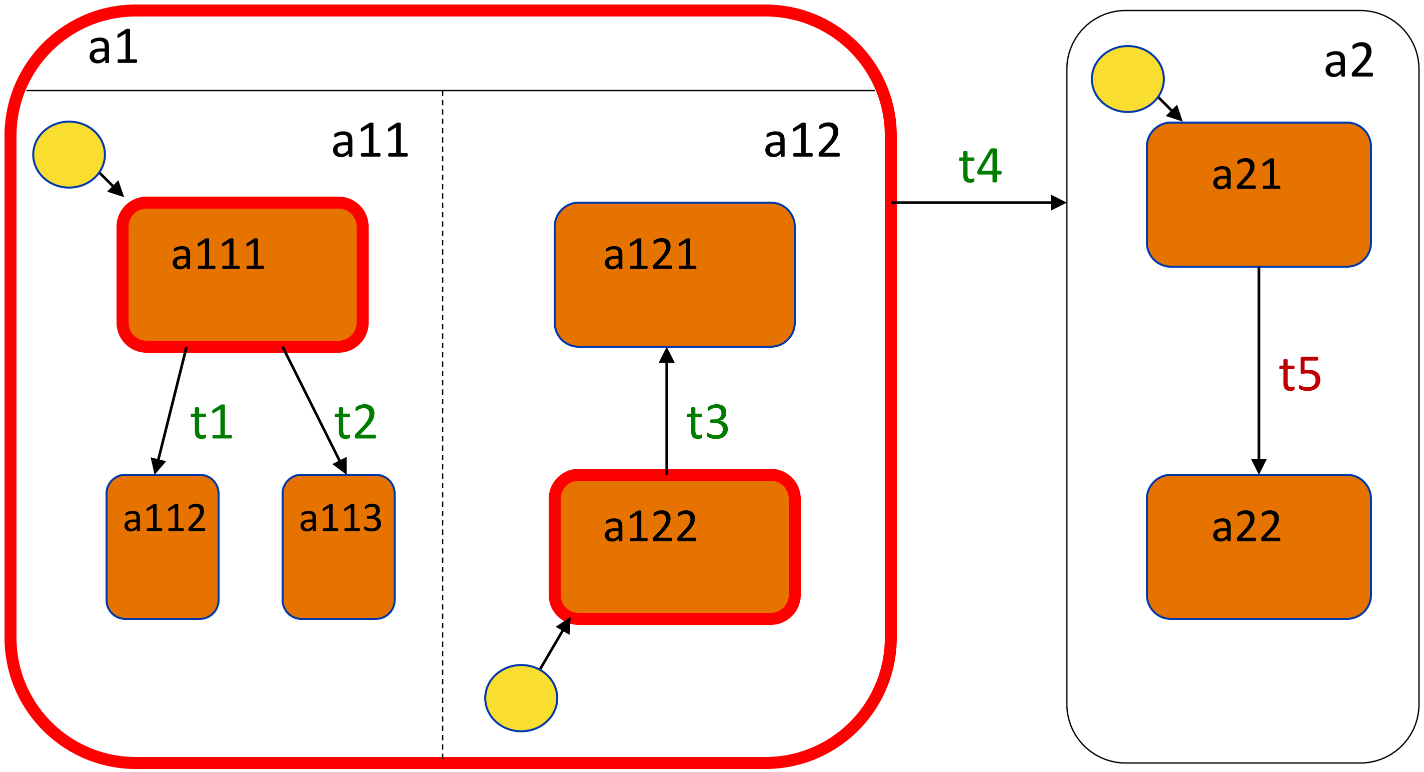
Cannot fire together (conflicting): {t1,t2}; {t1,t4}; {t2,t4}; {t3,t4}

Process of firing

5. **Conflict resolution:**
   - *Priority:* defined for a pair of transitions
     - Def: $t1 > t2 \Leftrightarrow$ source state of *t1* is **transitive child** of *t2*
     - *t1* is lower in the hierarchy, it is more „specialized”
     - ≈ inheritance and overriding in object oriented languages
   - *Fireable transitions*:
     - Highest priority among all enabled transitions
6. Selection of **transitions to fire**:
   - Every *conflict-free*, *maximal* (not further extendable) subset of *fireable transitions*
   - Selection from these: **non-deterministic**

Conflict resolution (example)

Every transition is triggered by the same event **e**: which sould fire?



Cannot fire together:	{t1,t2}; {t1,t4}; {t2,t4}; {t3,t4}
Priorities:	t1 > t4; t2 > t4; t3 > t4
Fireable:	**{t1,t3}; {t2,t3}**

Process of firing

7. Firing of the selected transitions
   - Order of individual firings is again **non-deterministic**
     - As usual for parallel behaviors…
   - Process of **firing a single transition**:
     1. Execution of *exit* actions of *left* (deactivated) source states (outwards)
     2. Execution of the action(s) belonging to the transition
     3. Execution of *entry* actions of *entered* (activated) target states (inwards)

Process of firing

7. Firing of the selected transitions
    - Order of individual firings is again **non-deterministic**
        - As usual for parallel behaviors…
    - Process of **firing a single transition**:
        1. Execution of *exit* actions of *left* (deactivated) source states (outwards

           ~{States in target conf.} \ {States in source conf.}
        2. Execution
        3. Execution of *entry* actions of *entered* (activated) target states (inwards)
8. If a completion event is generated, firing of completion transitions (steps 1-7. again with completion transitions)

Completion transition

- Transition without a trigger: **completion transition**
  - Triggered by a completion event
- A **completion event** is generated when
  - The *entry* and *do* actions have been finished
  - For complex states, additionally:
    - Each region have reached a *Final* pseudostate
- Completion events are processed **immediately**
  - Even if other events are in the queue
  - Multiple completion events in different orthogonal regions are processed in an *undefined order*

