

## Activity diagrams & State machines

Alexander Serebrenik



**TU** / **e**

Technische Universiteit  
**Eindhoven**  
University of Technology

Where innovation starts

# This week sources



## OMG Unified Modeling Language™ (OMG UML)

*Version 2.5*

Slides by



David Meredith,  
Aalborg University, DK

Site by



Kirill Fakhroutdinov  
GE Healthcare, USA

# Before we start...

## True or False?

1. A web server can be an actor in a use case diagram.
2. *Guarantee* is an action that initiates the use case.
3. Use case “Assign seat” includes the use case “Assign window seat”.
4. Generalization is represented by an arrow with a hollow triangle head.
5. Every use case might involve only one actor.

# Before we start...

T, unless it is a part of the system you want to model

## True or False?

1. A web server can be an actor in a use case diagram.

Guarantee is a postcondition.  
An action that initiates the use case is called “trigger”.

2. *Guarantee* is an action that initiates the use case.

3. Use case “Assign seat” includes the use case “Assign window seat”.

4. Generalization is represented by an arrow with a hollow triangle head.

5. Every use case might involve only one actor.

# Before we start...

## True or False?

1. A web server can be an actor in a use case diagram.
2. No, the correct relation here is extension (<<extend>>); <<include>> suggests that “Assign window seat” is always called whenever “Assign seat” is executed.
3. Use case “Assign seat” includes the use case “Assign window seat”.
4. Generalization is represented by an arrow with a hollow triangle head. Yes
5. Every use case might involve only one actor. No, why?

# Phone call use case

## Place a call

**Pre-condition** The telephone set is connected to the telephone line “A”, it is on-hook and there is no incoming call (it is not ringing).

**Trigger** The user picks up the telephone hook connected of the telephone set (connected to line “A”) and dials number “B”.

**Guarantee** A communication between “A” and “B” will commence.

**Main Scenario**

- (a) The user picks up the telephone hook connected to the telephone line “A”.
- (b) If the line is free, the user receives a dial tone sent by the line .
- (c) The user dials number “B”.
- (d) The call request is forwarded to the switch center.
- (e) If line “B” is not busy, the call request is forwarded to “B” and a tone is sent to “A”.
- (f) “B”'s telephone rings.
- (g) If somebody at “B” picks up the hook, the ringing tone at “A” is stopped and a telephone connection will commence.

## Alternatives

- (b-1).1 If the telephone line is engaged in a conversation, the user will be connected to the same conversation.
- (b-2).1 If the user does not dial a number for a certain amount of time, a permanent tone is emitted by the switch center, no further call will be accepted and the user has to replace the hook.
- (e-1).1 If line “B” is busy, and “B” does not have call waiting the user at “A” will receive a busy tone.
- (e-2).1 If line “B” is busy, and “B” has call waiting the user at “A” will receive a call-waiting tone from the switch center. When line “B” becomes free, sub-scenario (e-g) follows.
- (g-1).1 If nobody picks the call at “B”, after a certain amount of time and a telephone set at “B” has its answering machine enabled, then the *Record Message* scenario at “B” will follow.
- (g-2).1 If nobody picks the call at “B”, after a certain amount of time and no telephone set at “B” has its answering machine enabled, then the user at “A” will receive a no response tone from the switch center (via the line).

# Phone call use case

## Place a call

Pre-condition The telephone set is connected to the telephone line “A”, it is on-hook and there is no incoming call (it is not ringing).

Trigger The user picks up the telephone hook connected of the telephone set (connected to line “A”) and dials number “B”.

Guarantee A call is established.

Main Scenario

- (a)
- (b)
- (c)
- (d)
- (e)
- (f)
- (g)

- ### Disadvantages
1. To understand alternatives, one has to read them **simultaneously** with the main scenario.
  2. **Missing** alternatives are difficult to spot.
  3. Main scenario enforces sequential execution, missing potential for concurrent execution.

Alternatives

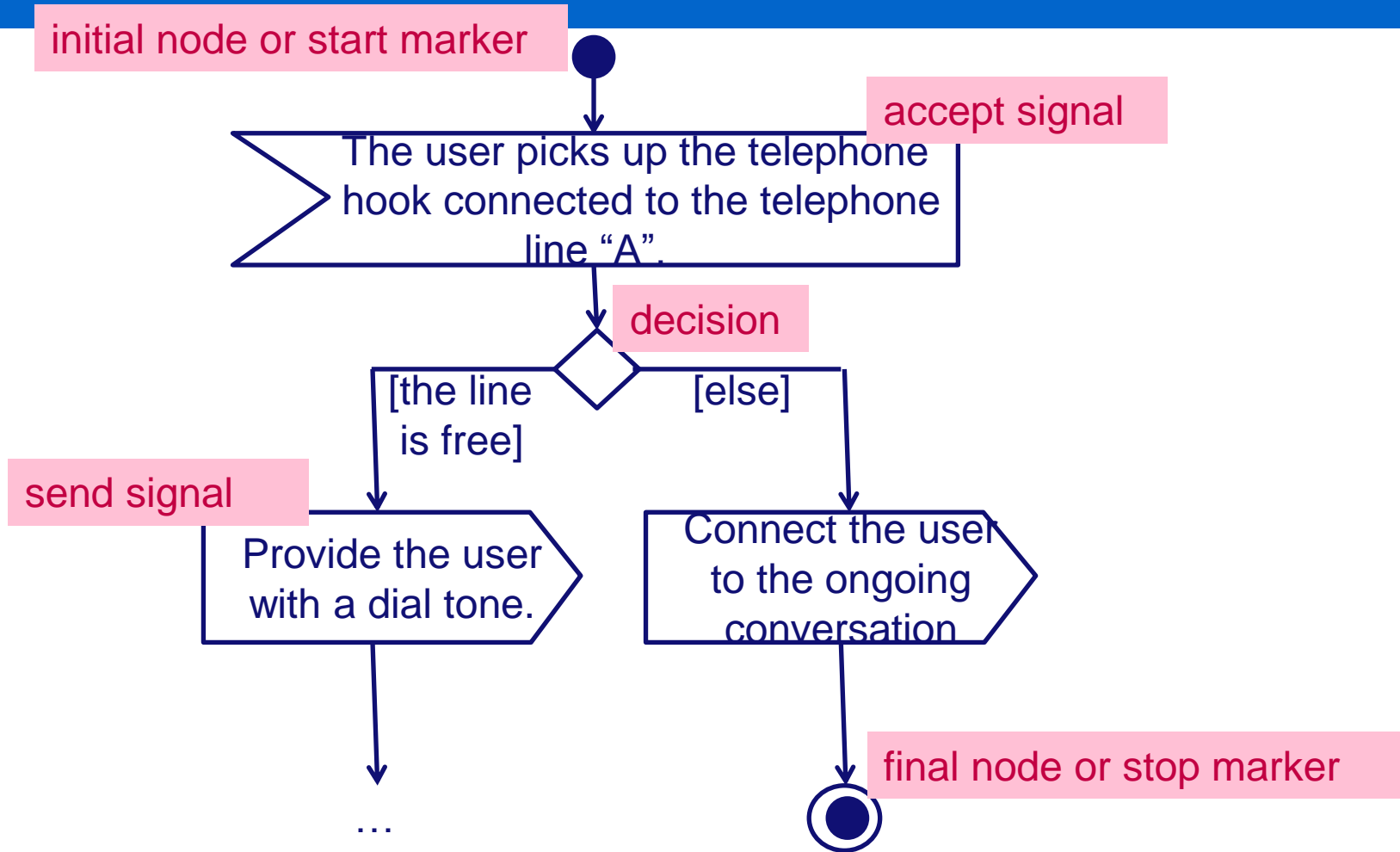
- (b-1).1
- (b-2).1
- (e-1).1
- (e-2).1
- (g-1).1
- (g-2).1

If line “B” is busy, and “B” has call waiting the user at “A” will receive a call-waiting tone from the switch center. When line “B” becomes free, sub-scenario (e-g) follows.

