PART 3: UML Dynamic Modelling Notations

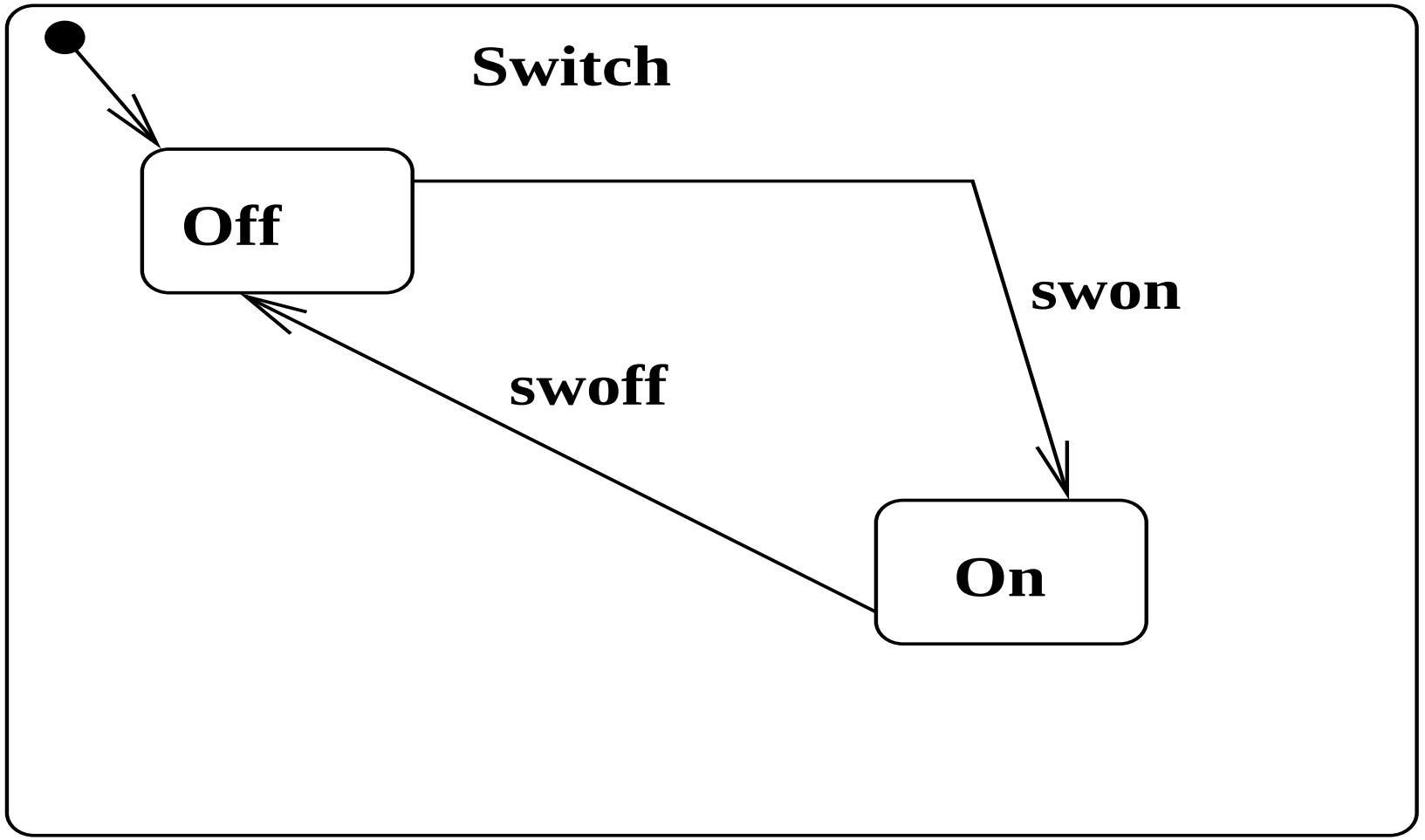
- State machines/statecharts
- Collaboration diagrams
- Sequence diagrams
- Activity diagrams.

Chapter 19 of the textbook is relevant to this part.

State machines

- State machines describe dynamic behaviour of objects, show life history of objects over time + object communications.
- Used for real-time system design (eg., robotics); GUI design (states represent UI screens/modes).

Example shows simple state machine with two states, *On* and *Off* and transitions between them.



Simple state machine

State machines

Elements of a state machine are:

States: Rounded-corner boxes, containing state name.

Transitions: Arrows from one state, *source* of transition, to another, *target*, labelled with event that causes transition.

Default initial state: State of object at start of its life history. Shown as target of transition from initial pseudostate (black filled circle).

Termination of state machine can be shown by 'bullseye' symbol.

State machines

UML divides state machines into two kinds:

1. **Protocol state machines** – describe allowed life histories of objects of a class. Events on transitions are operations of that class, transitions may have pre and post conditions.

   Transitions cannot have generated actions (although we will allow this).

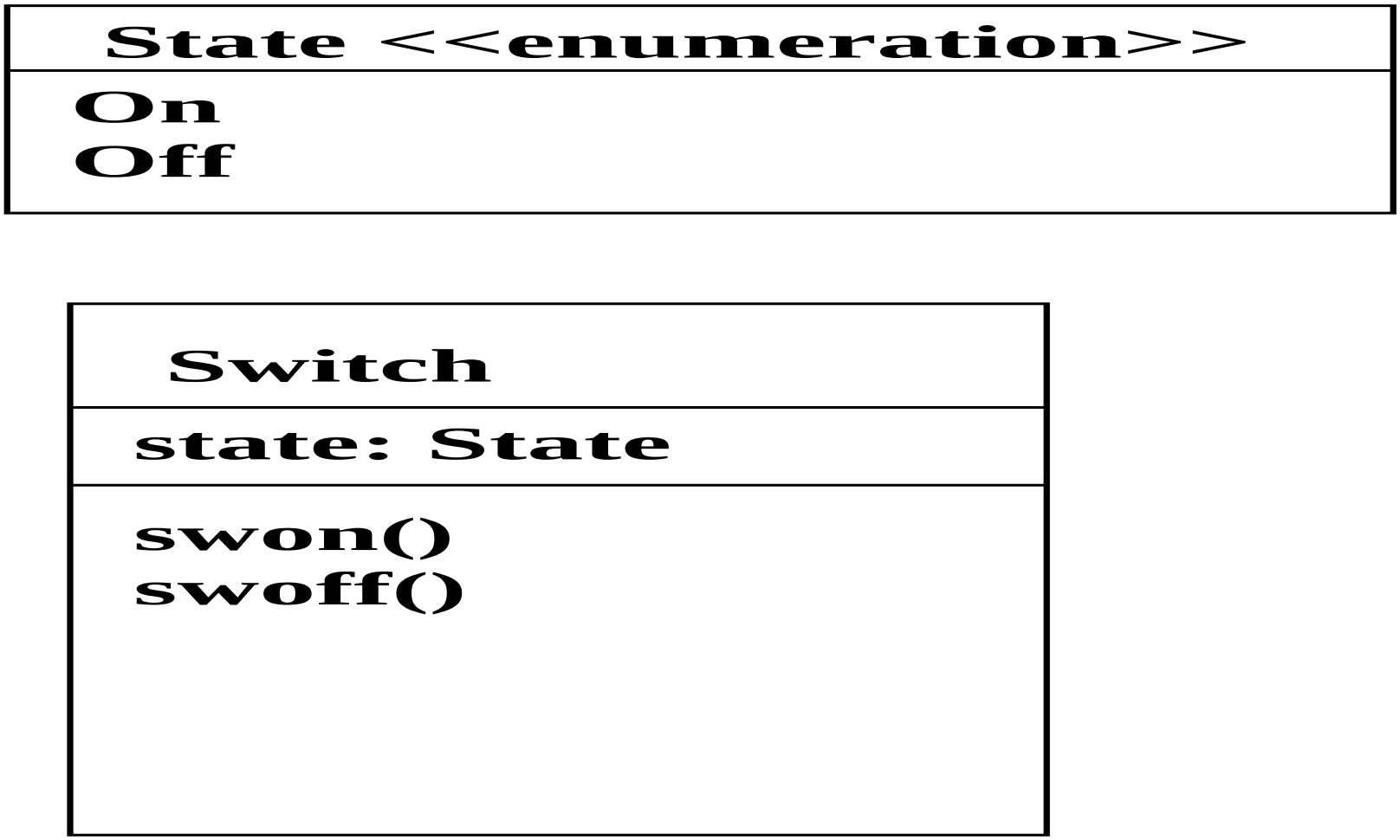
2. **Behaviour state machines** – describe operation execution/implementation of object behaviour. Transitions do not have postconditions, but can have actions.

State machines describe behaviour of objects of a particular class, or execution processing of an operation.

State machines

- Class diagrams describe system data, independently of time.
- State machines show how system/objects can change over time.
- Switch state machine is protocol state machine for objects of *Switch* class.
- Life history of object *obj* of *Switch* is sequence of alternations between *On* and *Off* states, beginning in *Off*:

$$Off \longrightarrow_{swon} On \longrightarrow_{swoff} Off \longrightarrow_{swon} \ldots$$



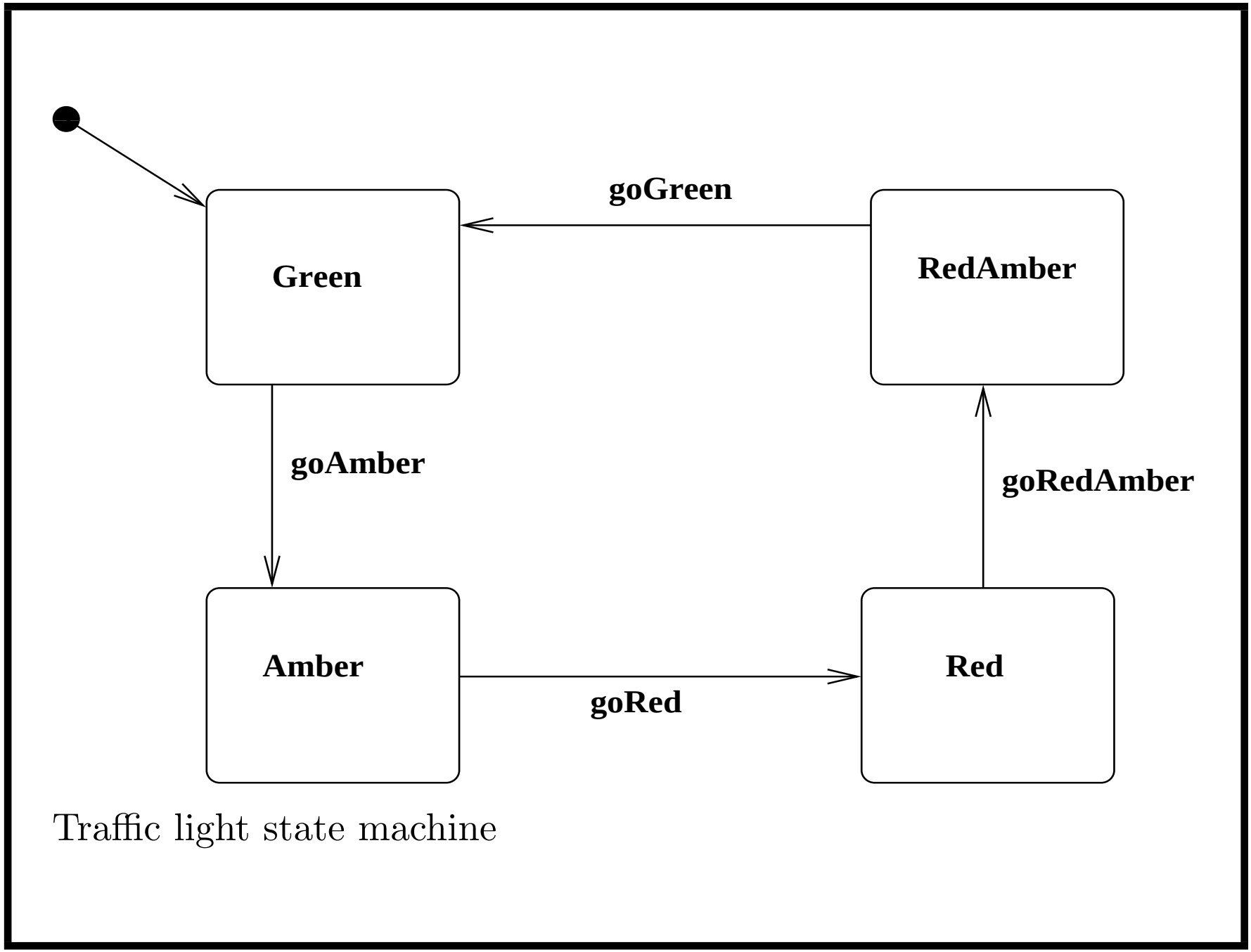
Switch class

State machine semantics

For protocol state machine attached to a class:

1. Each object of class begins life history in default state of state machine.
2. If object *obj* is in state s and event α occurs on *obj*, then if there is transition labelled with α with source s, this transition occurs, object moves into target state of transition.