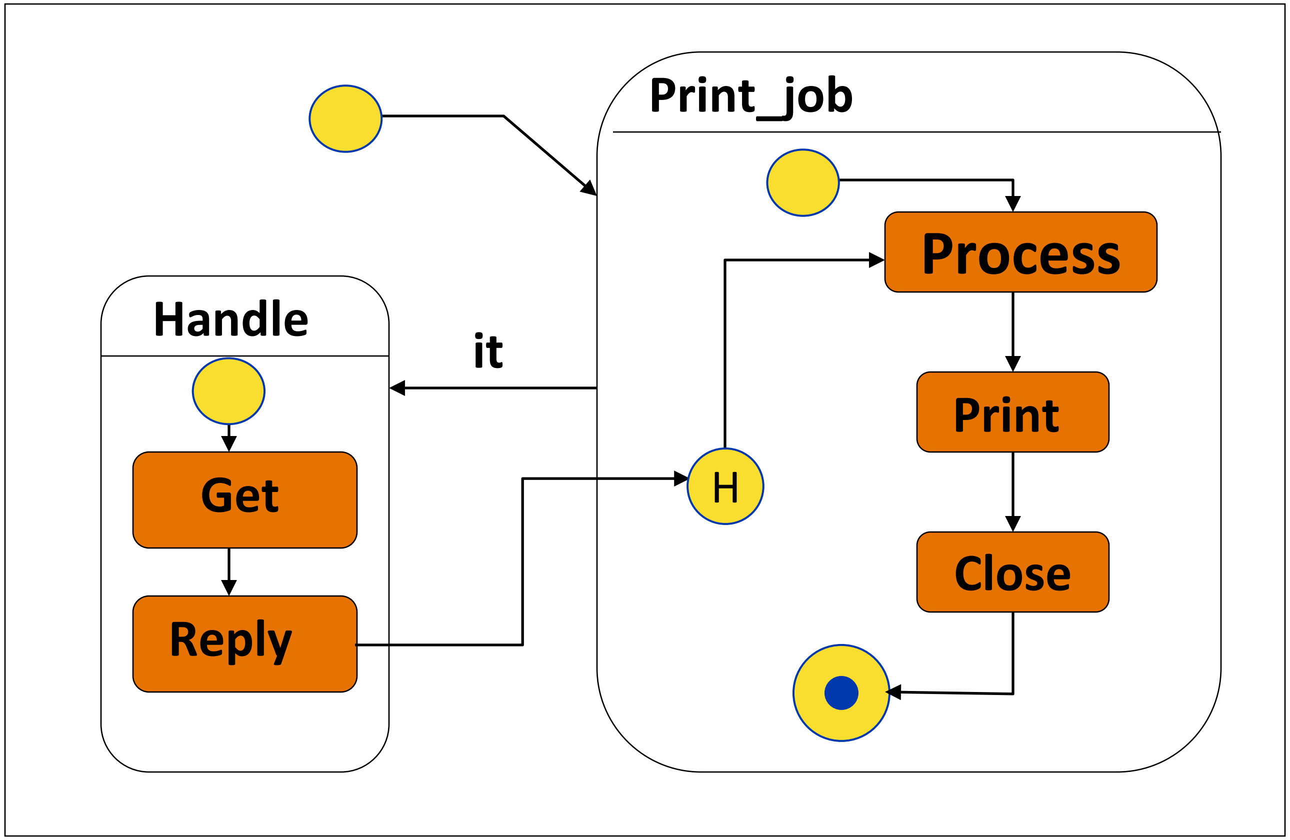
Completion transition

- Transition without a trigger: **completion transition**
  - Triggered by a completion event
- A **completion event** is generated when
  - The *entry* and *do* actions have been finished
  - For complex states, additionally:
    - Each region have reached a *Final* pseudostate
- Completion events are processed **immediately**

Facilitates the modeling of process-like behaviors in a state-based modeling language → *fuzzy semantics*

Do not use without a very good reason

Completion transitions (example)