If nobody picks the call at “B”, after a certain amount of time and a telephone set at “B” has its answering machine enabled, then the *Record Message* scenario at “B” will follow.

If nobody picks the call at “B”, after a certain amount of time and no telephone set at “B” has its answering machine enabled, then the user at “A” will receive a no response tone from the switch center (via the line).

# Activity diagram: Let us start

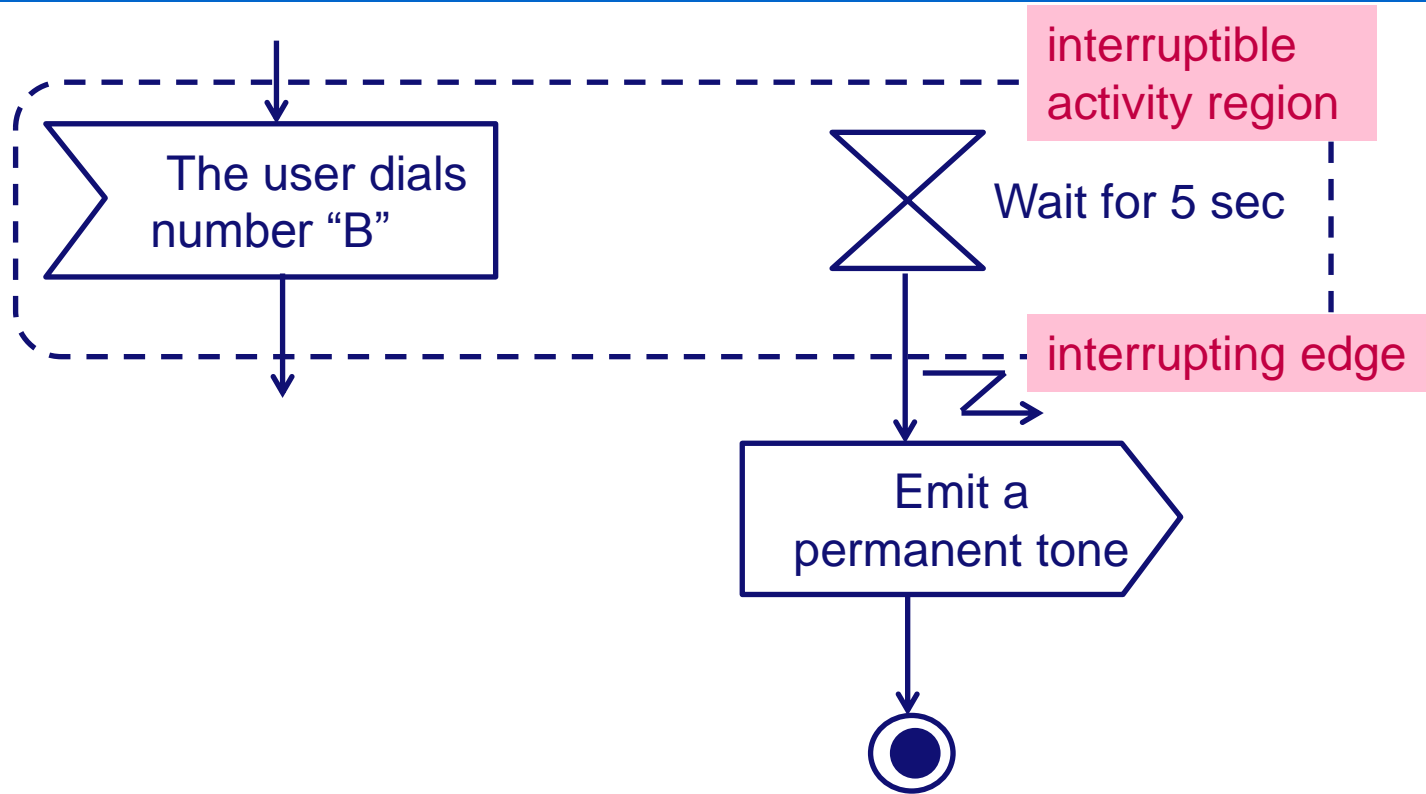




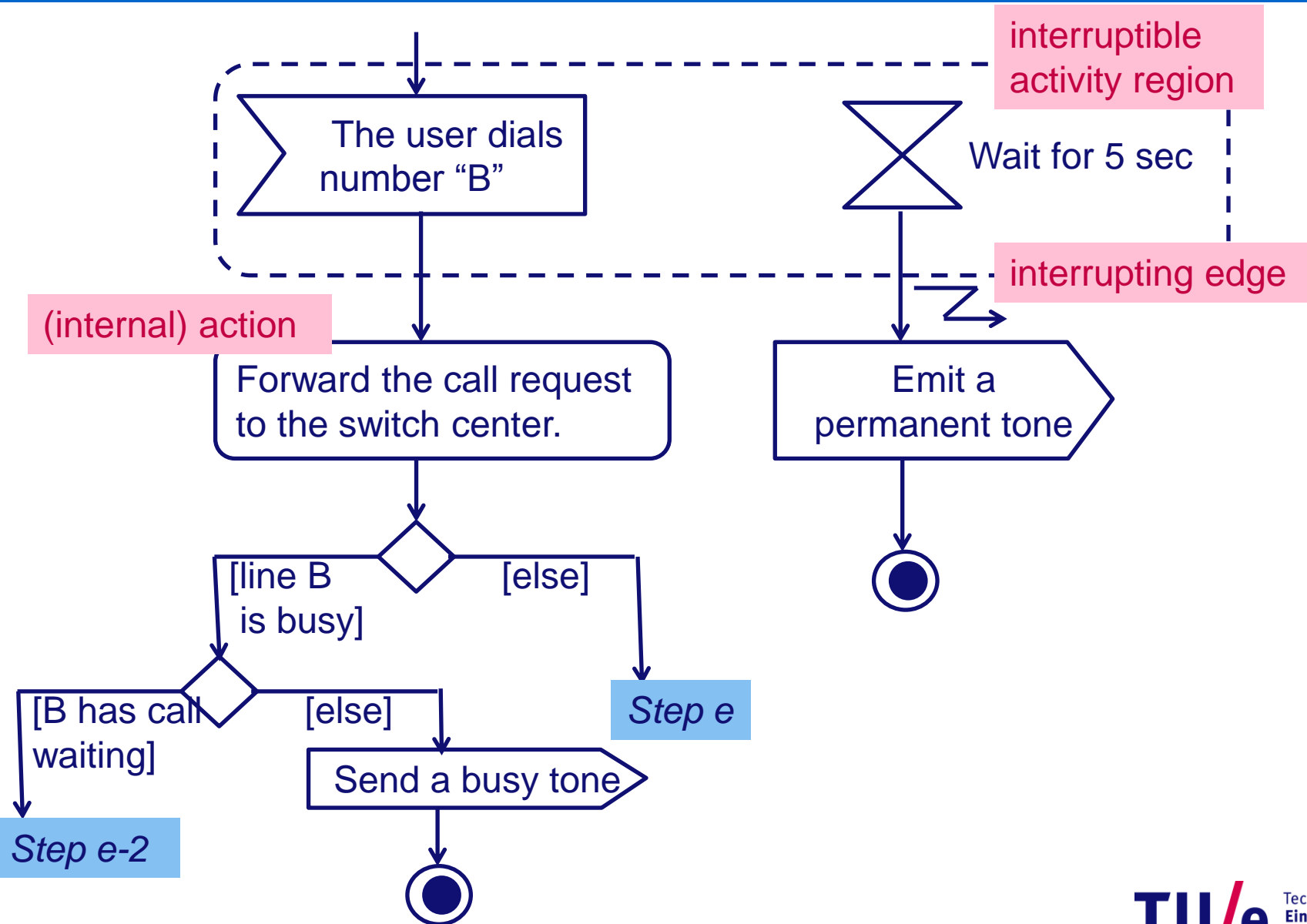
# The next step

- **Main scenario:** The user dials number “B”.
- **Alternative step:** If the user does not dial a number for a *certain amount of time*, a permanent tone is emitted by the switch center, no further call will be accepted and the user has to replace the hook.
- Two processes:
  - wait for the user to dial a number (“main scenario”)
  - wait for *certain amount of time*, say 5 sec;
- When 5 seconds passed, abort the “main scenario” waiting

# Modeling timeout

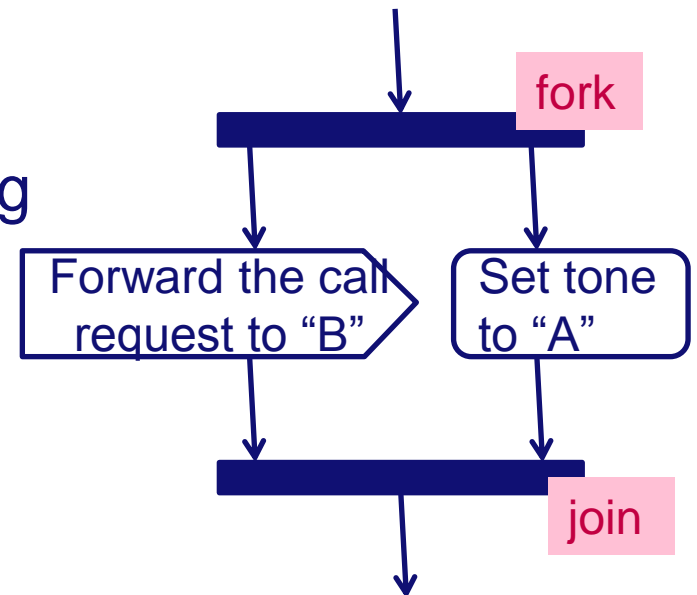


# Continuing...



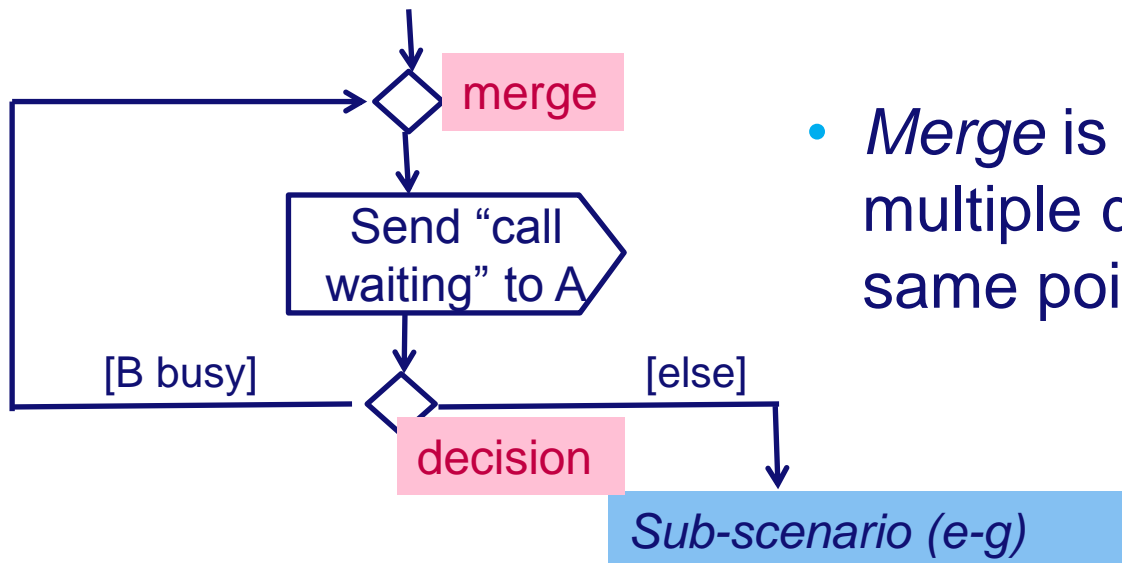
# Step e

- If line “B” is not busy, the call request is forwarded to “B” **and** a tone is sent to “A”.
  - No indication whether the call request should first be forwarded, and the tone sent next, or vice versa.
  - We should not disallow any of these options.
- Execute both in parallel
- Wait till both call request forwarding and tone setting are successful.



## Step e-2

- The user at “A” will receive a “call waiting” tone from the switch center. When the line “B” becomes free, sub-scenario (e-g) follows.