   If no transition for α from state s, object remains in state s.

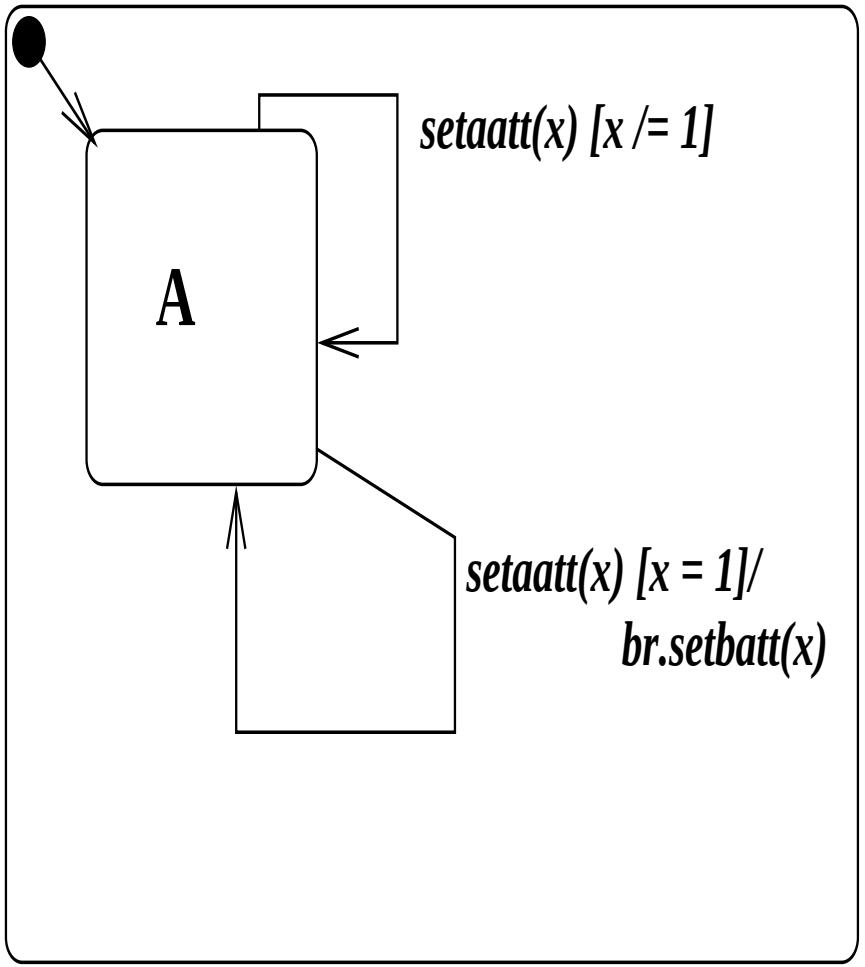


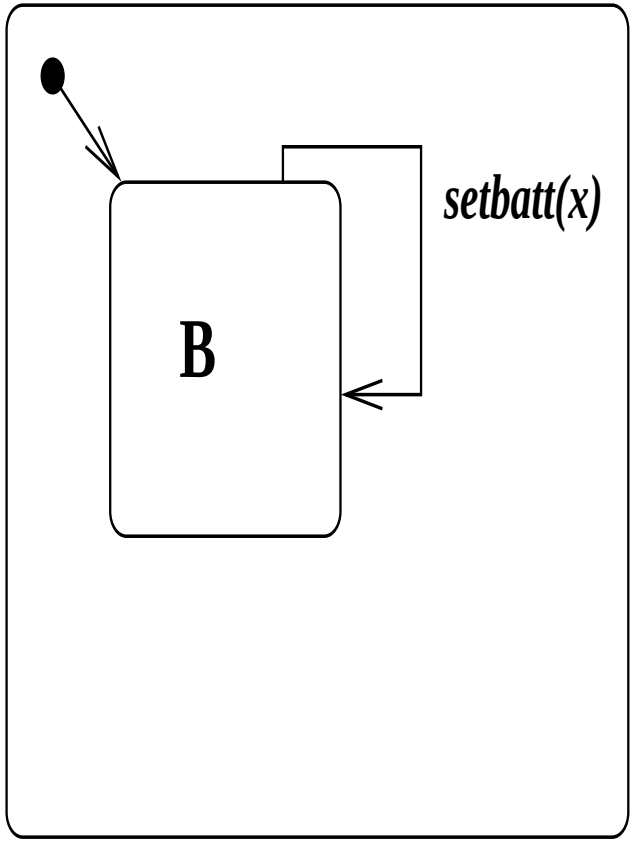
Traffic light state machine

Transition Guards and Actions

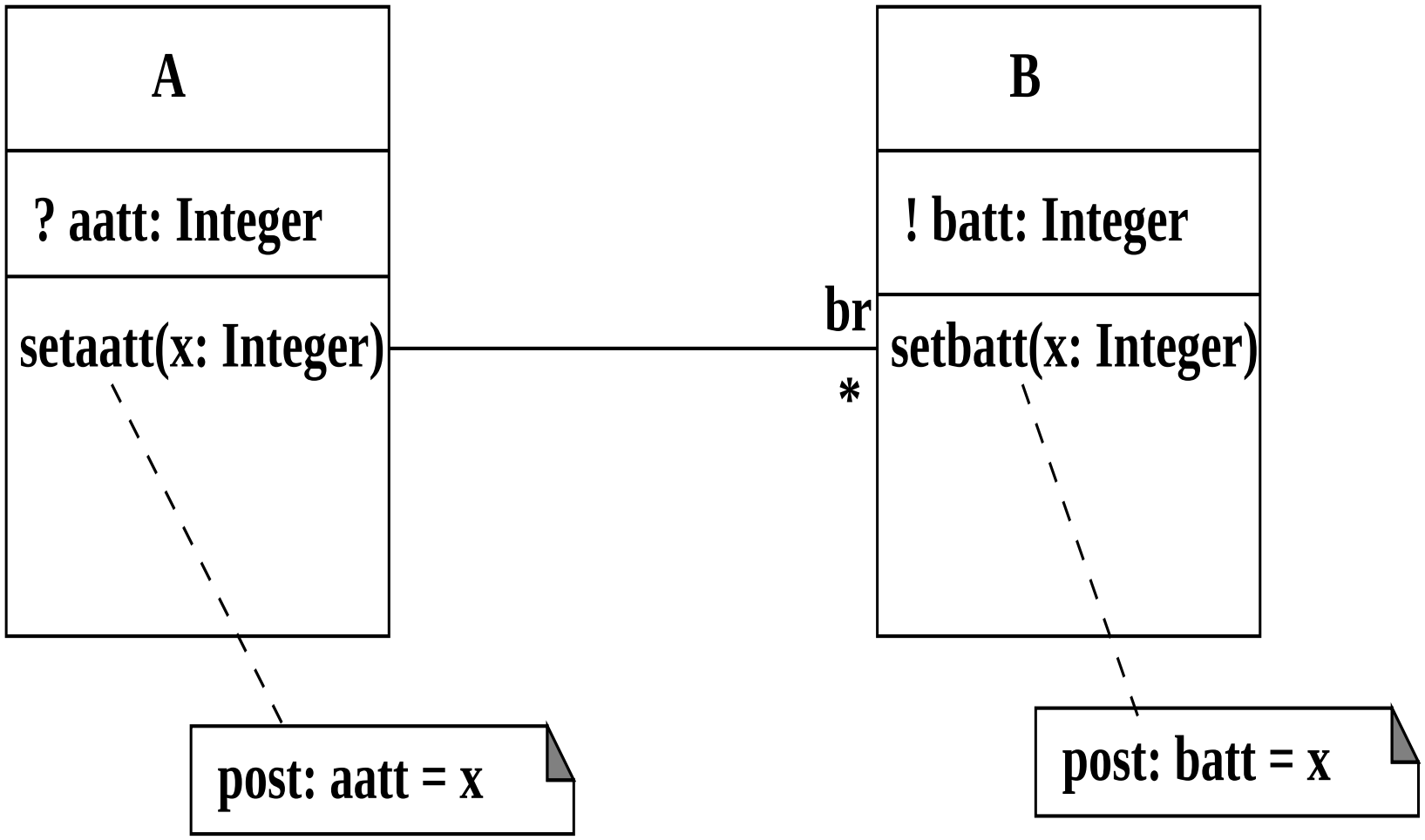
- Transitions may have *guards* and *actions*.
- Guard is evaluated when transition event occurs on object, if object is in source state of transition.
- If guard is true, transition takes place, otherwise it does not.
- Action on transition describes event(s) generated when transition occurs. Usually operation invocations on objects of supplier classes of state machine class.

Example: $setaatt(x)$ also sets $b.batt$ to x for each $b : br$, when $x = 1$:





State machines of classes A and B



Classes A and B

Actions

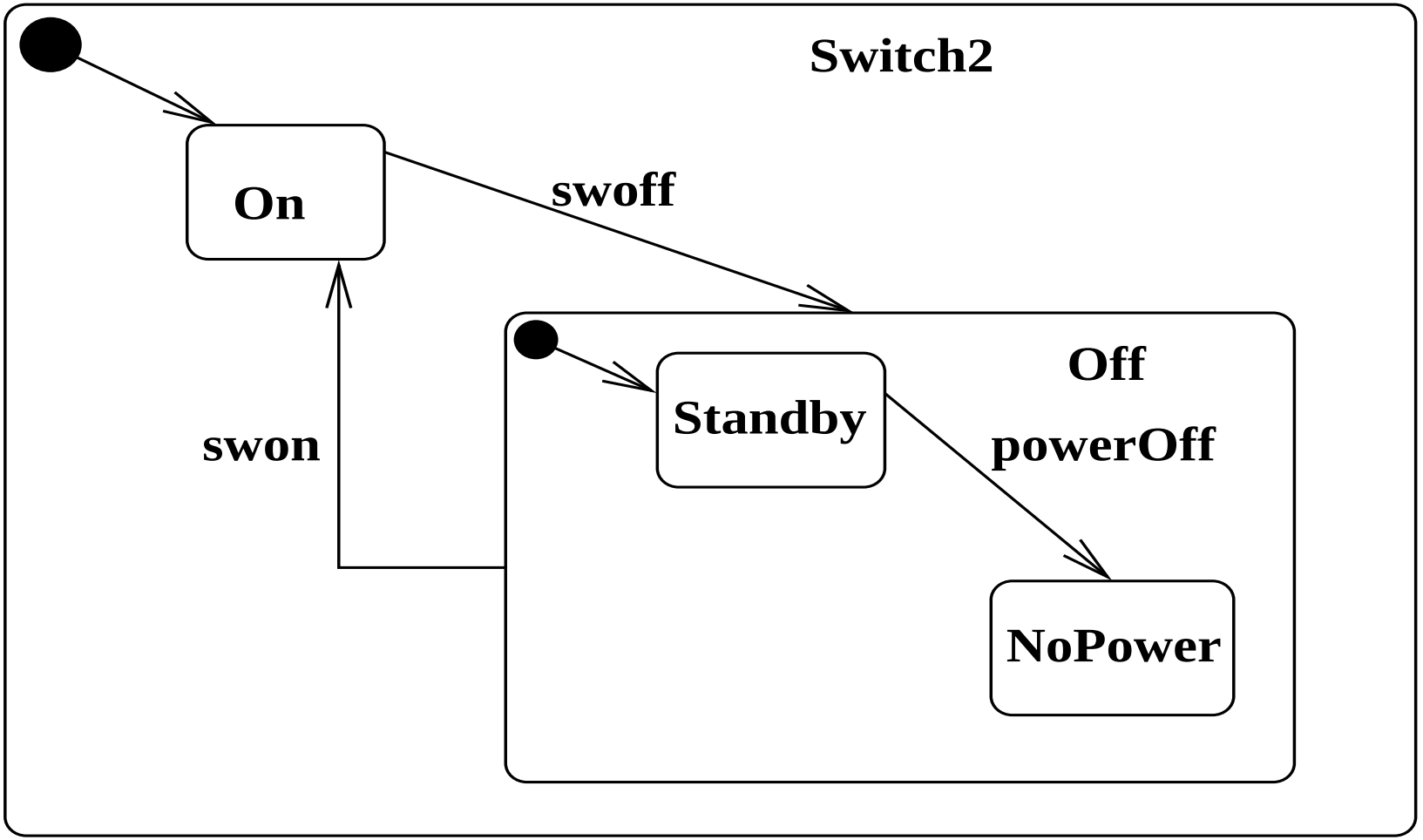
- Actions written following forward slash, after event and guard (if present). Multiple actions can be specified, separated by / or ; .
- Actions can include updates *feature* := *val* of local attributes/roles, or invocations of operations on supplier objects.
- Postconditions of operation *op* express its state-independent behaviour, actions of *op*-triggered transitions express its state-dependent behaviours.