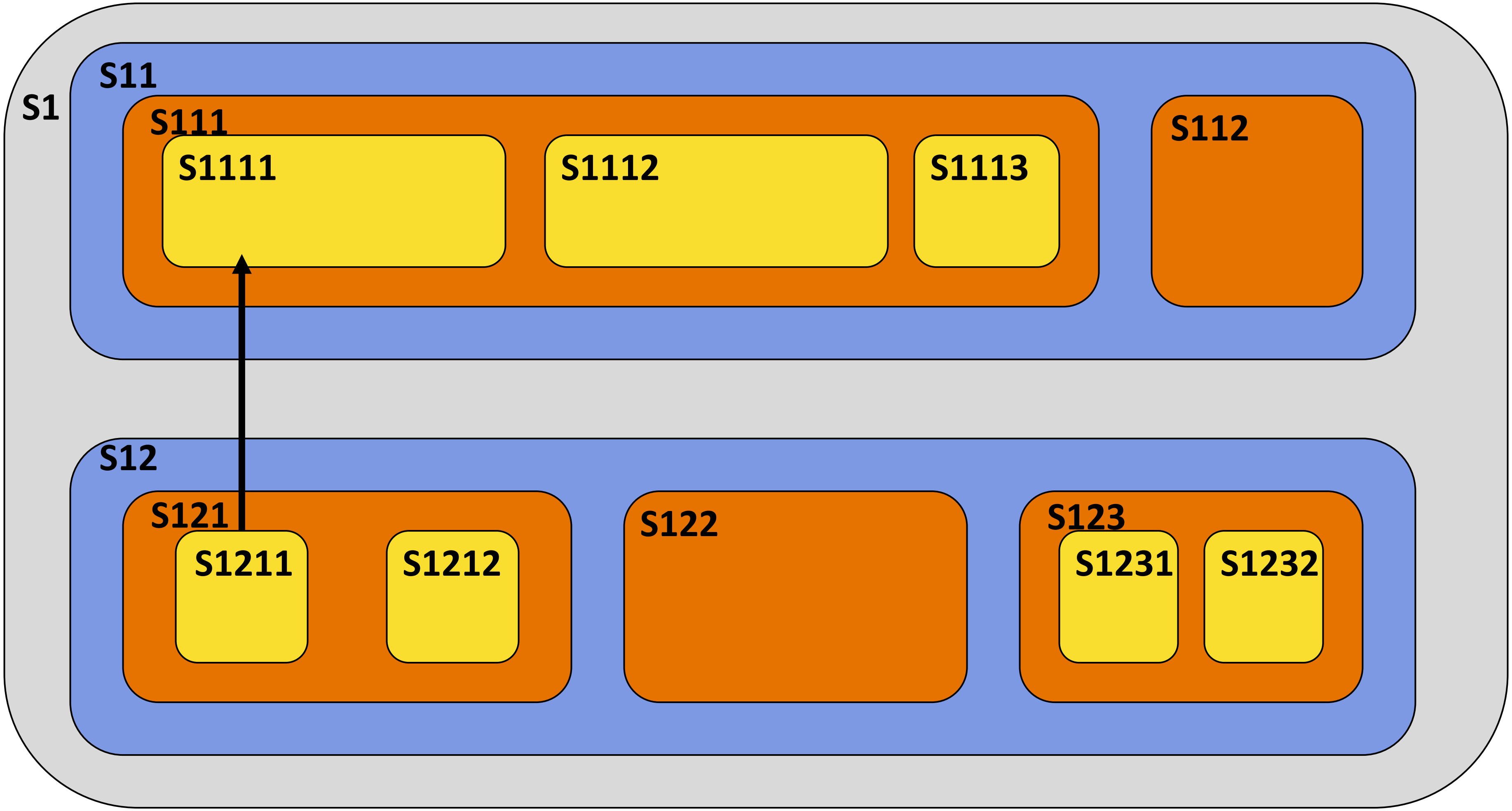


Identifying target state configuration

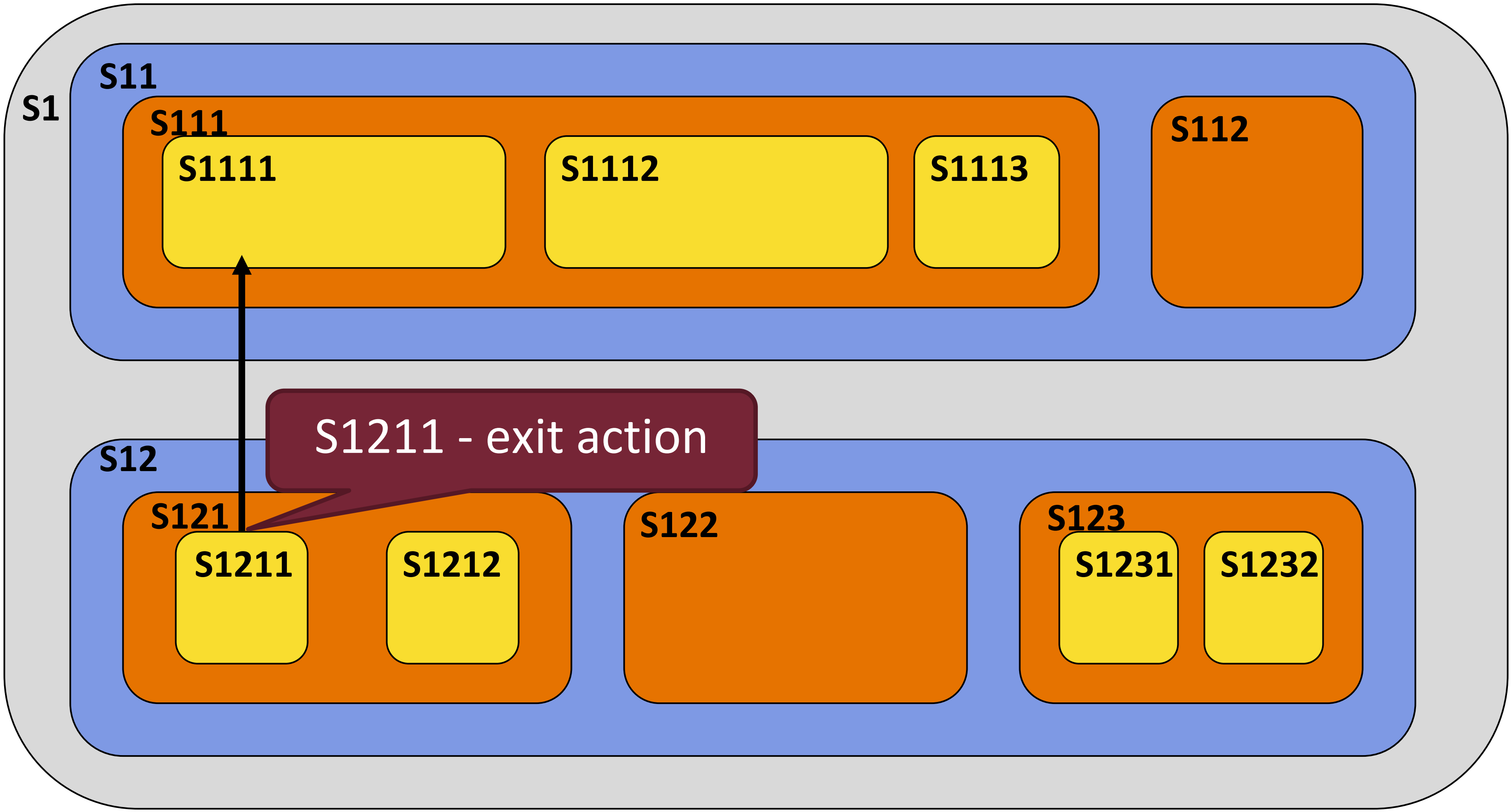
If the target of the transition is a…

1. **…simple state:** the new configuration is the state and all of its parents (transitively)
2. **…OR-refined state:** like case 1 and
   - In case of a History State: last state configuration
   - Otherwise: the state denoted by the Initial State
   - + States activated through the activation of any complex state
3. **…AND-refined state:** like case 1 and
   - For every parallel region, like case 2

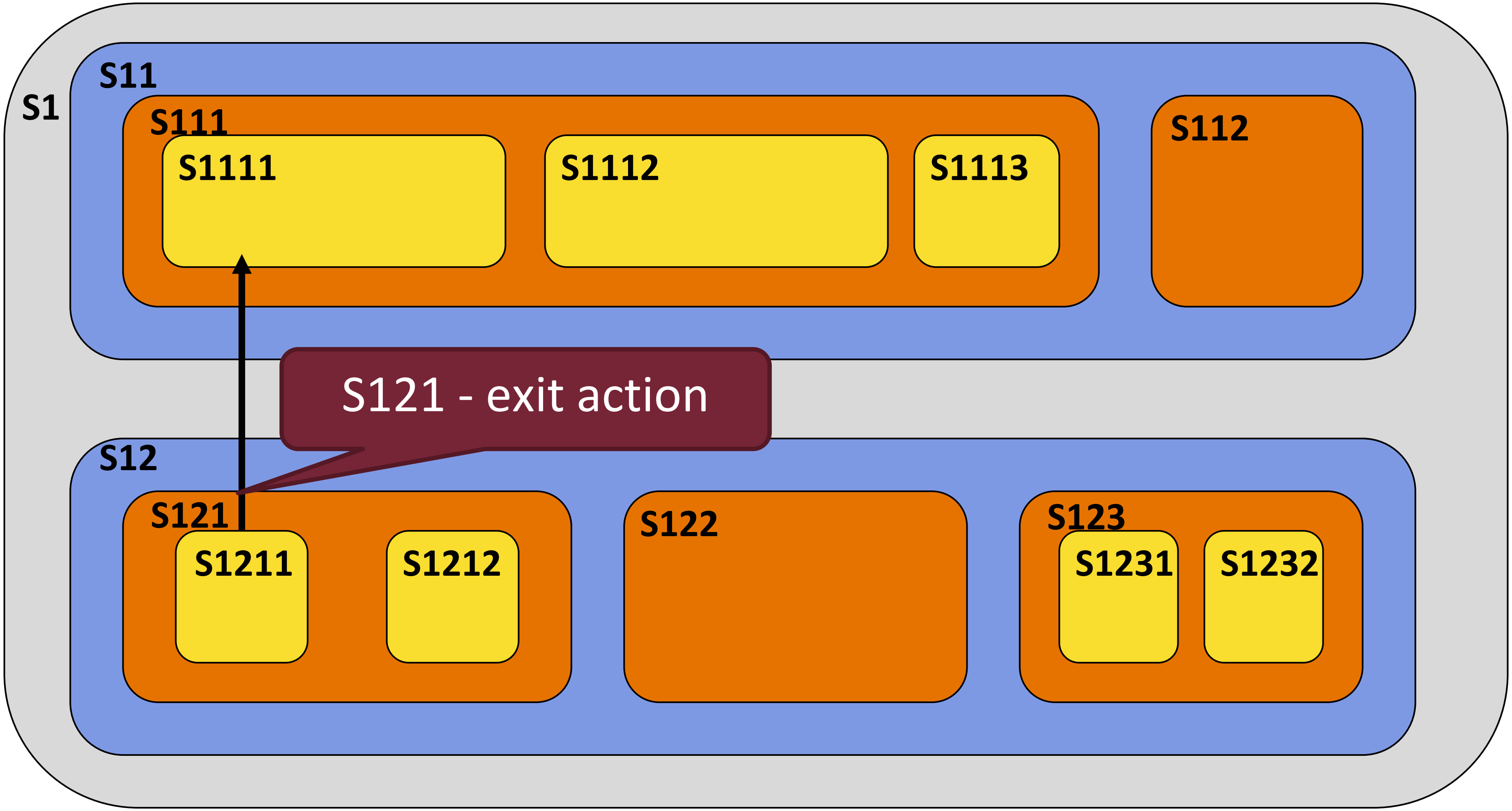
Changing of state configurations (example)