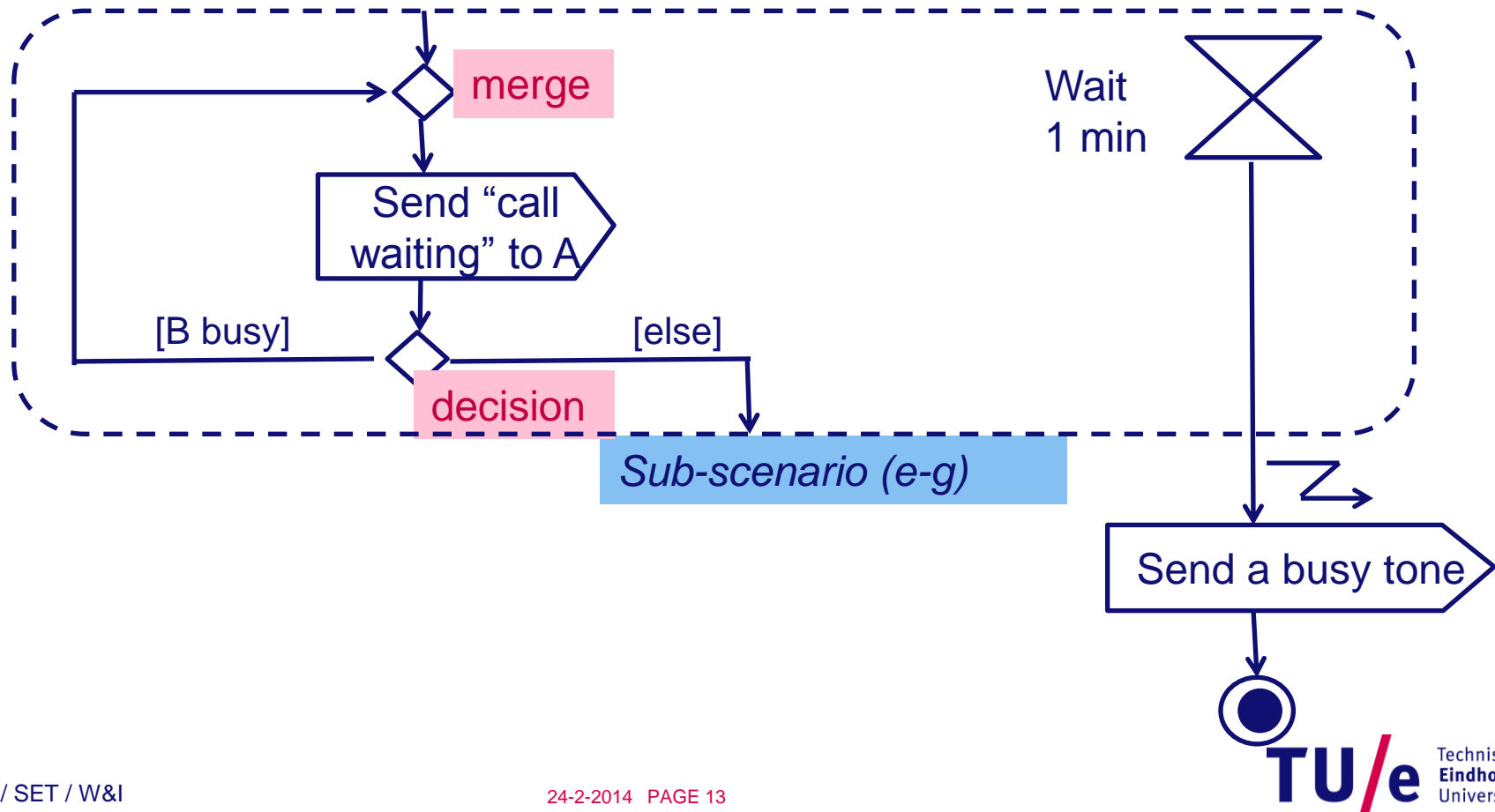


- Merge* is needed to allow multiple data flows in the same point.

- What would happen if B stays busy all the time?
  - Omission in the use case.

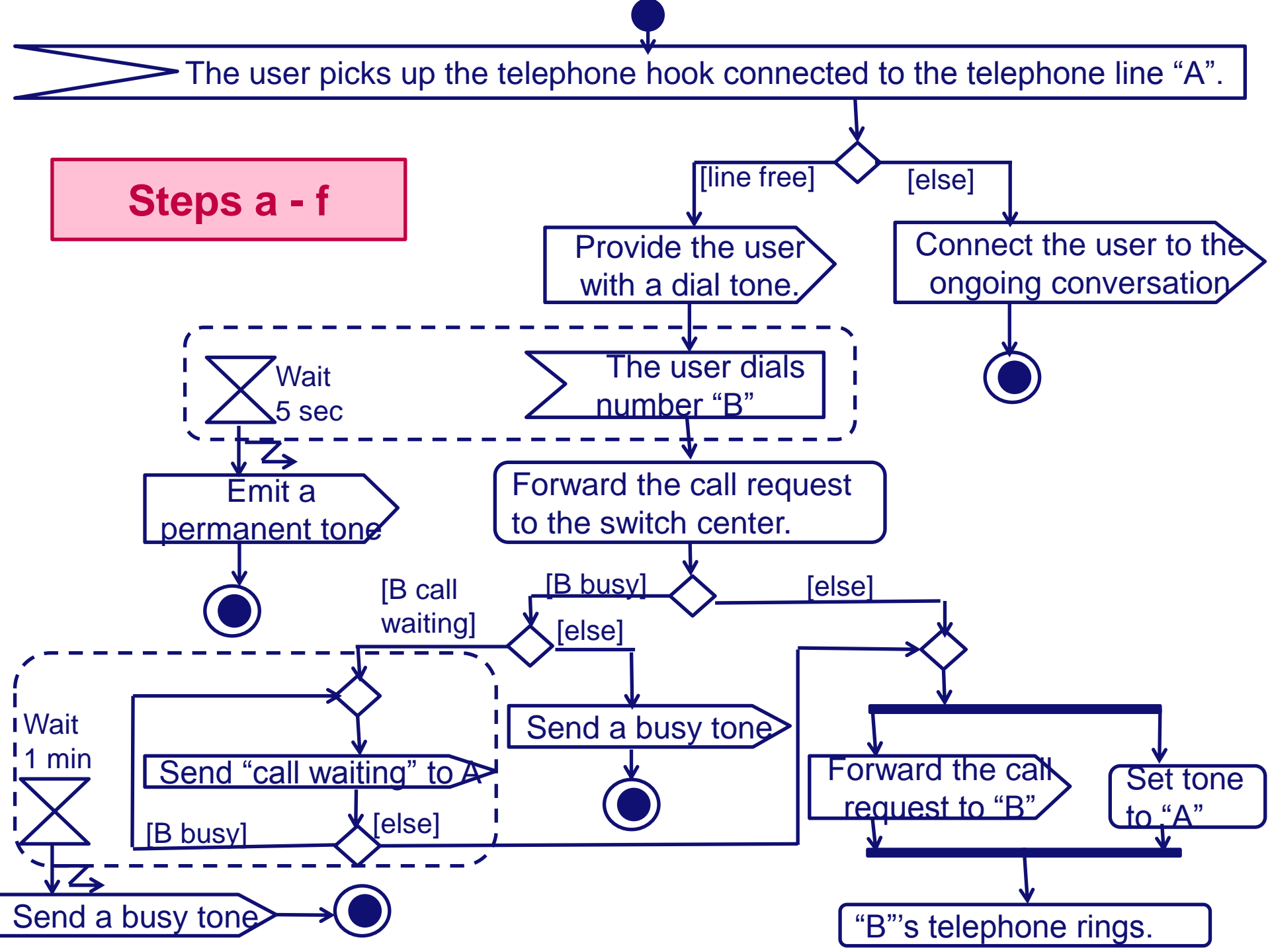
# Step e-2

- The user at “A” will receive a “call waiting” tone from the switch center. When the line “B” becomes free, sub-scenario (e-g) follows.