Rules for state machines

1. For each state s, and event α of state machine, at most one unguarded transition for α from s.
2. If several transitions for α, all must be guarded and have disjoint guards, so impossible for more than one to be true at once.
3. For simple state machines an object is in exactly one state at all times after its creation and before its destruction.

Composite states

- States can be *nested* within others to express relationships of generalisation/specialisation, and to simplify diagram.
- Following shows example of composite state *Off*, containing substates *Standby* and *NoPower*.
- When *swoff* occurs in state *On*, system goes to *Standby*, default state of *Off*. When *swon* occurs, if system is in either *Standby* or *NoPower*, then next state is *On*.

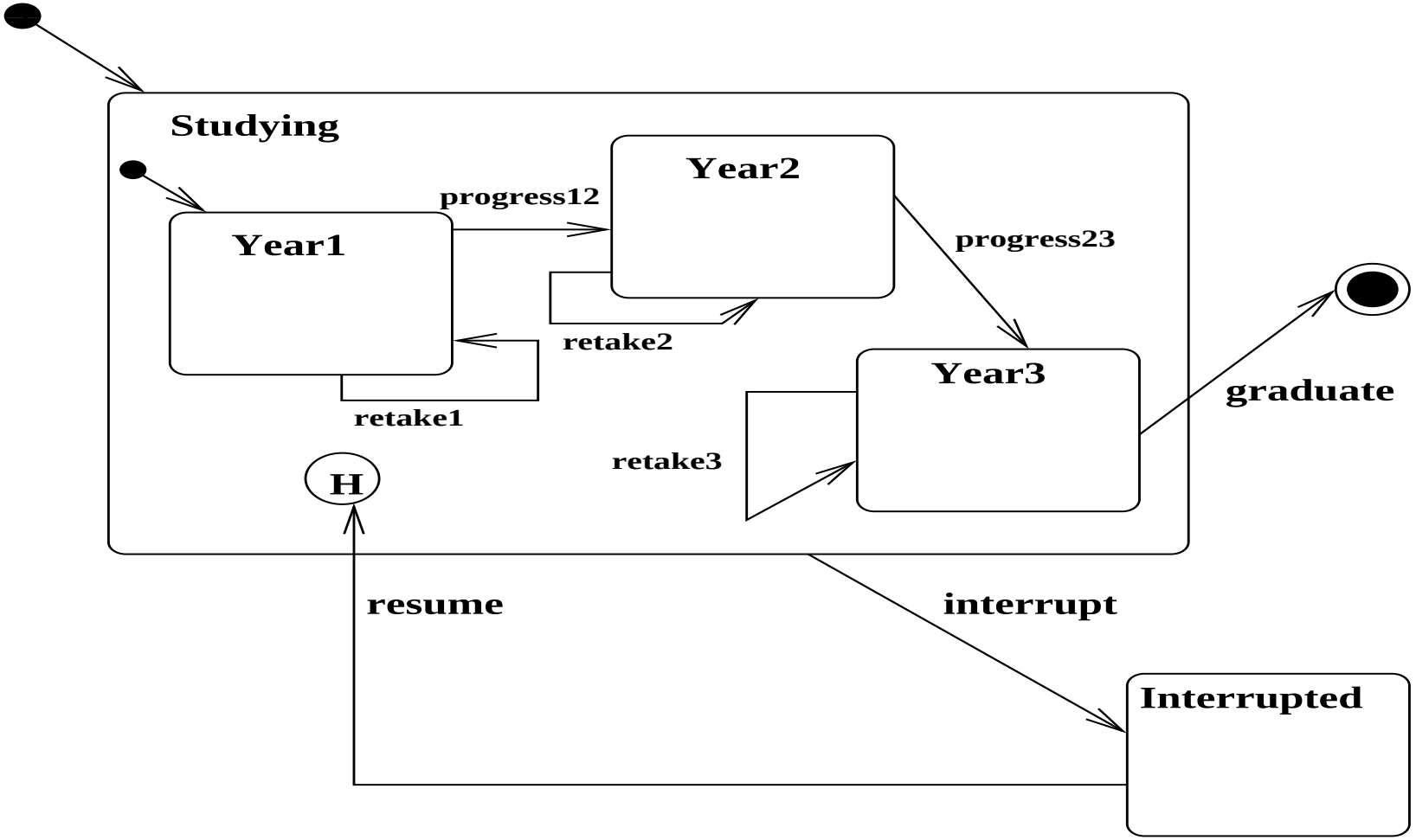


State machine with nesting

Student life history example

State machine for BSc students (3 year), with:

- States $Year1$, $Year2$, $Year3$ as substates of *Studying*
- States *Interrupted* and terminal (graduated) at top level
- Transitions for progression, repeating year, interruption and graduation.



State machine of students

Student life history example

- Transition from superstate *Studying* to *Interrupted* equivalent to 3 separate transitions from $Year1$, $Year2$, $Year3$ to *Interrupted*.
- *graduate* transition only occurs for $Year3$ students.
- *Final state* shown as bullseye symbol, *history state* denoted by H symbol.
- A transition to history state has actual target the most recently occupied substate of composite state containing history state.
- When student resumes their studies, they return to year they were in when they interrupted.

History states should only be used if essential.

Nested states

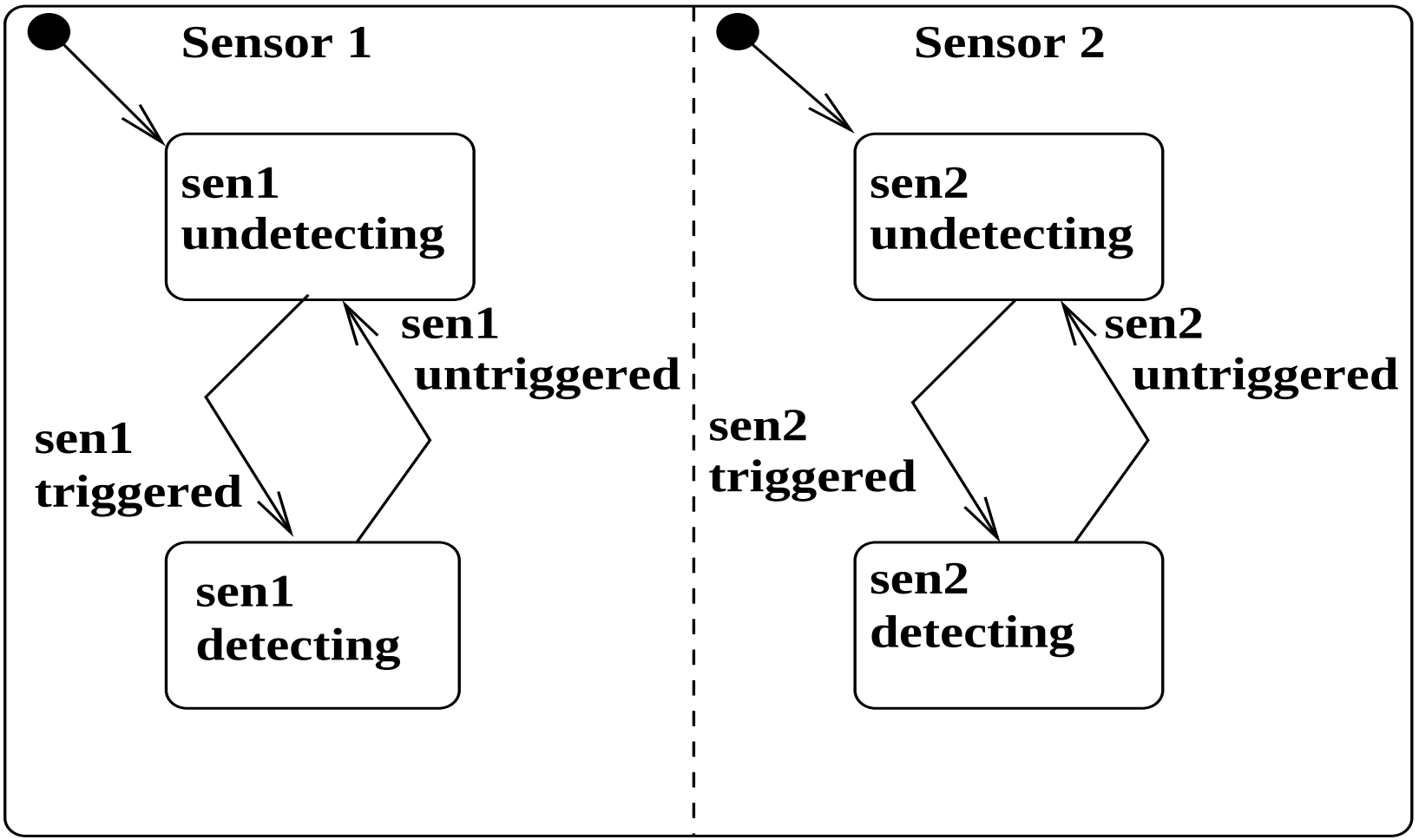
- If object is in basic state s, also in every state that contains s.
- A $Year1$ student is also in *Studying*, etc.
- States can have attached constraints, evaluate true for any object while it is in state, including any substate of state.
- Constraint written in square brackets after state name.

Concurrent states

- States can be *concurrent*, consist of 2 or more composite states, *simultaneously* occupied.
- Eg., monitoring system with two sensors, sensor 1 and sensor 2 can change state independently of each other.

  Possible sets of states are

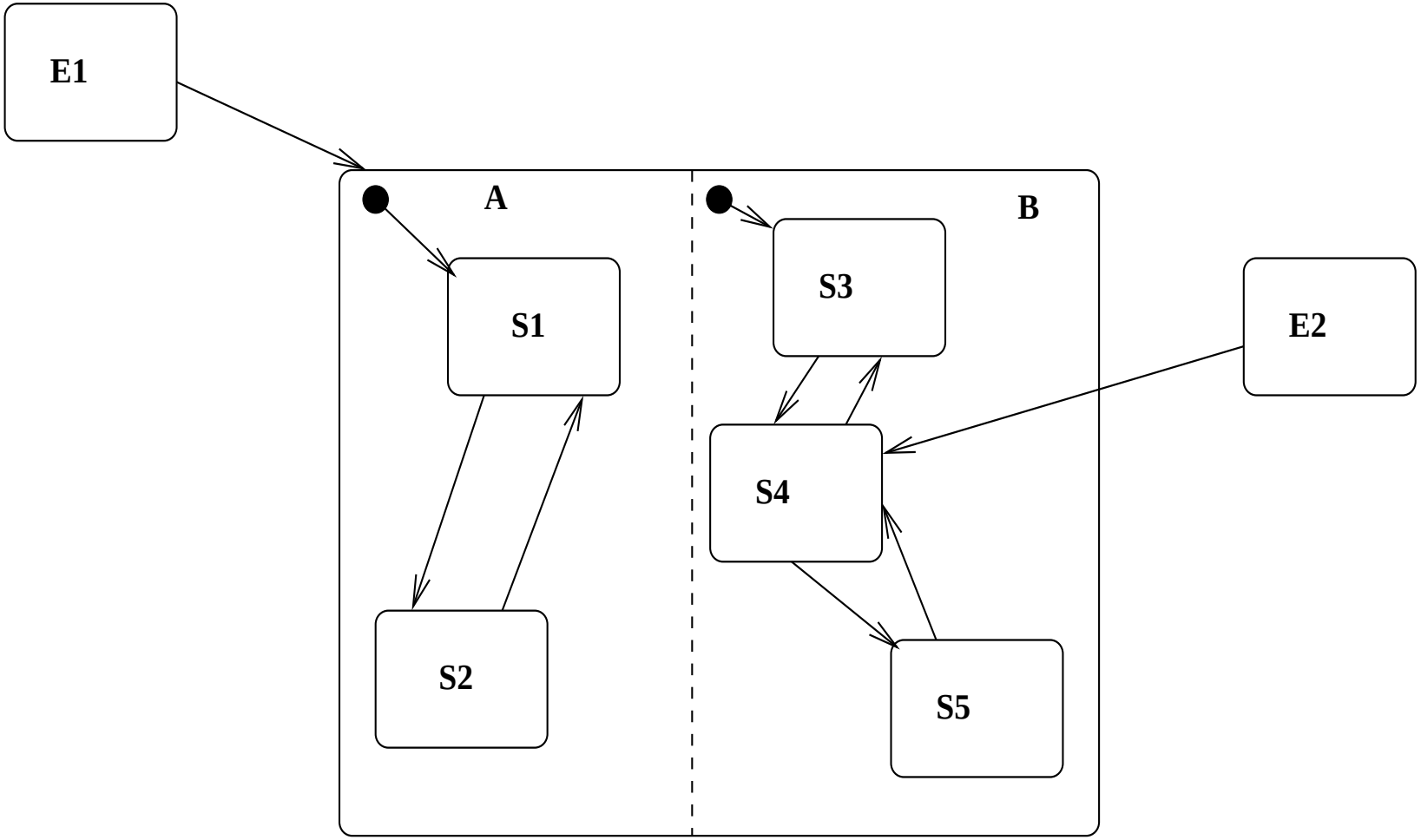
  $\{sen1undetecting, sen2undetecting\}$
  $\{sen1undetecting, sen2detecting\}$
  $\{sen1detecting, sen2undetecting\}$
  $\{sen1detecting, sen2detecting\}$