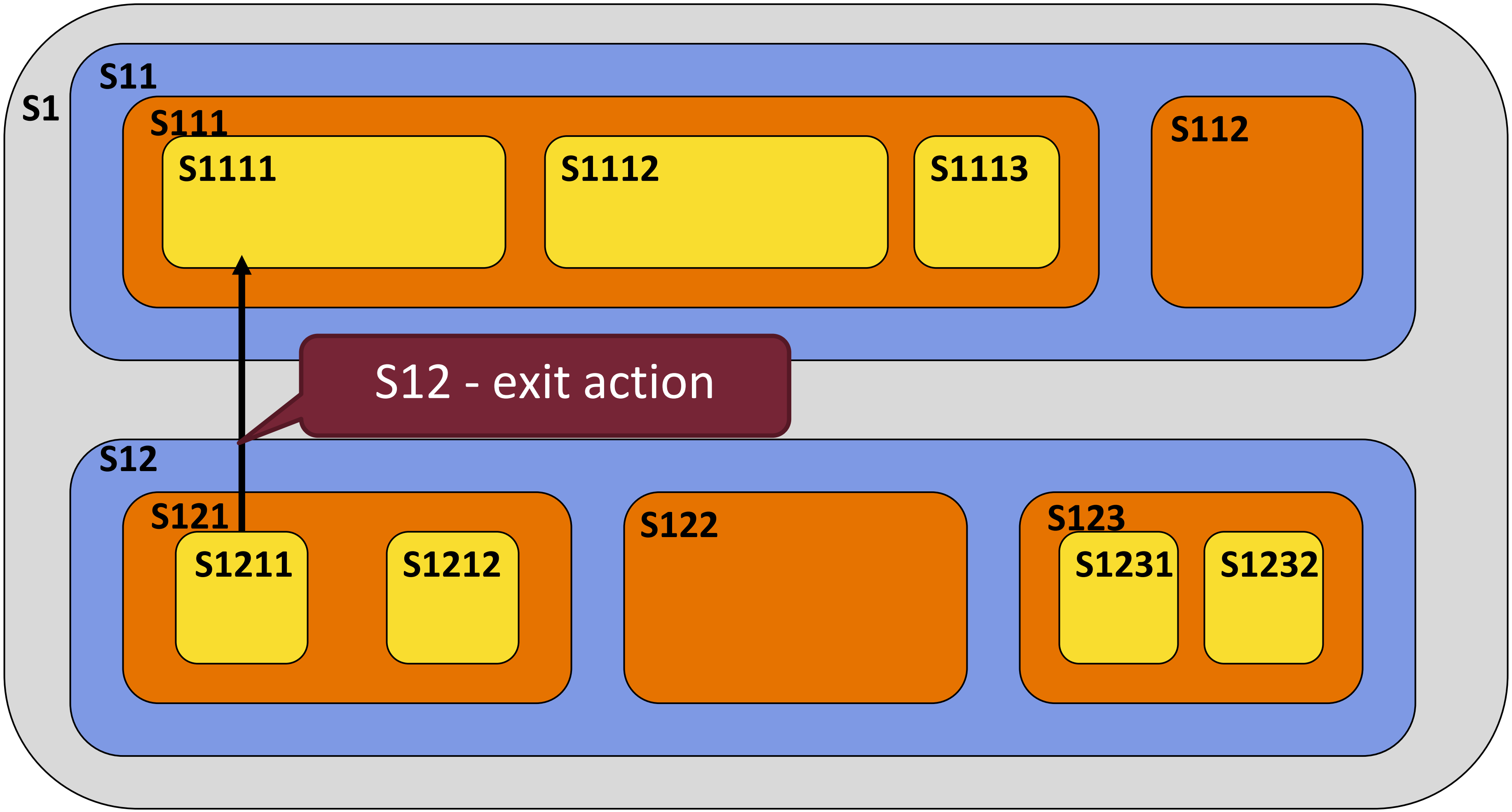
Changing of state configurations (example)



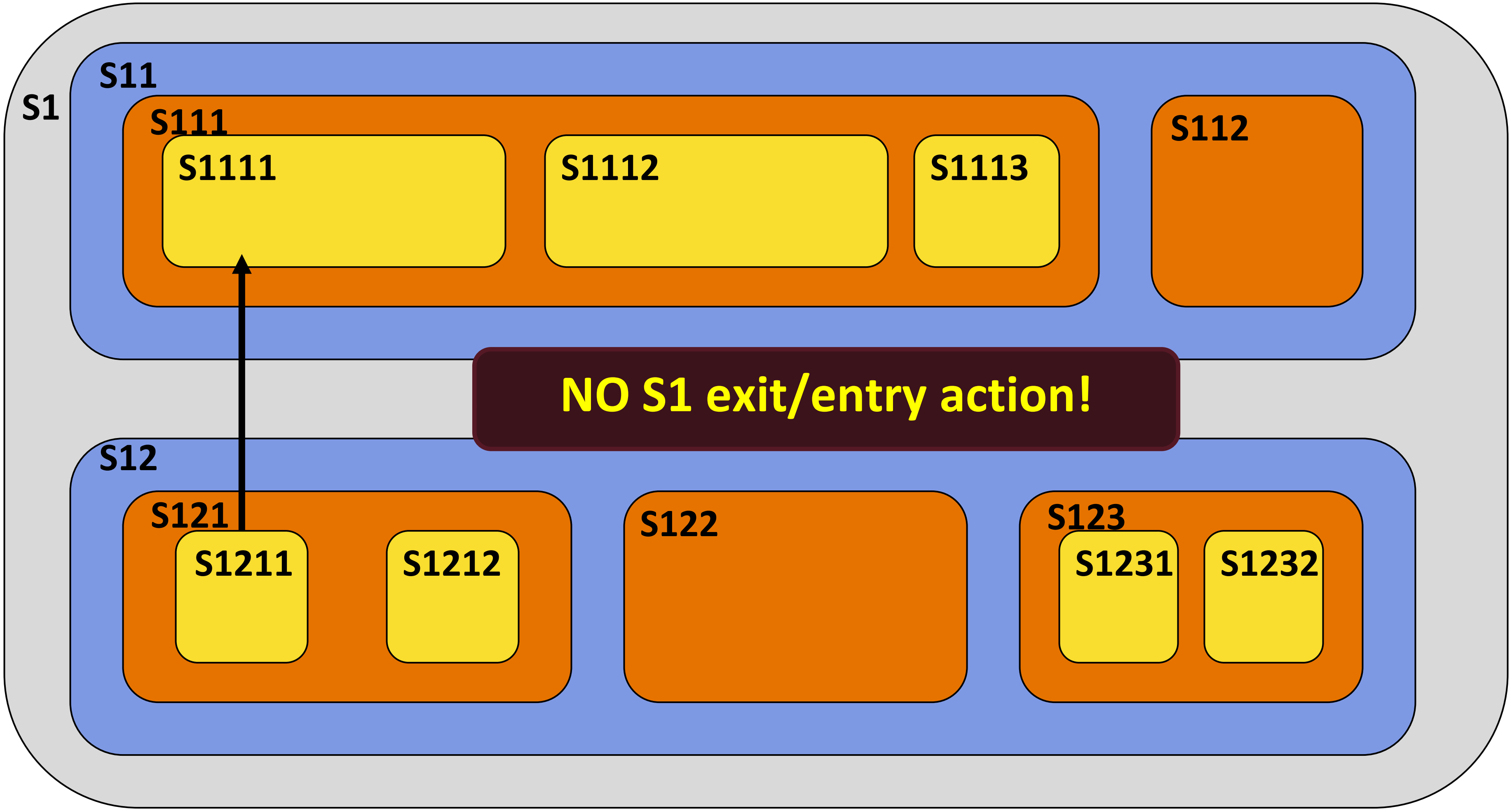
Changing of state configurations (example)