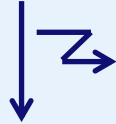



The user picks up the telephone hook connected to the telephone line "A".

**Steps a - f**



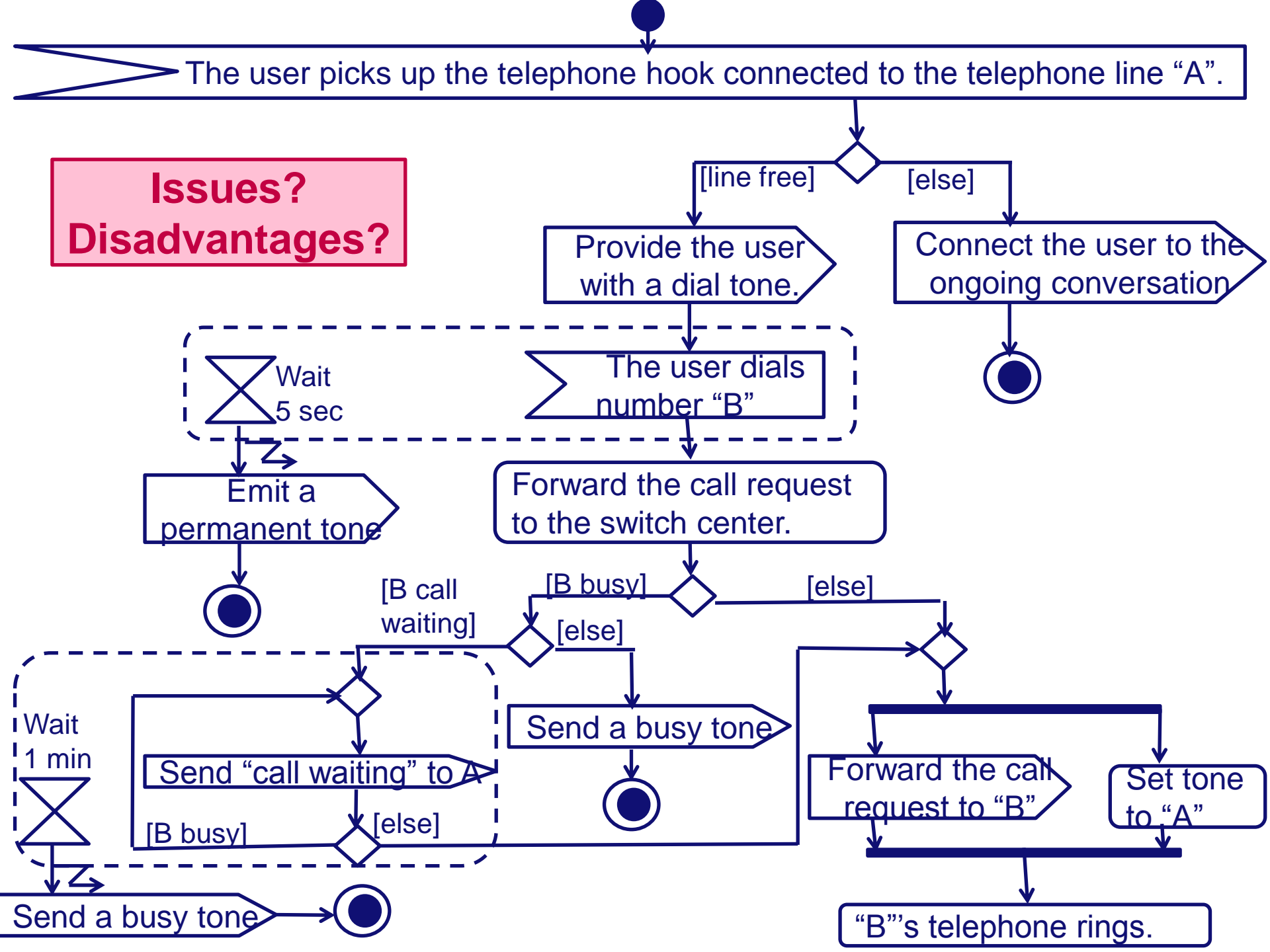
# Summary of activity diagram elements (so far)

	Graphical representation	Description	“Keywords”
Action			
Control flow	 	start / stop markers	
		decision	if
		fork / join	loops, end-if and, parallel
Signals	  	incoming (accept), outgoing (send)	user
		time-based	when, time
Interrupts	 	interruptible activity region, interrupting edge	cancel, interrupt
		an alternative notation for an interrupting edge	



The user picks up the telephone hook connected to the telephone line "A".