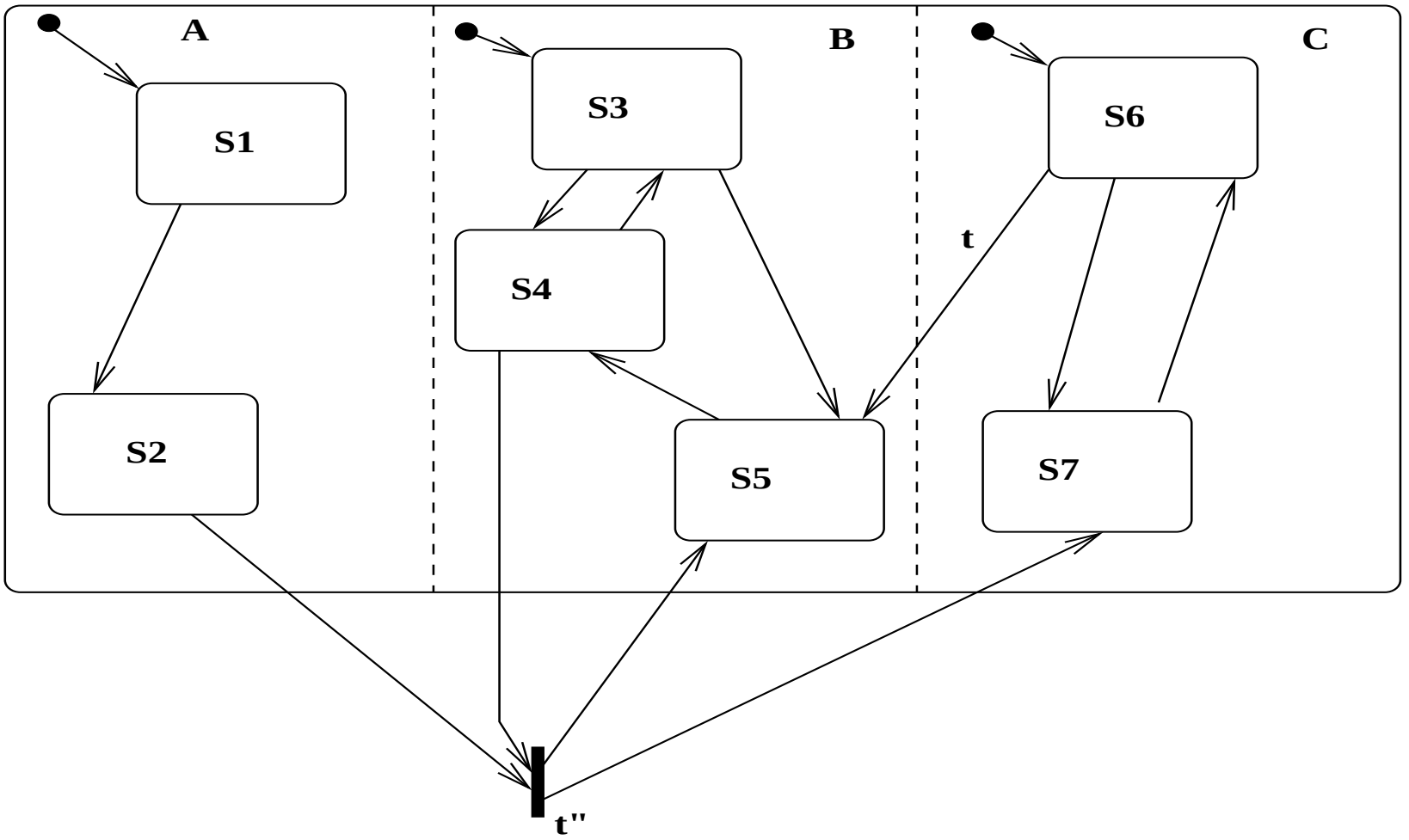
State machine with concurrent states

Concurrent States

- When concurrent composite state entered, each subcomponent is entered, usually at default states.
- Also possible to specify direct entry to specific component states.
- Transitions may exit from multiple sources: these sources must be from different concurrent components of concurrent state.
- Transitions can enter multiple targets, in different concurrent components of concurrent state.
- Such multi-transitions are written with vertical bar joining exit/entry arrows.



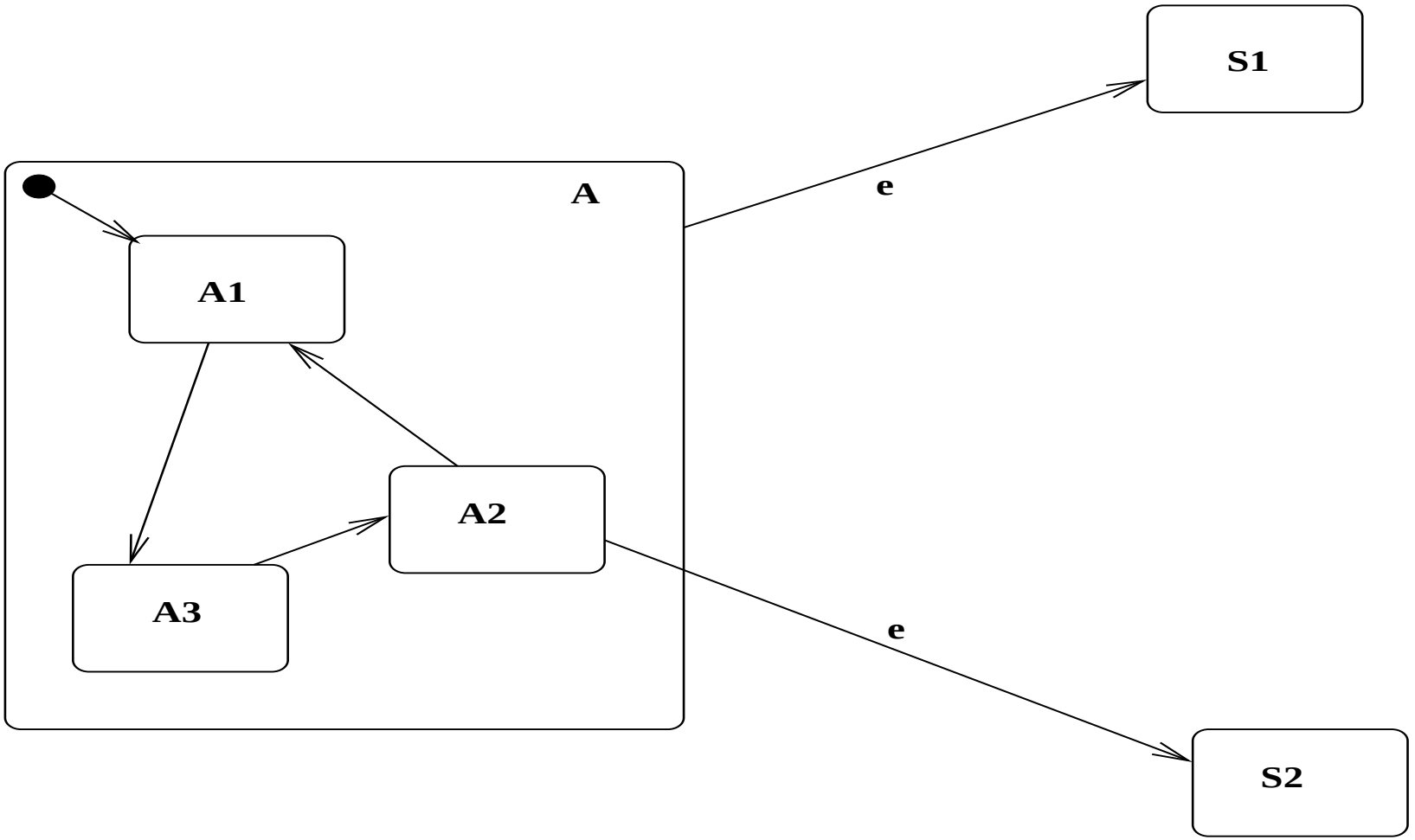
Entering a concurrent state



State machine with strange transitions

Transition Priorities

- If event has transitions both from substate of a state and from state itself, substate transition has *higher priority* and will fire if event occurs when substate is occupied.
- In following diagram, transition from $A2$ has priority for event e when object is in state $A2$.

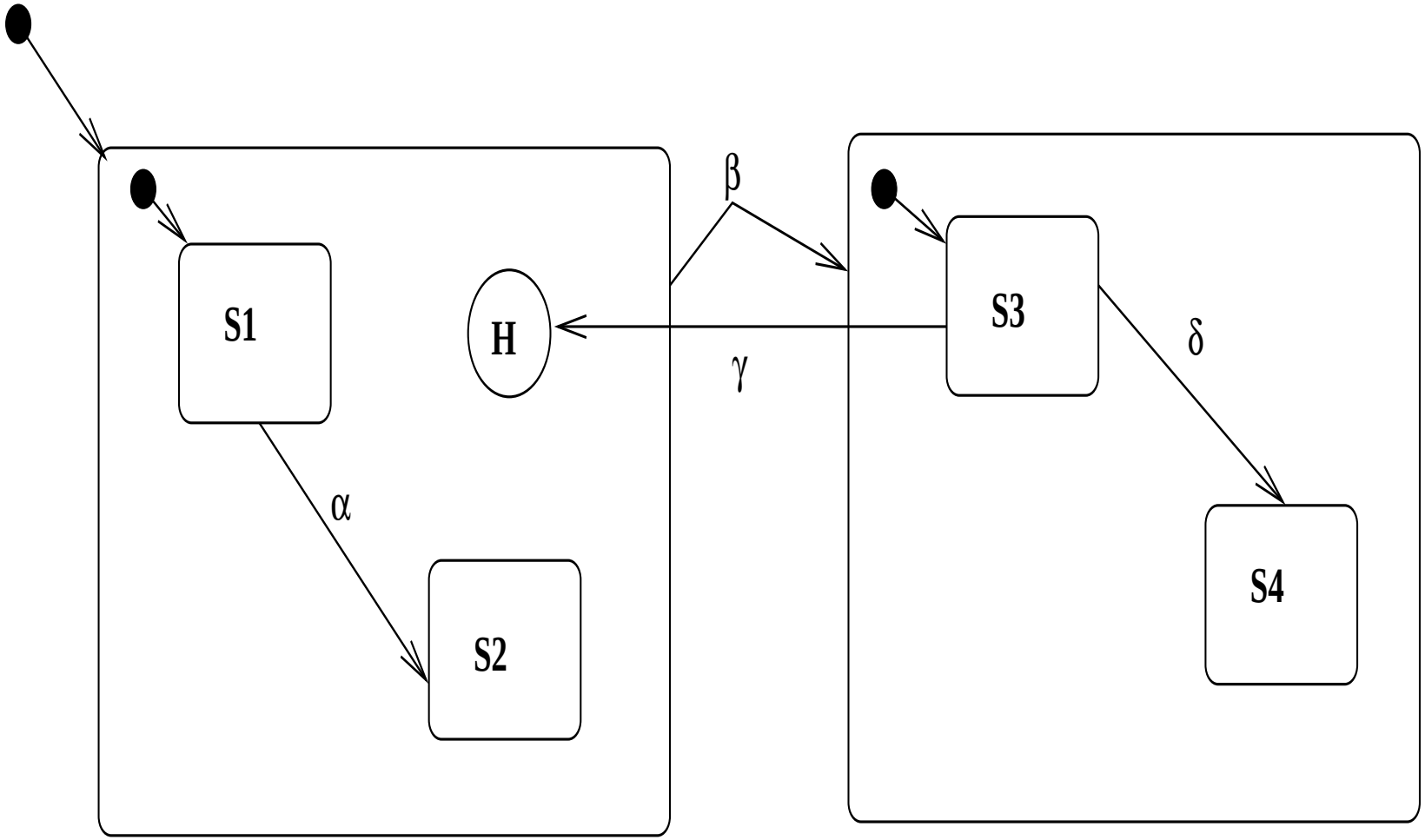


Transition priority example

History States

- History states 'remember' what substate of a composite state was last occupied.
- History state written as an H within a circle.
- Transition to circle actually goes to the substate of closest enclosing (non-concurrent) composite state which was last occupied when composite state last exited.
- If composite state not previously occupied, default substate of composite state is entered instead.