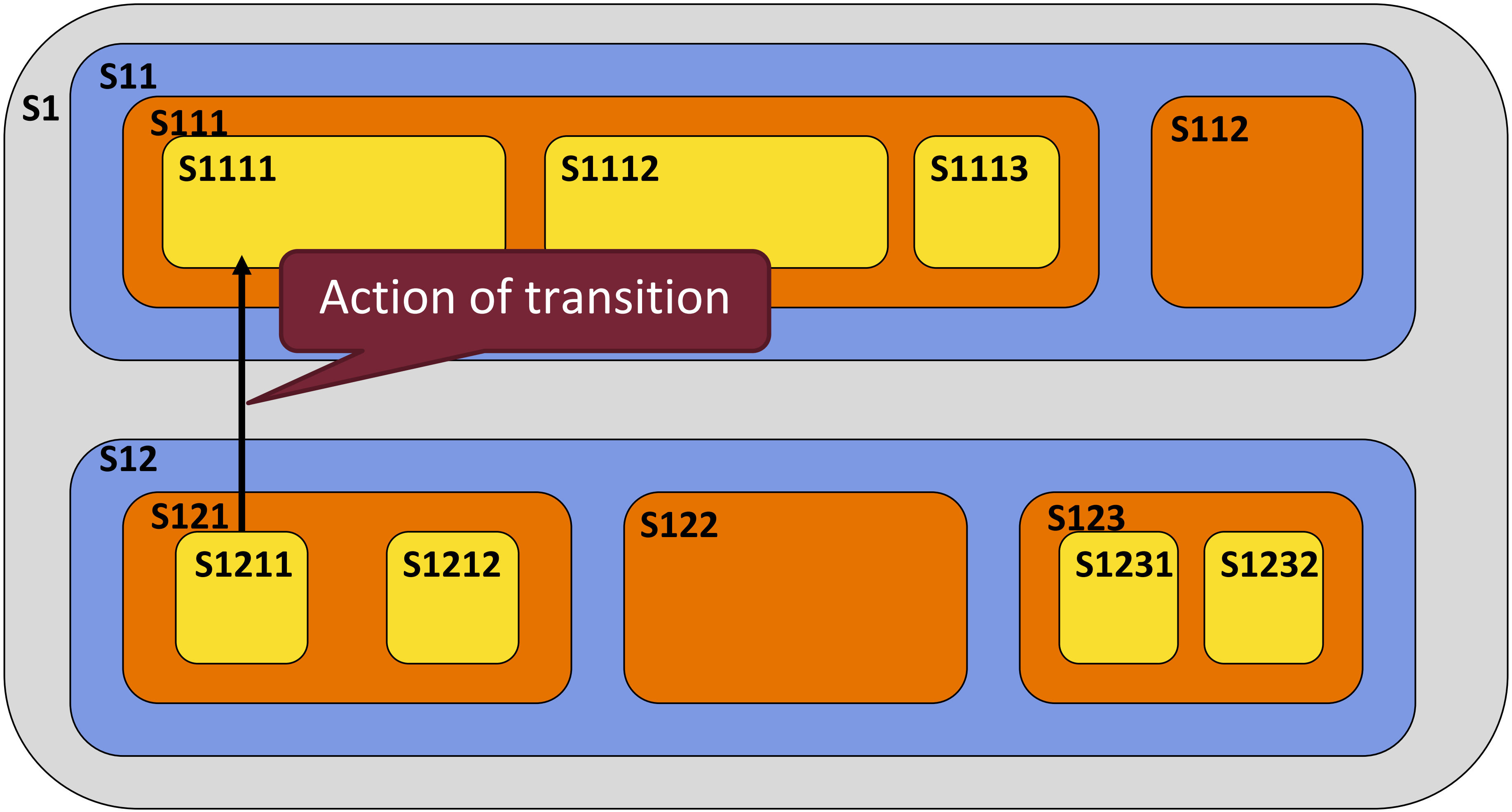
Changing of state configurations (example)



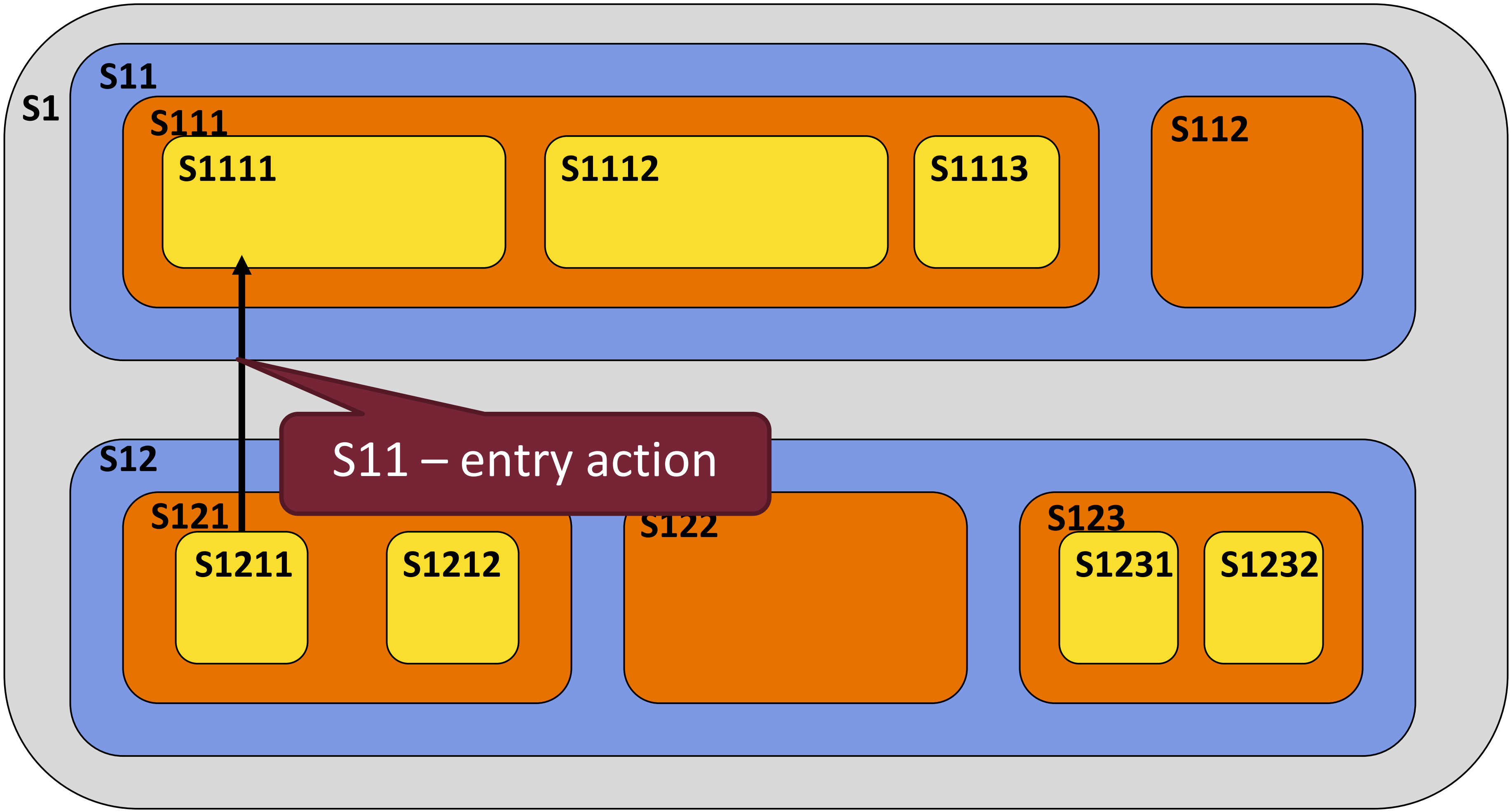
Changing of state configurations (example)