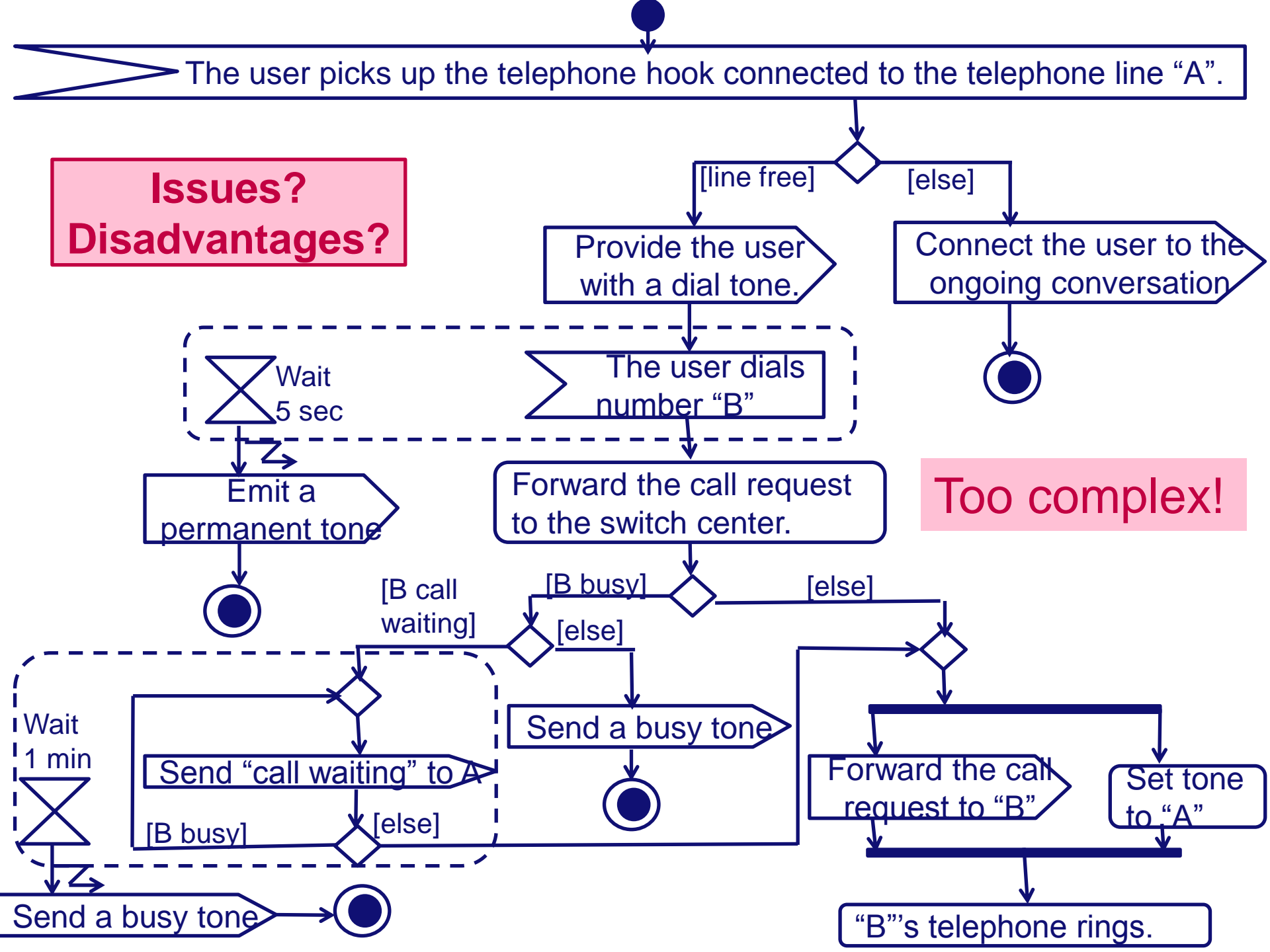
**Issues?**  
**Disadvantages?**



The user picks up the telephone hook connected to the telephone line "A".

**Issues?**  
**Disadvantages?**

**Too complex!**



# The diagram became quite complex

- Reorganize **links** to reduce the number of self-crossings, links between different cases of the activity
  - Fosters reuse similarly to *labels*
- Reorganize the activity using **subactivities**
  - Fosters reuse similarly to *subroutines*