Sequence of events α; β; γ puts system into state $S2$:

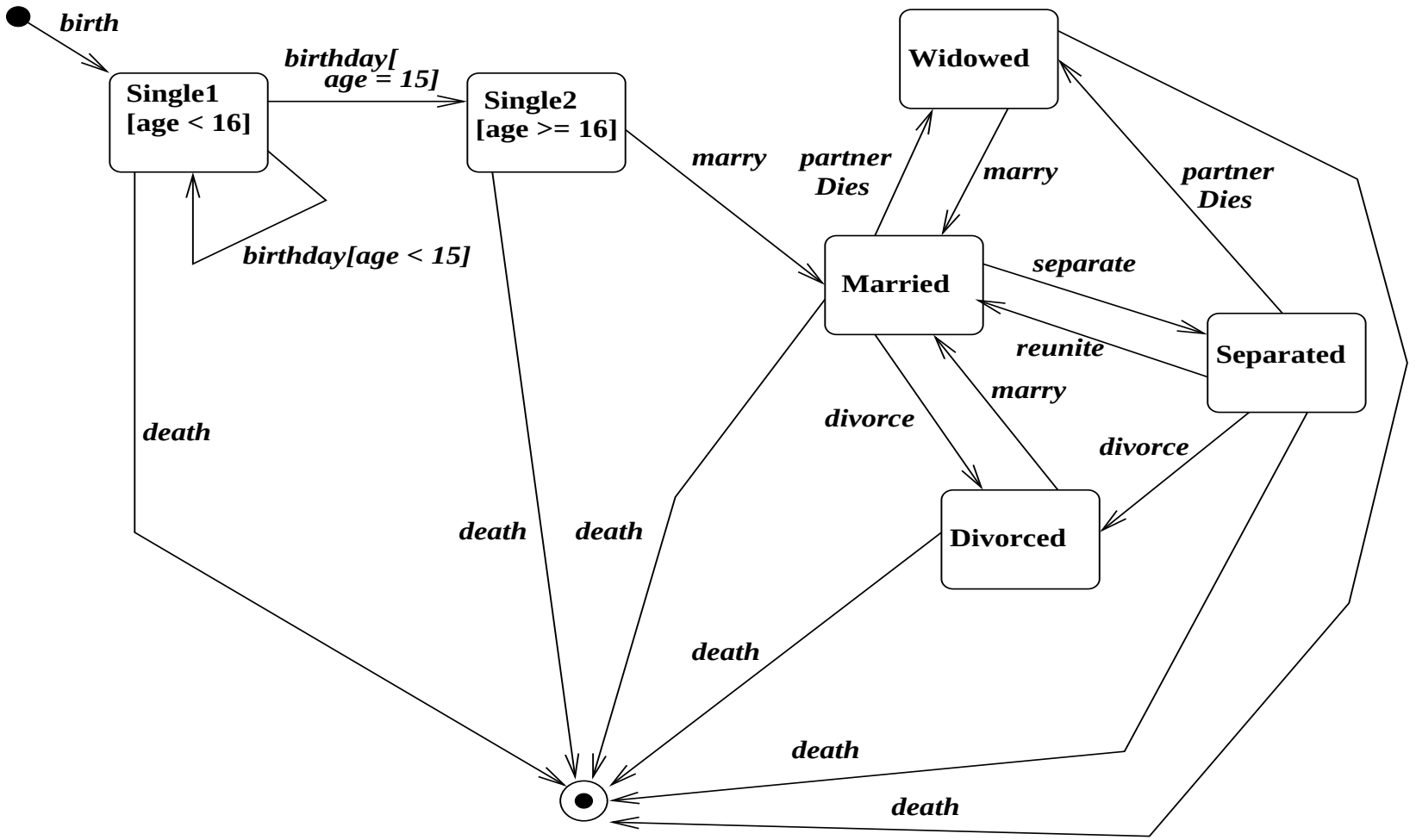


History state example

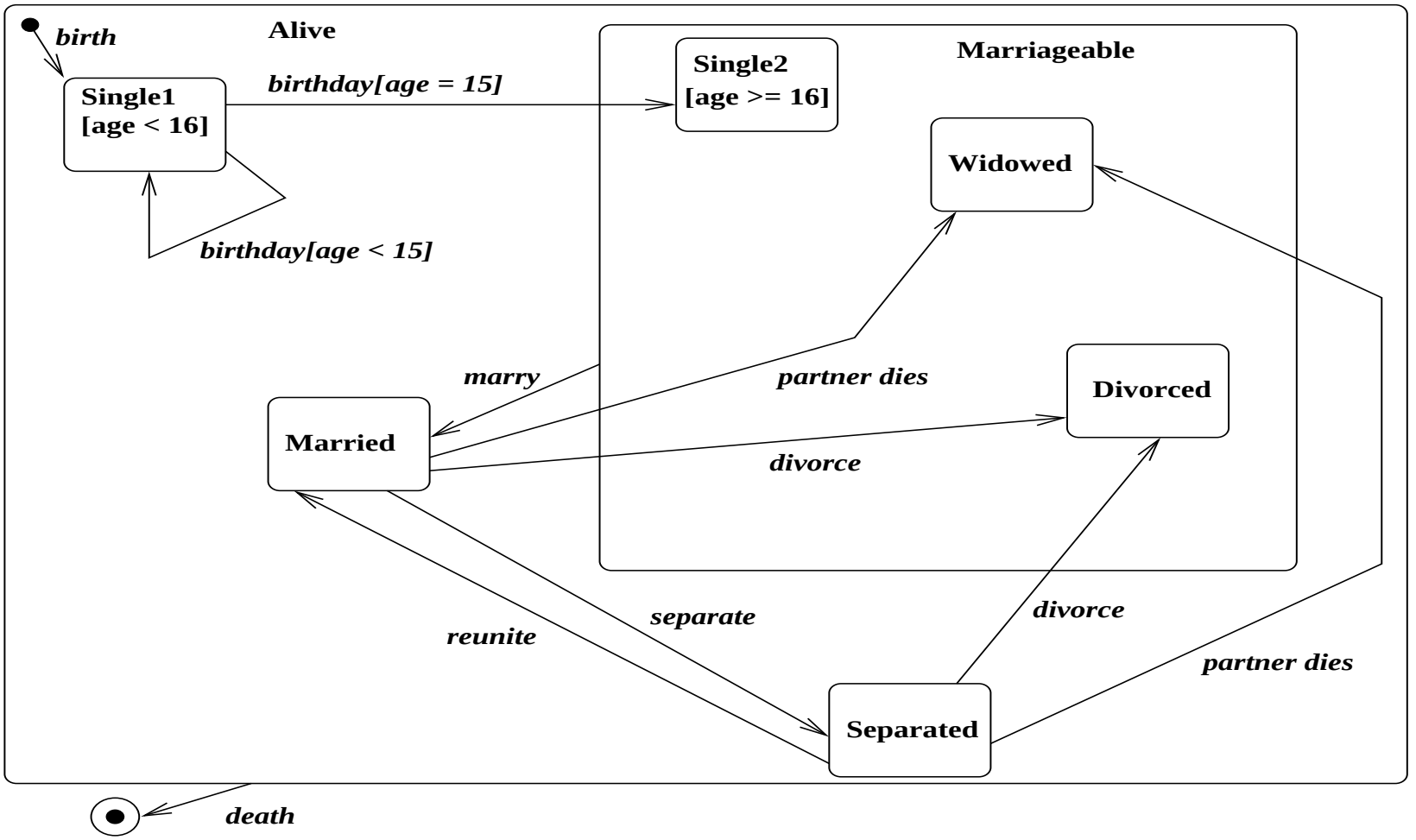
Example of refactoring of state machine: Marriage

Refactoring means re-writing model to be simpler, retaining same meaning.

- Consider problem of specifying history of a person's marital state over time.
- Identify states *single*, *married*, *divorced*, *separated*, *widowed*, and events *marry*, *divorce*, *separate*, *partnerDies*, etc.
- Simplify diagram by grouping similar states together. States $s1$, ..., sn can be grouped if have identical outgoing transitions (same events, guards, target states) to states not in group.



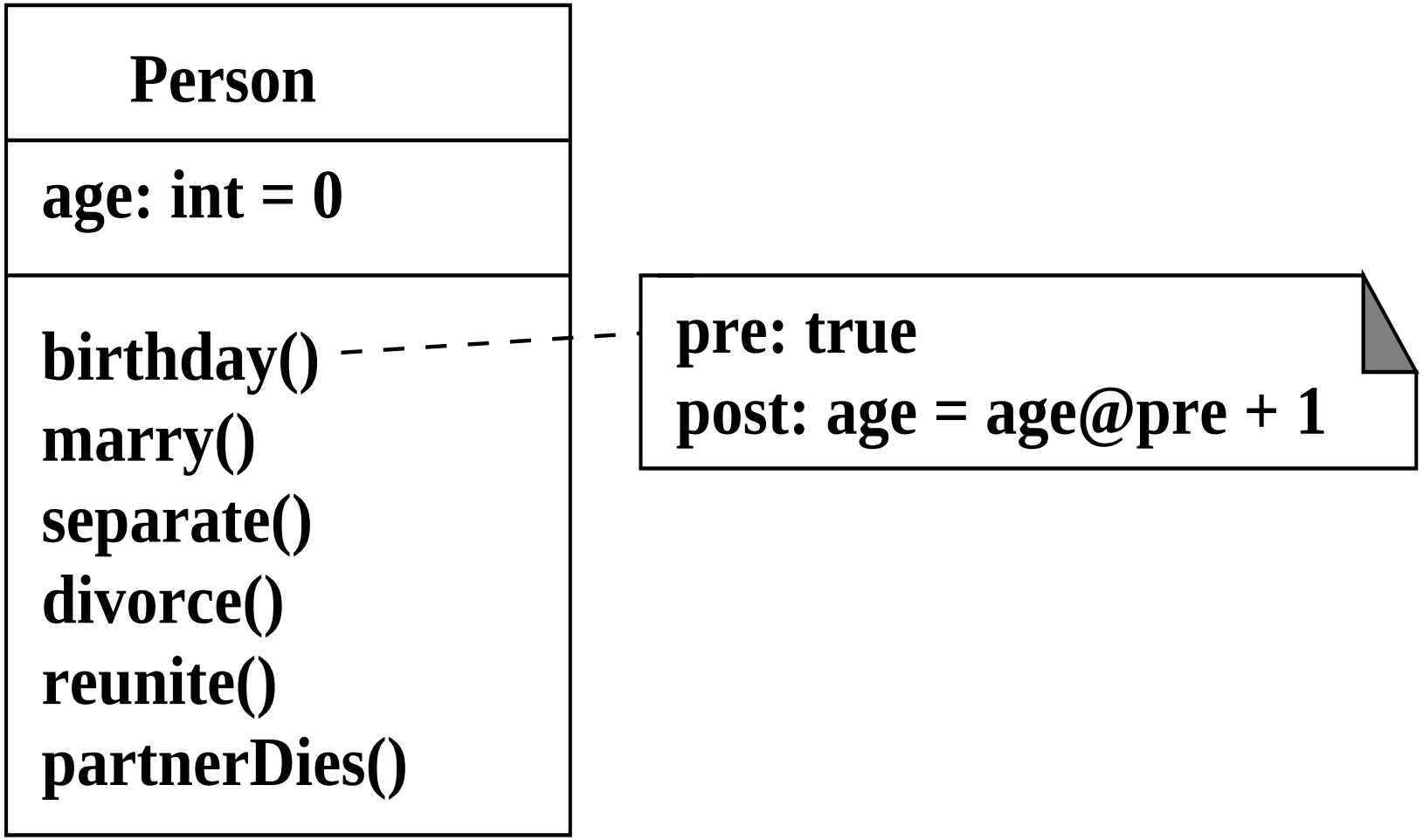
Marital status state machine



Marital status state machine: marriageable states grouped

Improved marriage state machine

- *birthday* event has transitions on every state, only transitions on states where event may change the state are shown.
- Class description for persons shows rest of *birthday* behaviour.