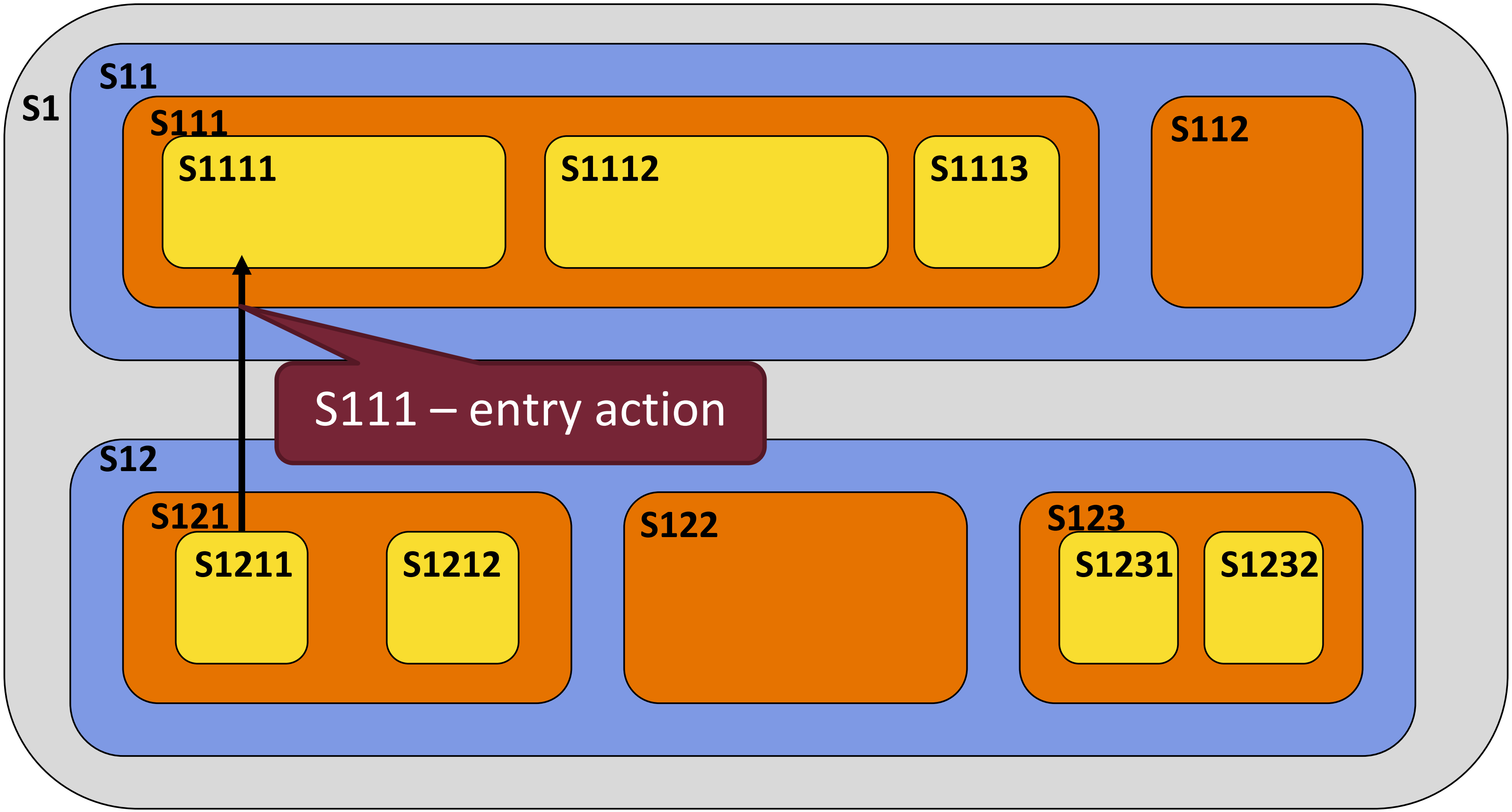
Changing of state configurations (example)



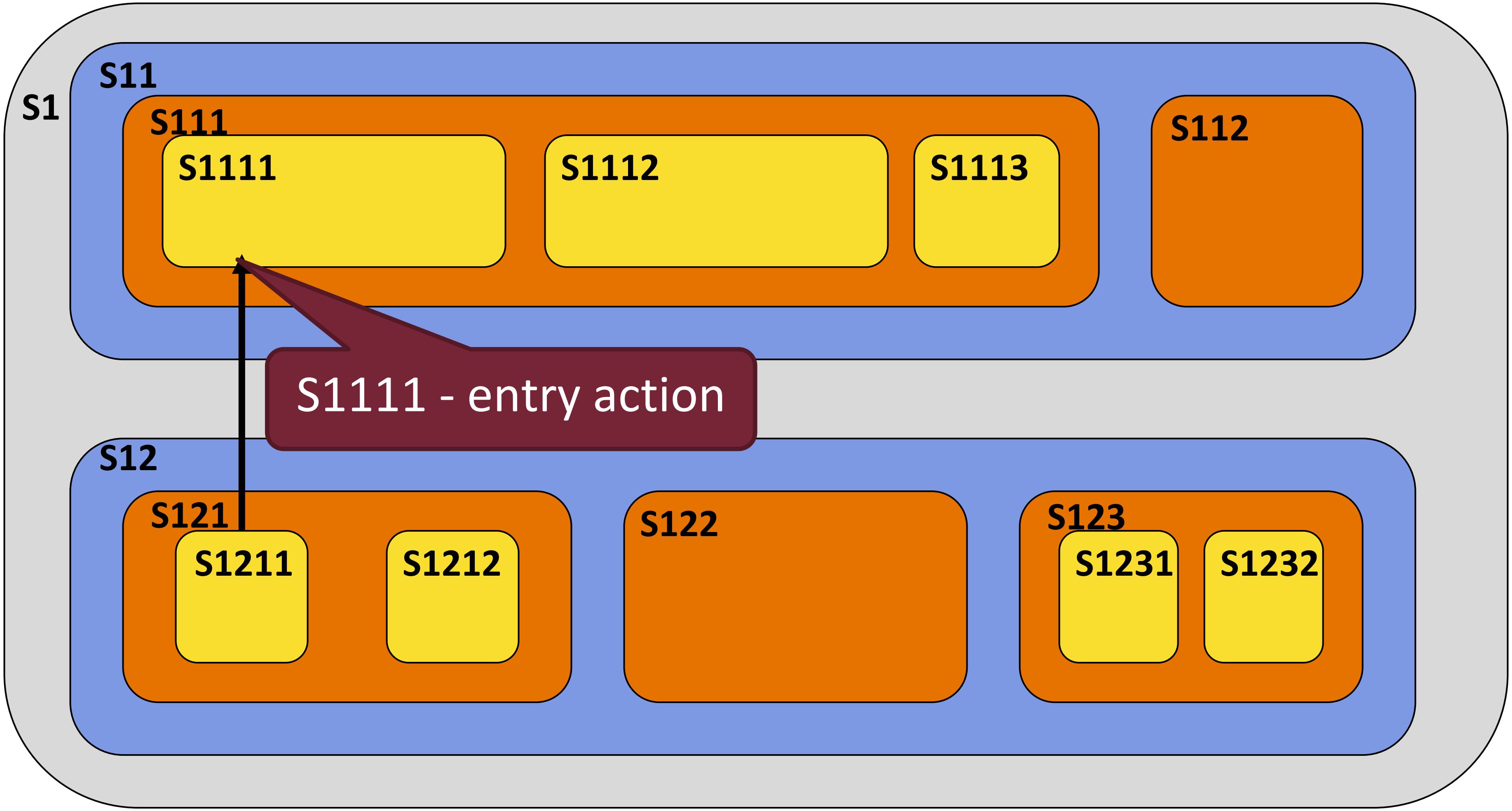
Changing of state configurations (example)