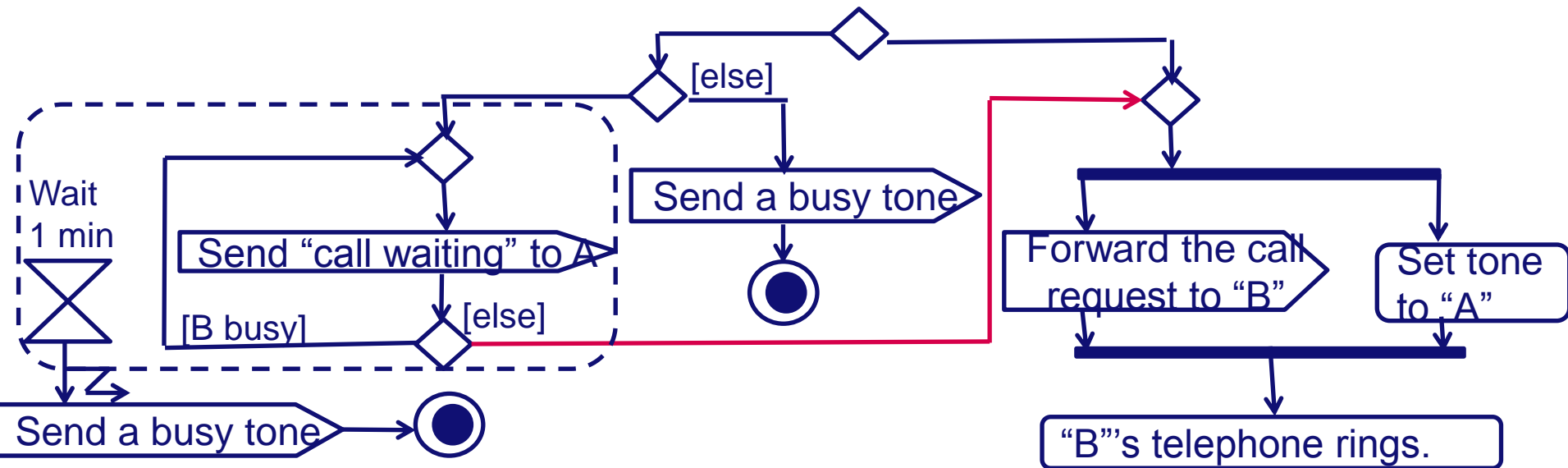
# Reorganize links

- Use **connectors** instead of uninterrupted lines



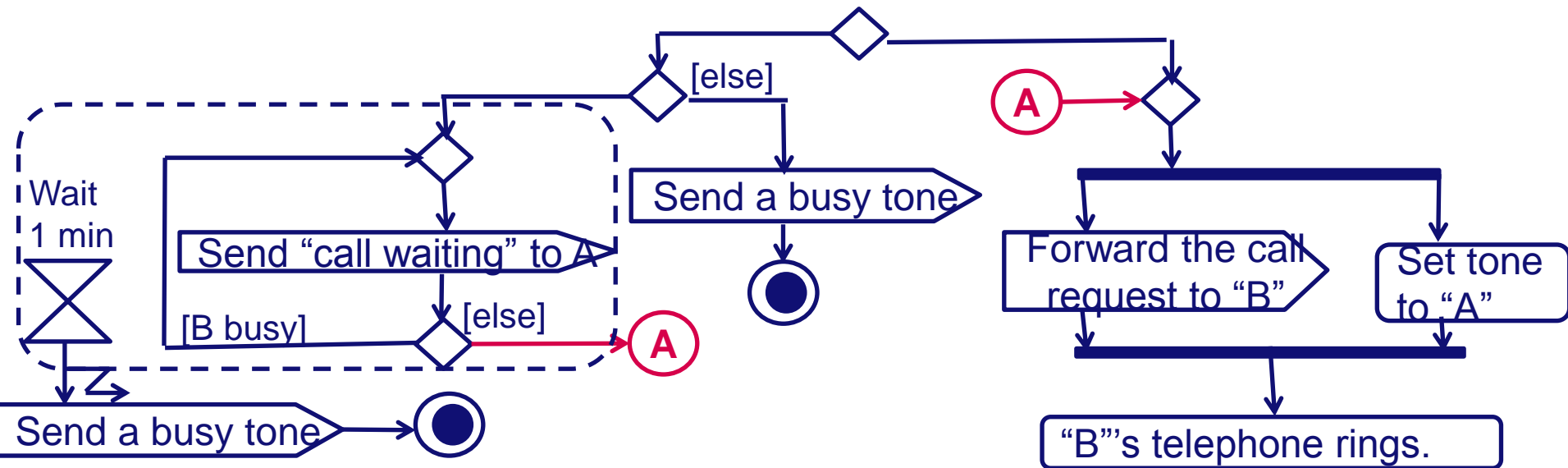
# Reorganize links

- Use **connectors** instead of uninterrupted lines



# Reorganize links

- Use **connectors** instead of uninterrupted lines

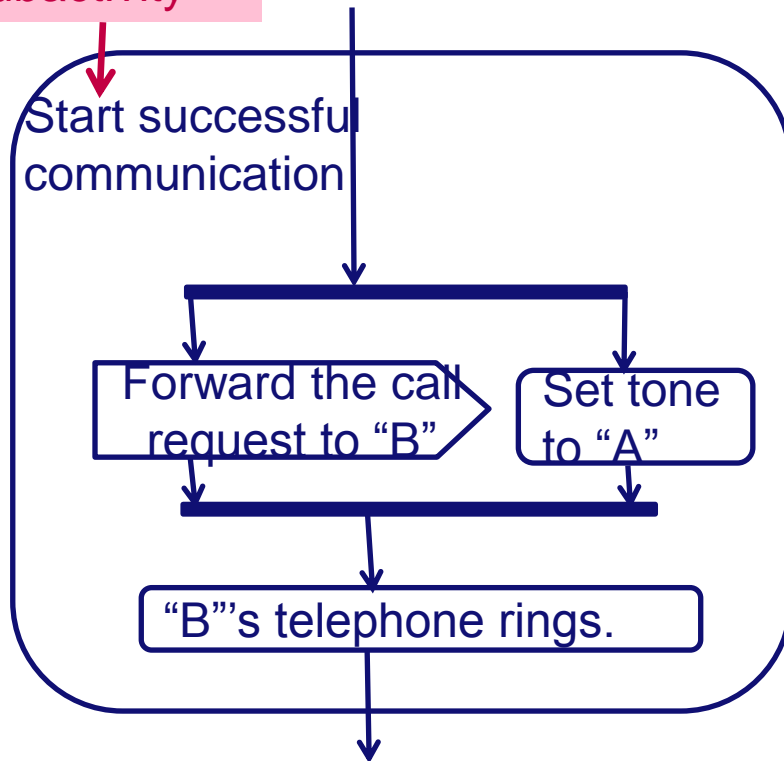