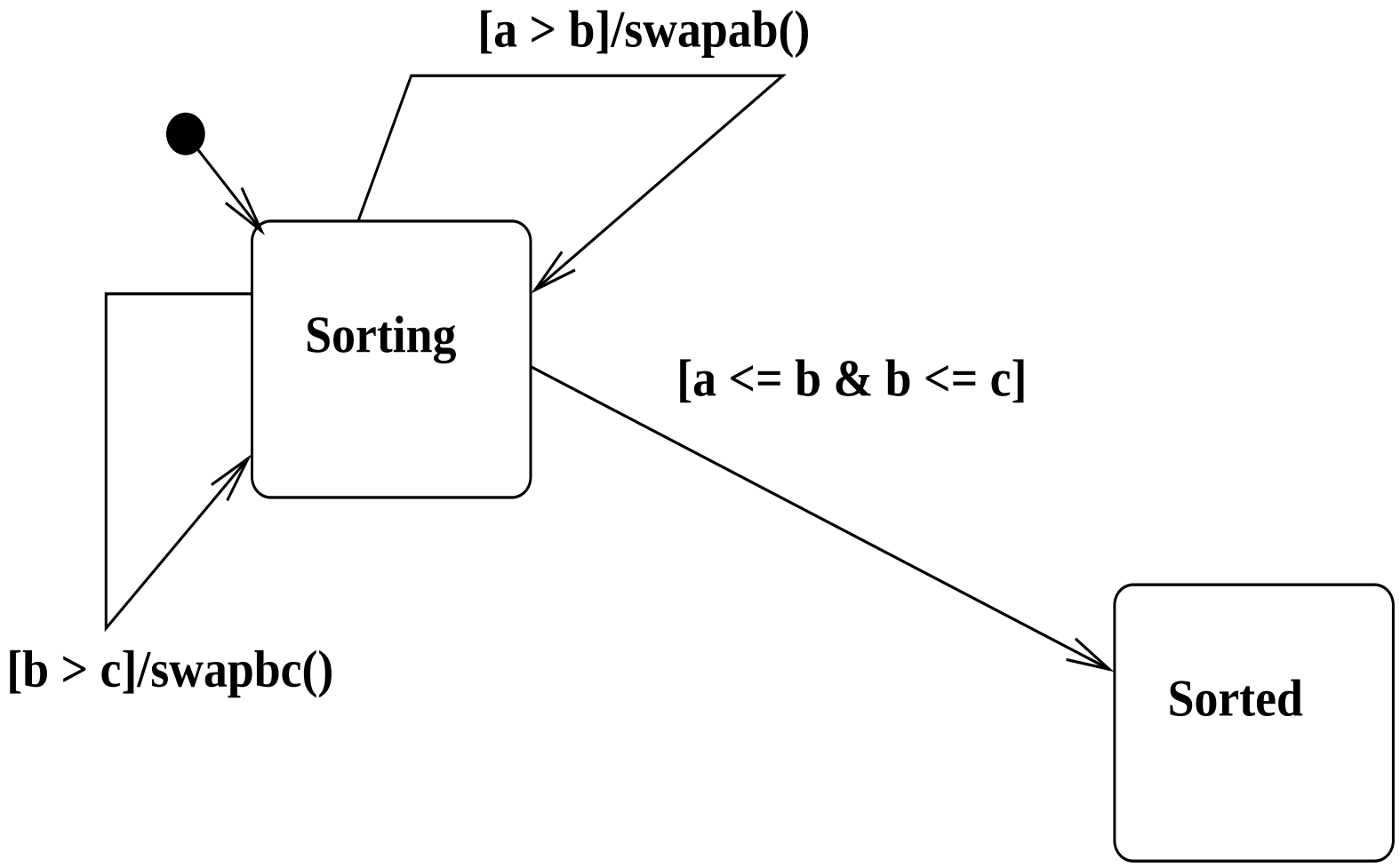


Person class

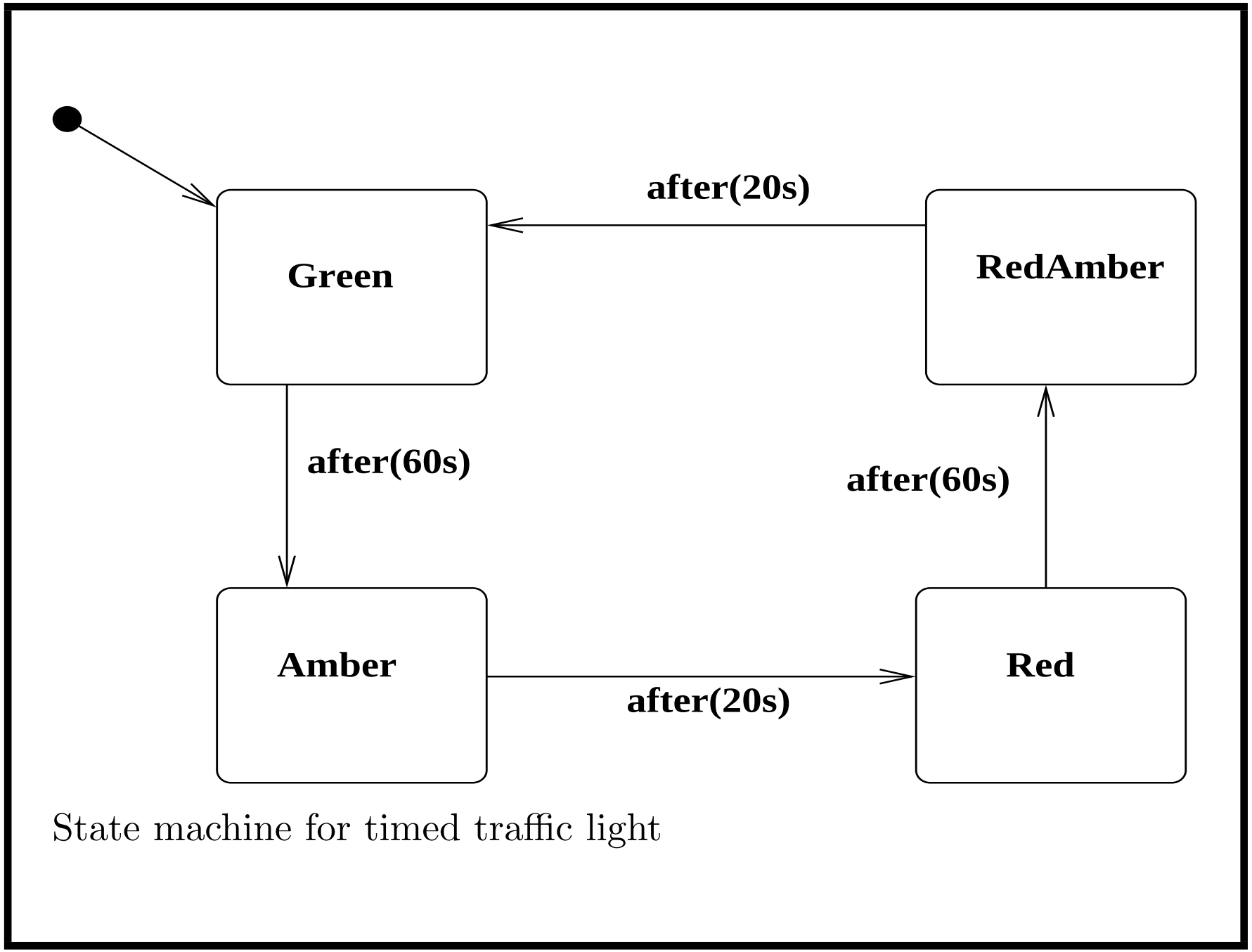
Behaviour state machines

- Define processing of operation, or behaviour of an active class.
- Transitions do not have explicit triggers – take place as soon as source state is completed. May have timeout triggers $after(t)$.
- Guards can overlap – designer must ensure behaviour is sensible if non-deterministic.

Eg., bubblesort to sort values in variables a, b, c in increasing order:



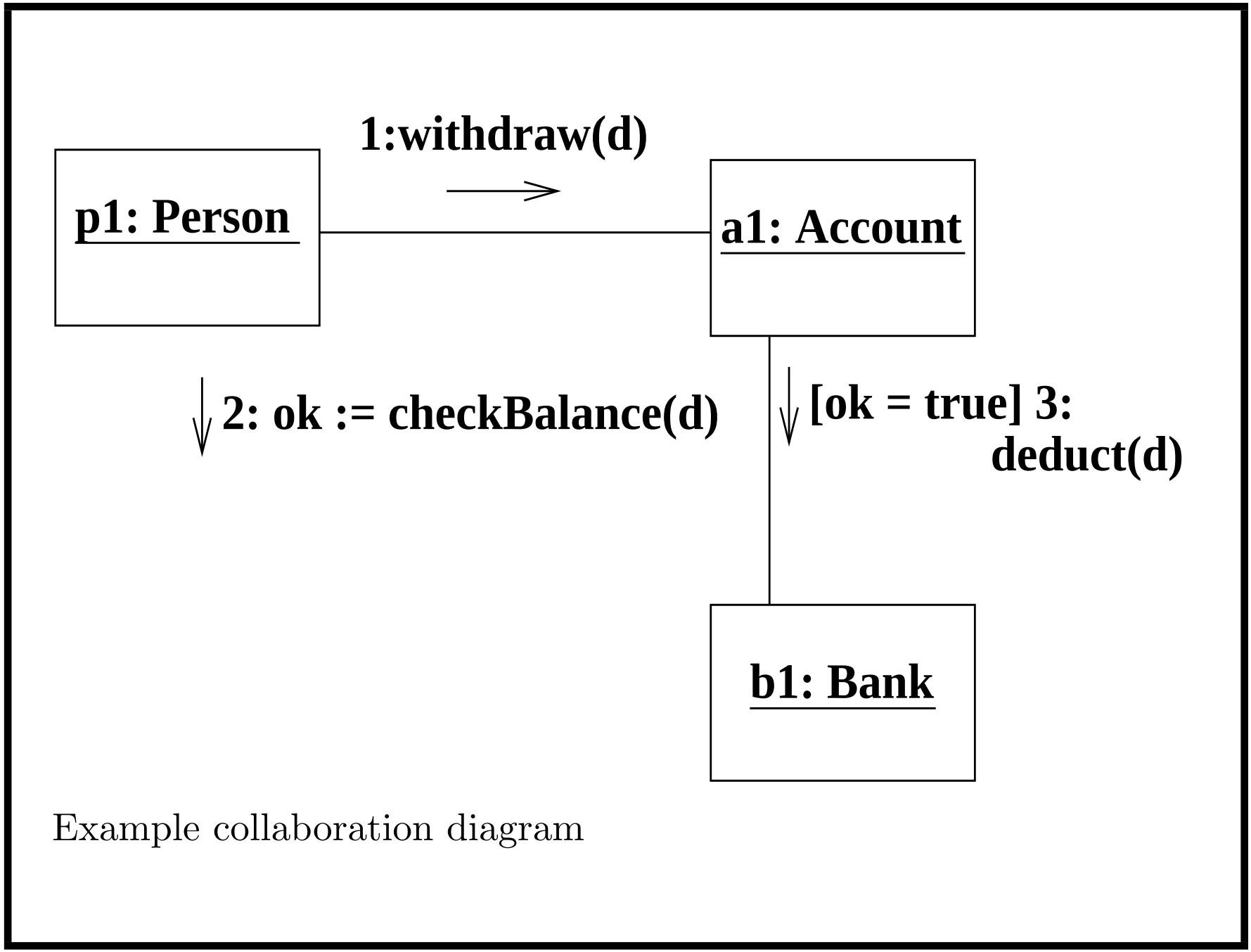
State machine for bubblesort operation



State machine for timed traffic light

Interaction diagrams

- Interaction diagrams show examples (scenarios) of particular use cases.
- Two forms of interaction diagram in UML: *Sequence diagrams* emphasise timing and history of objects. Sequence diagrams relate interaction to state machine models.
- *Collaboration diagrams* emphasise connection to class diagram. Show an interaction as set of messages sent between objects.