Changing of state configurations (example)



Changing of state configurations (example)

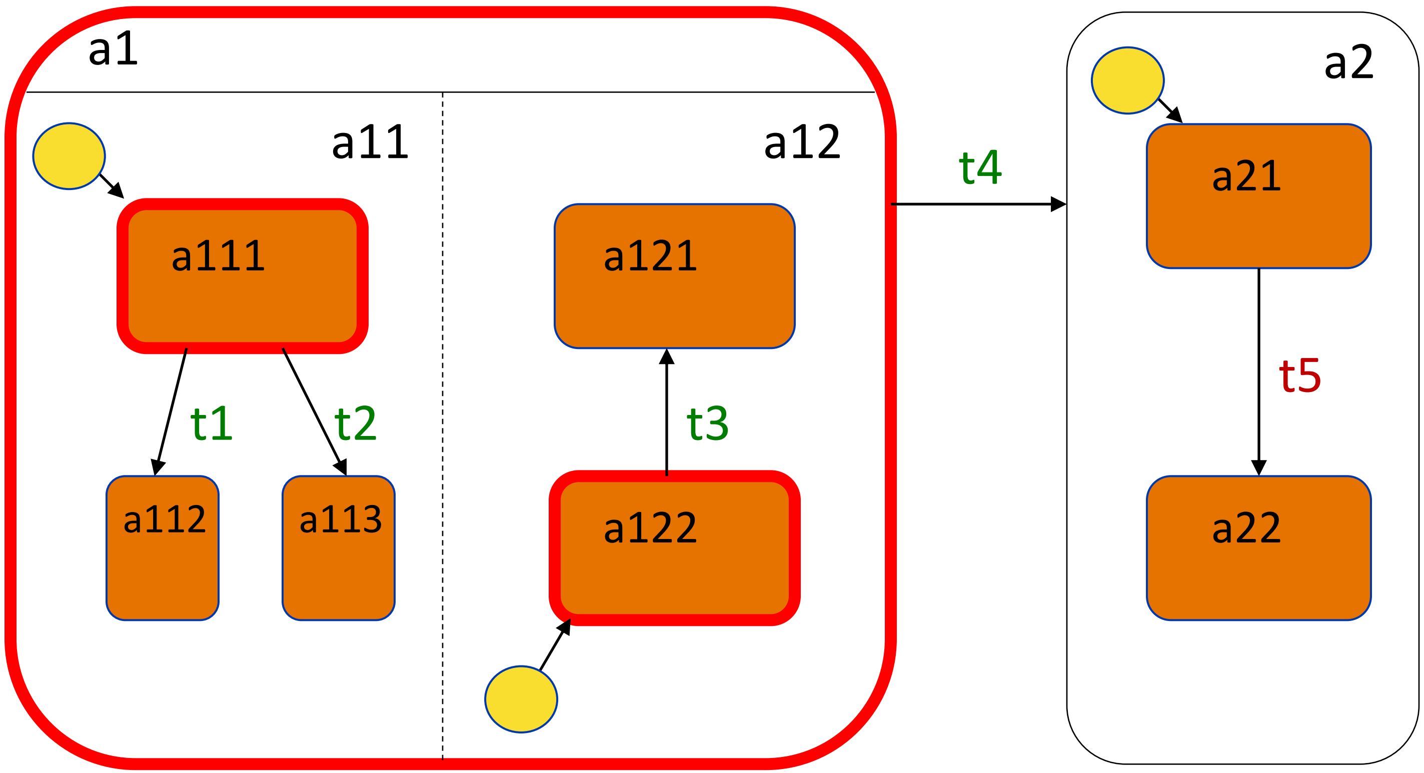


Summary of semantics

1. Start from a stable state configuration
2. Collection of enabled transitions
3. Decision based on number of enabled transitions
4. Detection of conflicts (cannot fire together)
5. Conflict resolution (priority, fireable transitions)
6. Selection of transitions to fire
7. Firing of selected transitions
    - exit actions (outwards), transition, entry actions (inwards)
8. Firing of completion transitions $\rightarrow$ stable config.

Summary of semantics (example)

Every transition is triggered by the same event **e**: which sould fire?



Enabled transitions:	t1, t2, t3, t4
Cannot fire together:	{t1,t2}; {t1,t4}; {t2,t4}; {t3,t4}
Priorities:	t1 > t4; t2 > t4; t3 > t4
Fireable:	**{t1,t3}; {t2,t3}**

MODELING WITH UML STATE MACHINES

Completeness, Unambiguity

Best practices

Modeling hardware interrupts

Complex example

Completeness and Unambiguity

- **Completeness:**
  - In every state configuration, for every guard evaluation, for every event: ≥ 1 behavior
    - Easier to check, but stricter:
      For every event and guard evaluation, there should be a transition *in every state or one of its parents*
- **Unambiguity:**
  - In every state configuration, for every guard evaluation, for every event: ≤ 1 behavior
    - Easier to check, equivalent:
      For every event and guard evaluation, there should be at most one transition *in every state*

Best practices

- Start from a simple state machine and use **state refinement**! Model level-by-level!
- Make sure there is an **initial state in every region**!
- Strive for **completeness**!
  - Add an **internal transition** for events that should not be handled
  - Complex states should define a default behavior for every relevant event
    - In the other direction: Use only those events in child states that are handled by the parent state as well
- Avoid transitions that **cross hierarchy levels**!

Best practices

- **Ambiguous** models should be built only for specification purposes or if the behavior is really random/not controllable (e.g. the model of the environment)!
- Use *entry/exit* actions for behaviors related to reaching or leaving states!
- **Avoid** using *do* actions!
- **Use a Final State** if the State Machine is meant to no longer process events!
- Use History States **lightly**!