- Beware: **Too many** connectors hinder comprehension!

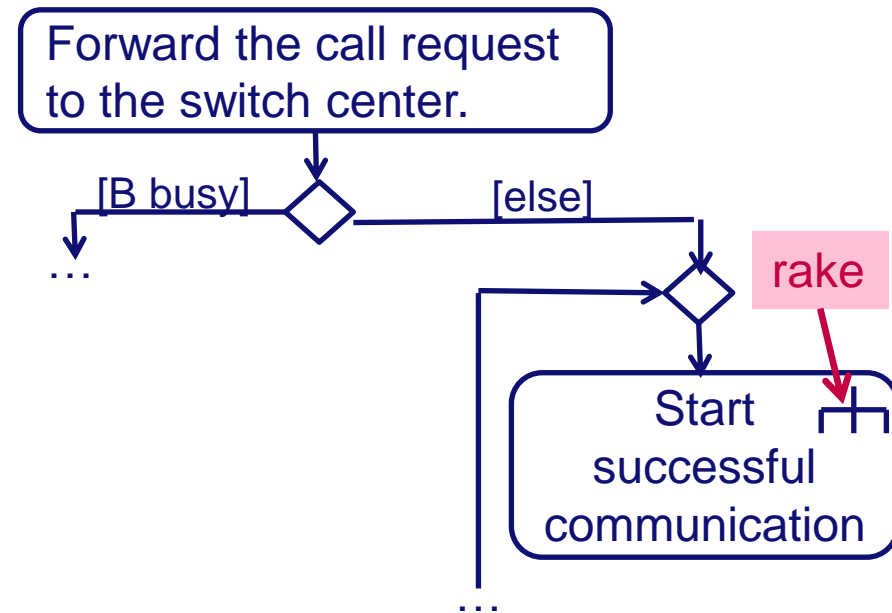
# Reorganize the activity using subactivities

- Use **subactivities** to reuse or simplify the structure

name of the subactivity

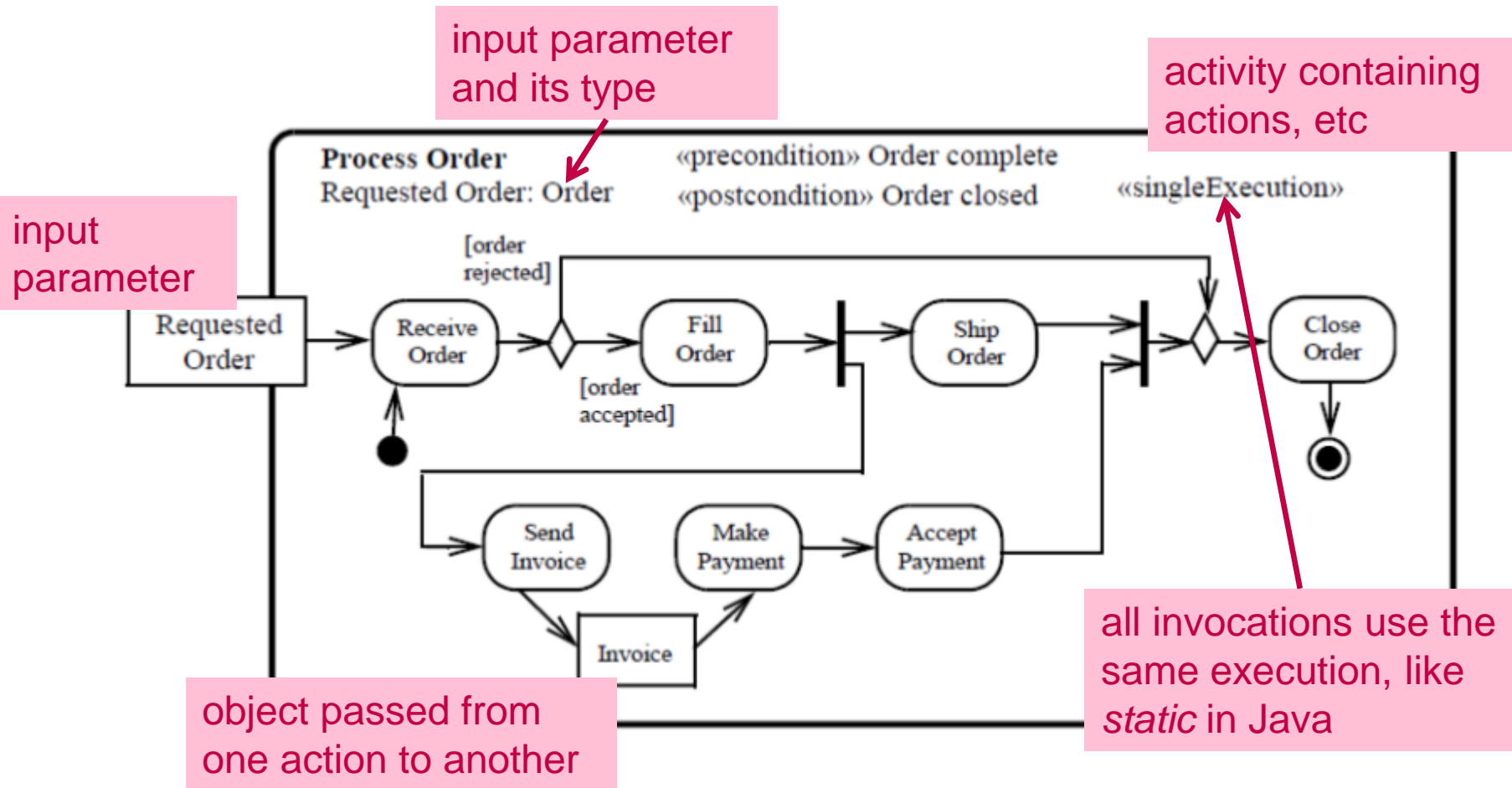


*Callee*



*Caller*

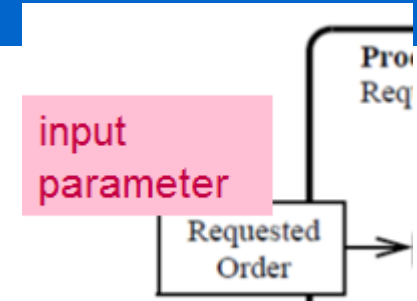
# Example of a subactivity: processing an order