Example collaboration diagram

Collaboration Diagrams

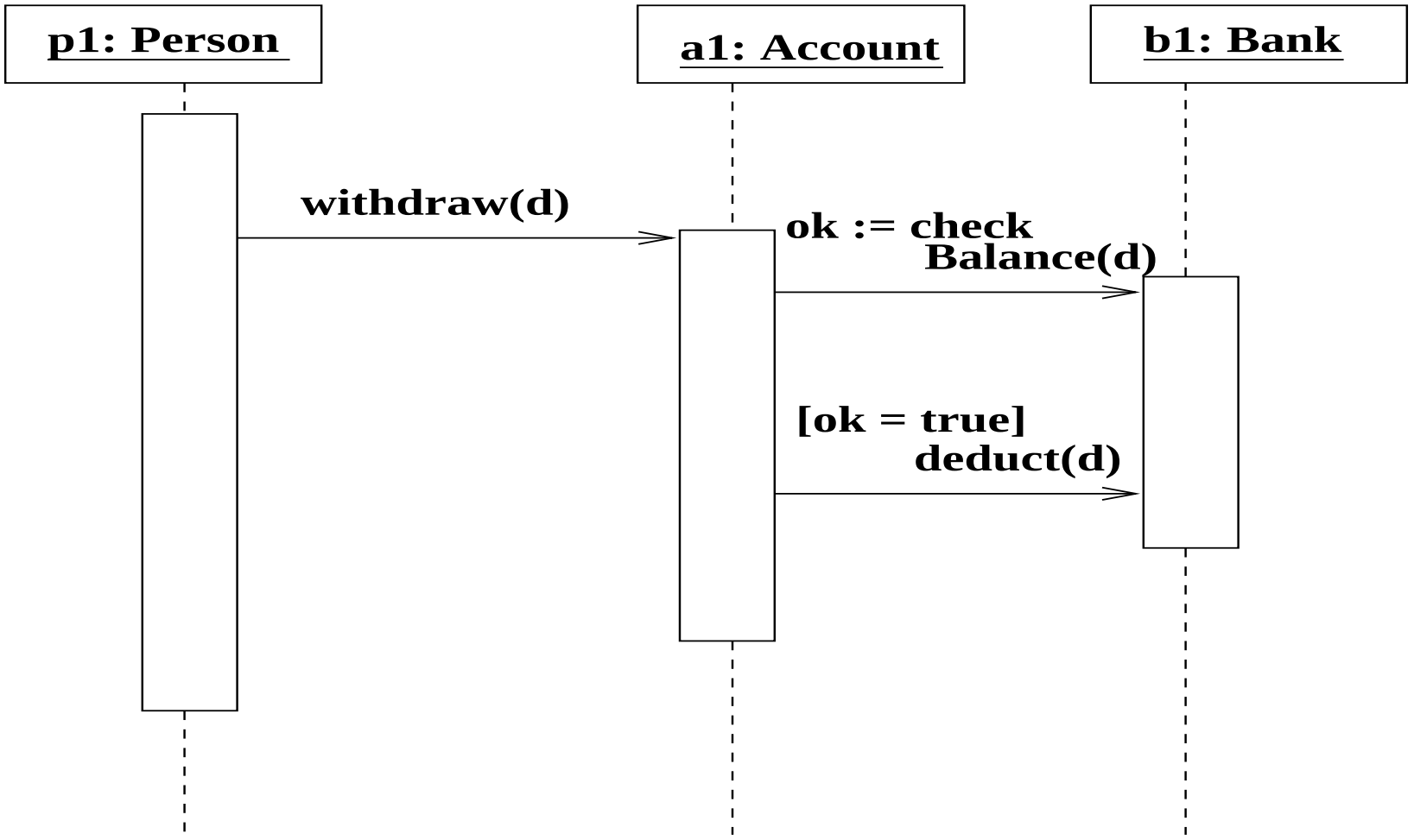
- Objects represented by rectangles containing name/identifier of object, name of its class, all underlined: object: Class.
- Objects created during operation execution noted as $\{new\}$.
- Links (instances of associations) can have arrows indicating navigability: if message is sent from object a of class A to object b of class B then a must have access to b, eg, by association from A to B.
- Messages numbered consecutively starting at 1, nesting indicated by suffixes: eg: 3.1.4 follows 3.1.3 within subprocedure 3.1.
- Conditions given in square brackets: message only sent if condition is true.

Sequence Diagrams

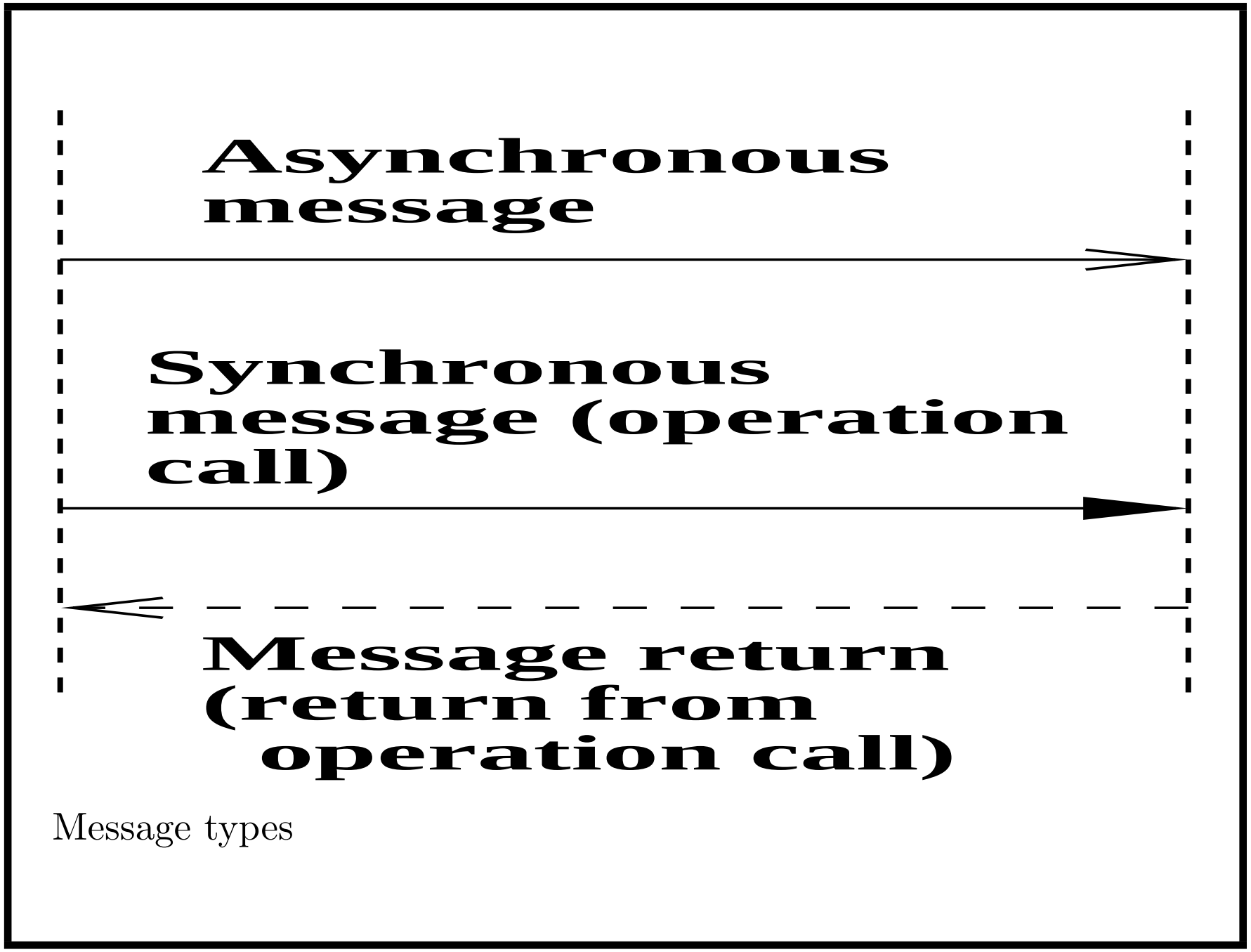
Consist of:

- *Object lifelines* – vertical dashed lines.
- Vertical rectangles indicate activity of object (usually an execution of an operation on object), duration of activity.
- Arrows from one lifeline to another represent messages, eg., operation invocations or signals.

Show scenarios of system behaviour + object interactions.



Example sequence diagram

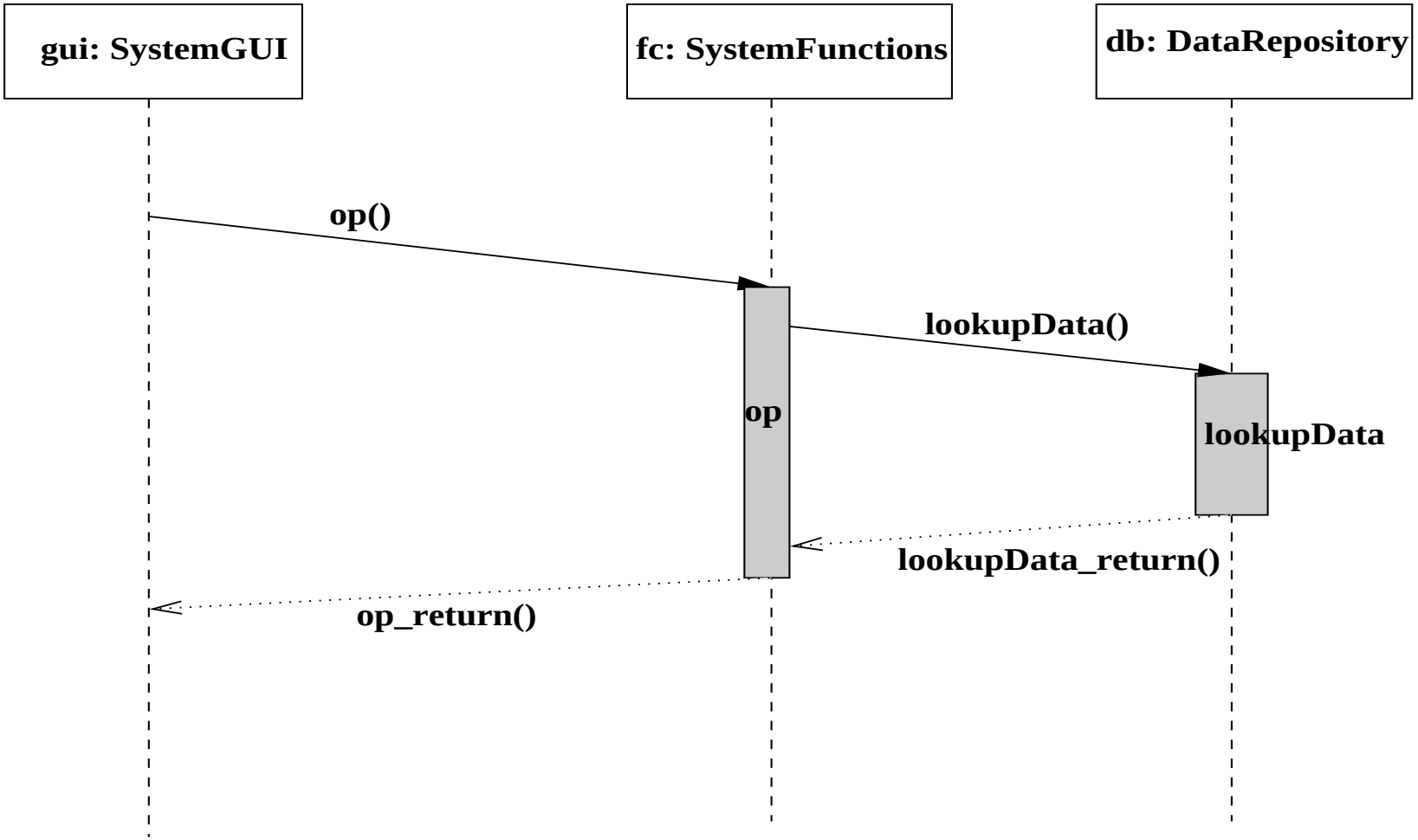


Message types

Sequence diagrams

- Interactions describe intended scenarios of behaviour of system, different cases of execution of use cases.
- In requirements analysis, show variations of behaviours of a use case.

Figure shows typical interaction for use case which queries some data and produces result to GUI.



Use case interaction model

Activities

- Graphical description of program control flow + processing blocks
- Can show sequencing, choice, iteration, parallel execution (multiple concurrent threads)
- Any well-structured activity can be expressed instead as a state machine, but activity notation is simpler.

Activity Diagrams

Visually describe behaviour composed of collections of actions (eg., algorithms of operations, or workflows of business processes).

Activity: specification of behaviour as coordinated sequencing of subordinate units whose individual elements are actions.

Action: represents single step within an activity, that is, one that is not further decomposed. An action may be complex in its effect and not atomic.