RELATIONS TO OTHER DIAGRAMS

Class/Block Diagram

Activity Diagram

Interactions

Class/Block Diagram

- **Active Object** pattern: object has an own thread
  - Definition of behavior: UML State Machine
  - Events:
    - Method invocation/completion
    - Signal reception
    - (Timers)
  - Actions:
    - Activities
    - Methods of the class/block
  - Available variables
    - Attributes of the class/block

State Machine should be set as *classifier behavior* of the block

Activity Diagram, Interactions

Activity Diagram

- Definition of actions:
  - Directly in the State Machine
  - As the description of class/block methods
- Send Message action
  - Provides event for the State Machine

Interactions

- Sending and reception of messages
  - Provides events for the State Machine
- Behavior behind a Lifeline (protocol state machine)

EXTRA: CODE GENERATION FROM UML STATE MACHINES

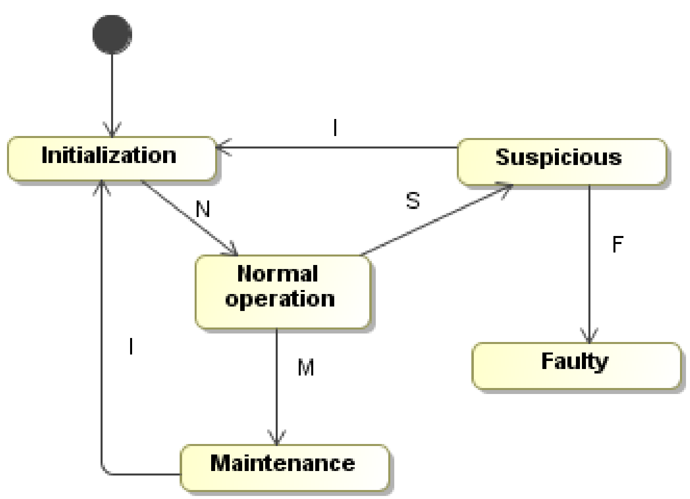
With Switch-Case

With arrays and pointers

Motivation

- Modeling of embedded systems/components
  - Usually with state machines
  - Diagram is easily comprehensible
  - Code can be very complex due to many branches

    → Code generation




```java
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;
        }
    }
}
```

Tools

Depending on the goal and platform

- Low-level embedded environments
  - State Machine: No hierarchy, parallelism
  - Language: C, Assembly
  - Constructs: goto, jmp, if-then-else, switch-case…

Tools

Depending on the goal and platform

- Low-level embedded environments
  - State Machine: No hierarchy, parallelism
  - Language: C, Assembly
  - Constructs: goto, jmp, if-th[illegible]lse, switch-case…

Every state machine can be „flattened”

Tools

Depending on the goal and platform

- Low-level embedded environments
  - State Machine: No hierarchy, parallelism
  - Language: C, Assembly
  - Constructs: goto, jmp, if-then-else, switch-case…
- High-level software environments (e.g. web protocols)
  - State Machine: May use every element
  - Language: C, C++, Java, C#, etc.
  - Constructs: switch-case, object orientation

Discussed methods

1. Simple state machines with Switch-Case

2. Simple state machines with arrays and pointers

Simple state machines with Switch-Case

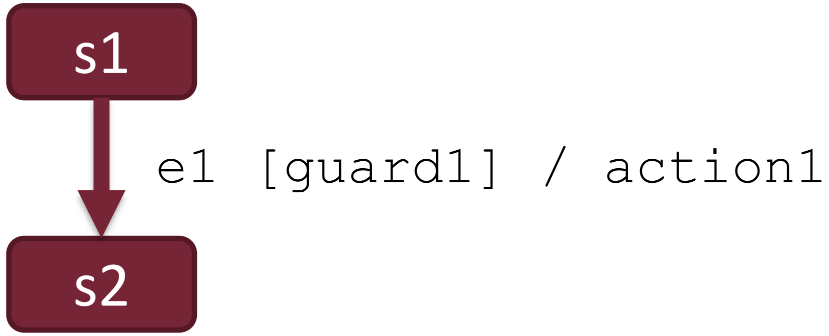
Needed:

- Integer or Enumeration type for states
- Integer, Enumeration type or class for events
- State variable + additional optional variables
  - *State s* = [initial state];
- Event handler method:
  - *void processEvent(Event e)*

Simple state machines with Switch-Case

Event handler method:

```
void processEvent(Event e) {
    switch (s) {
    case s1:
        switch (e) {
        case e1:
            if (guard1(e)) {
                action1(e);
                s = s2;
            }
            break;
        }
        break;
    ...
    }
}
```



guard(e):

- Evaluation of guard (can depend on *e*)

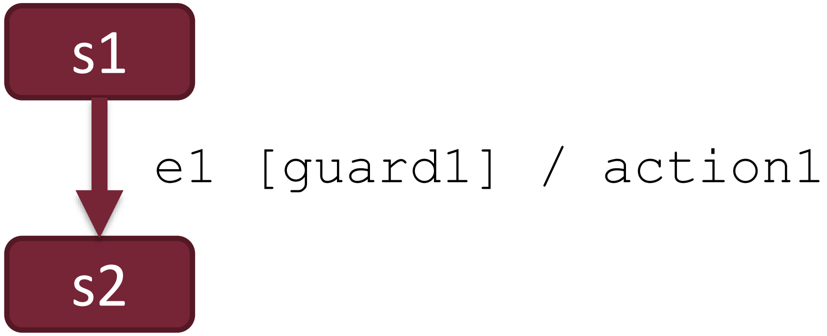
action(e):

- Execution of action (can depend on *e*)

Simple state machines with Switch-Case

Event handler method:

```
void processEvent(Event e) {
    switch (s) {
    case s1:
        switch (e) {
        case e1:
            if (guard1(e)) {
                action1(e);
                s = s2;
            }
            break;
        }
        break;
    ...
    }
}
```



guard(e):

```
…
s1exit();
action1(e);
s = s2;
s2entry();
…
```

Simple state machines + arrays, pointers

Needed:

- Everything as before
- A 2-dimensional array for *next states*
  - *State nextState[#states][#events]*
- A 2-dimensional array for *guard functions*
  - *bool (*guards)[#states][#events](Event e)*
- A 2-dimensional array for *actions*
  - *void (*actions)[#states][#events](Event e)*

Simple state machines + arrays, pointers

Initialization of arrays:

```
State nextState[#states][#events] =
      {{ s2, s1, ...}, { ... }, ...};
bool (*guards)[#states][#events](Event e) =
      {{ guard1, guard2, ...}, { ... }, ...};
void (*actions)[#states][#events](Event e) =
      {{ actions1, action2, ...}, { ... }, ...};
```

Event handler method:

```
void processEvent(Event e) {
      if (guards[s][e](e)) {
            actions[s][e](e);
            s = states[s][e];
      }
}
```

Simple state machines + arrays, pointers

Initialization of arrays:

```
State nextState[#states]
      {{ s2, s1, ...}, {
bool (*guards)[#states][#
      {{ guard1, guard2,
void (*actions)[#states]
      {{ actions1, actio
```

```
…
exit[s]();
actions[s][e](e);
s = states[s][e];
entry[s]();
…
```

Event handler method:

```
void processEvent(Event e) {
      if (guards[s][e](e)) {
            actions[s][e](e);
            s = states[s][e];
      }
}
```