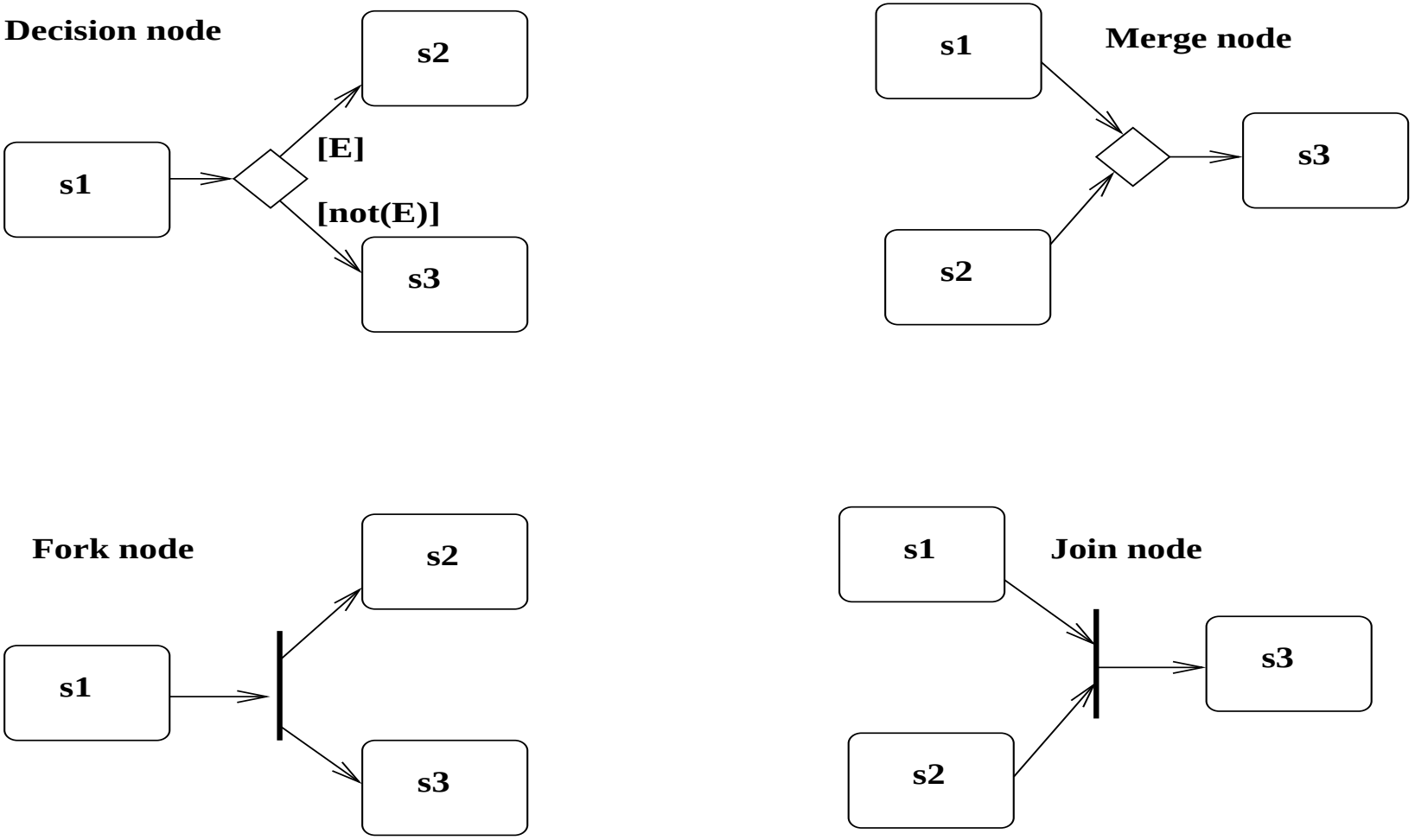
Activities are generalisation of programming constructs such as sequencing, conditionals and loops.

Activity diagrams

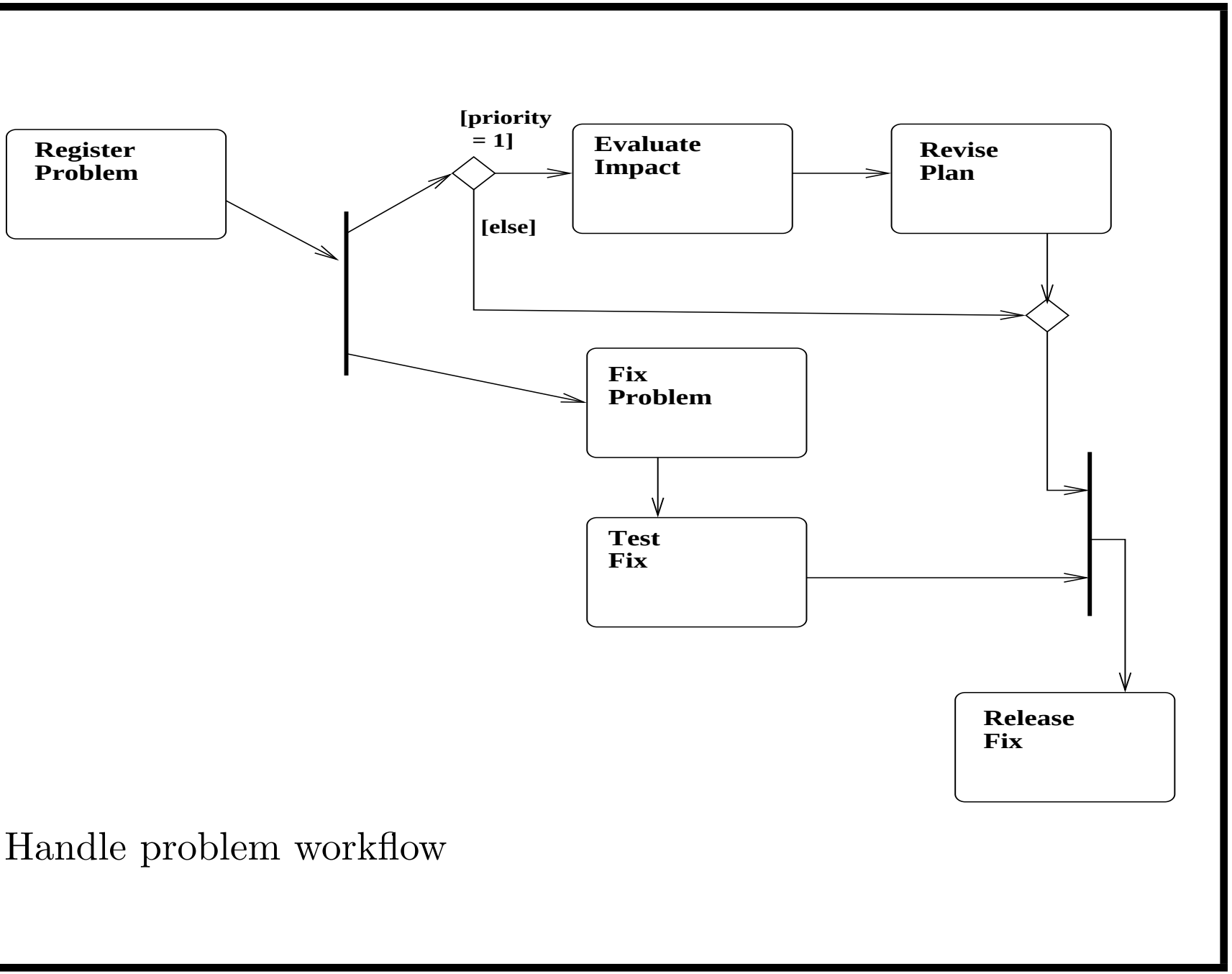
Consist of:

- Actions (state boxes)
- Arrowed lines denote control flows (sequencing of actions)
- choice points branching and joining (diamonds)
- parallel flows (starting and ending at vertical bars).

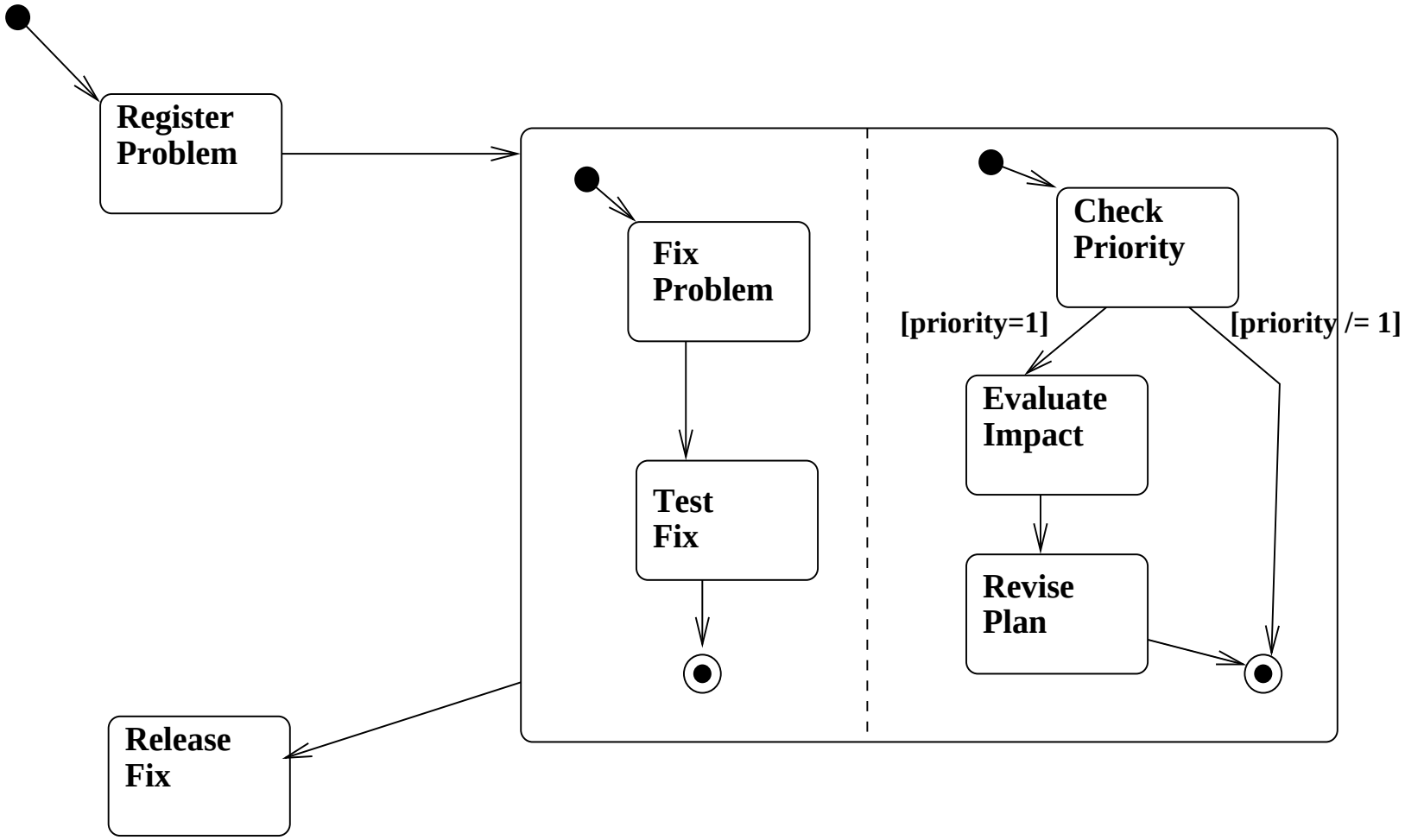
Parallel flows mean separate threads of control.



Activity diagram notations



Handle problem workflow



Handle problem workflow, as behaviour state machine

Activities

Structured activities can also be expressed as pseudocode:

- if E then $S1$ else $S2$
- while E do S
- $S1$; $S2$
- for $x : s$ do S
- $S1$ parallel $S2$
- execute E

Handle problem workflow as statement

```
registerProblem() ;
( ( if priority = 1
    then evaluateImpact() ; revisePlan()
    else skip ) parallel
   ( fixProblem() ; testFix() ) ) ;
releaseFix()
```

More direct to derive code from activity diagrams, compared to from state machines.

Use case activity example: addCustomerAccount

```
addCustomerAccount(cId: String, aId: String)
activity:
  if cId /: Customer->collect(customerId)
  then
    "No customer with cId"->display()
  else if aId /: Account->collect(accountId)
  then
    "No account with aId"->display()
  else
  ( c : Customer := Customer[cId] ;
    a : Account := Account[aId] ;
    if a : c.accounts
    then "Account already in customer's accounts"->display()
    else
      execute a : c.accounts )
```

Operation activity example: gcd

For gcd, there is well-known recursive computation:

```
static query gcd(x: int, y: int): int
pre: x >= 0 & y >= 0
post:
  (x = 0  =>  result = y) &
  (y = 0  =>  result = x) &
  (x = y  =>  result = x) &
  (x < y  =>  result = gcd(x, y mod x))  &
  (y < x  =>  result = gcd(x mod y, y))
```

But inefficient for large x, y. Instead, define explicit algorithm:

```
static query gcd(x: int, y: int): int
pre: x >= 0 & y >= 0
post: true
activity:
  l : int ; k : int ; l := x  ; k := y  ;
```

```
while l /= 0 & k /= 0 & l /= k
do
    if l < k then k := k mod l
    else l := l mod k ;
if l = 0 then result := k
else result := l ;
return result
```

Language-independent, can be directly mapped to C, Java, etc.

General specification guidelines

- Define system by its declarative specification (eg., class invariants, operation + use case pre/postconditions).
- Make this as complete and thorough as possible.
- Make specification as clear + efficient as possible.
- Operation activities can be derived from declarative constraints.

Primary difference between specification models and design models is that specifications specify *what* system should do, and designs define *how* system carries out its functions.

Summary of Parts 2 and 3

UML notations provide comprehensive notations to define specifications:

- Use cases describe functional requirements
- Class diagrams define data of system
- OCL allows expression of detailed functionality and data properties
- State machines + activities can precisely describe behaviour
- Sequence diagrams can give examples of behaviour.

Coursework: Design

- Write pseudocode activities for use cases + operations, defining steps of processing.
- Eg.:

```
query payout() : double
post:
  result = payments->collect(amount)->sum()
```