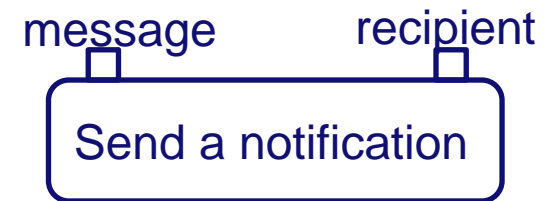
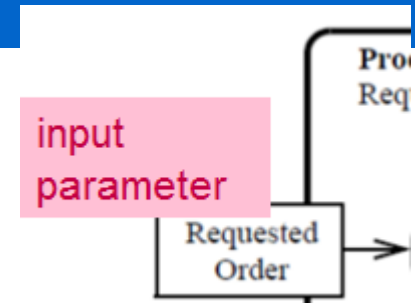
# Parameters

- Activities can have parameters,
  - what about individual **actions**?



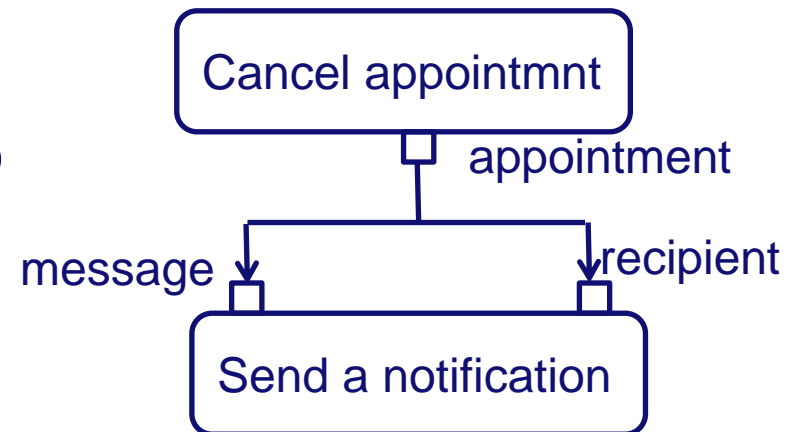
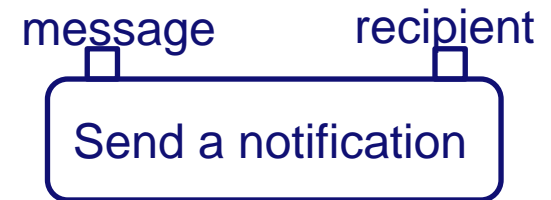
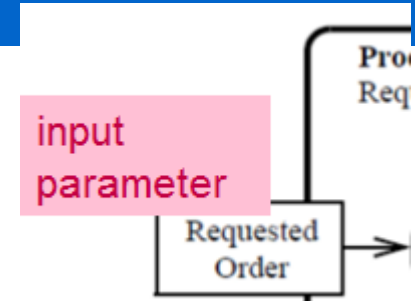
# Parameters

- Activities can have parameters,
  - what about individual **actions**?
- **Pins** represent action parameters
  - Optional: use only for data produced and used



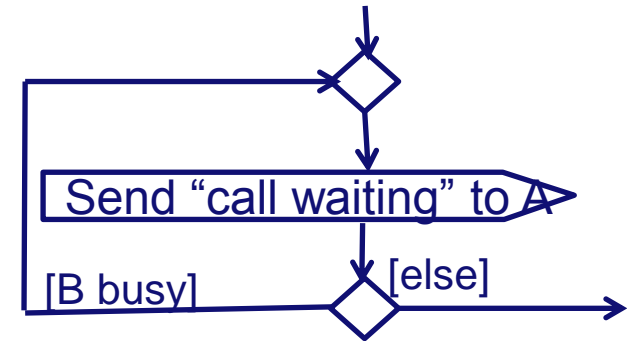
# Parameters

- Activities can have parameters,
  - what about individual **actions**?
- **Pins** represent action parameters
  - Optional: use only for data produced and used
- Data can be also produced by **transforming** “output pin” data to “input pin” data



# Pins can also be used for expansion regions

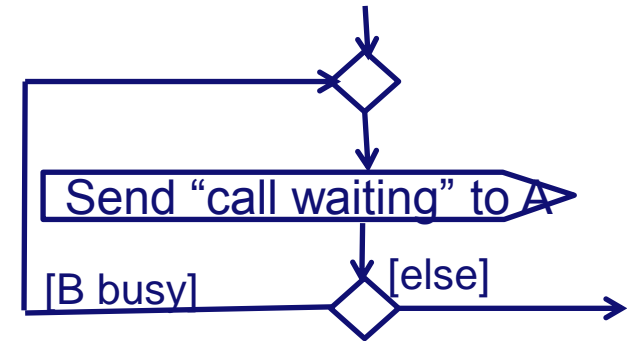
- We've seen an example of a **loop**
- But what if a loop is used to **traverse a collection**?



```
void reviewSubmissions(Collection<Submission> c) {  
    for (Submission s : c)  
        s.review();  
    ...  
}
```

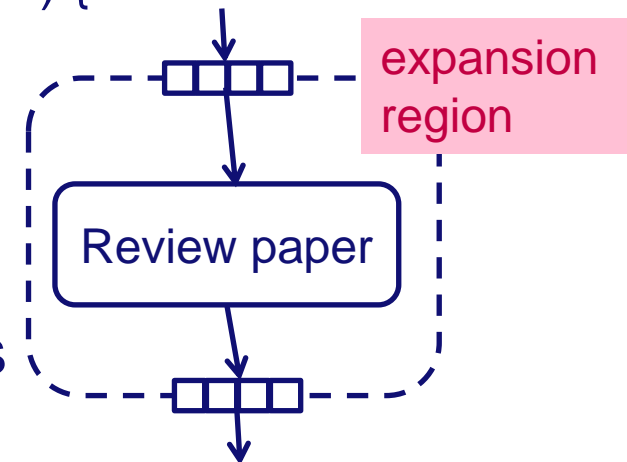
# Pins can also be used for expansion regions

- We've seen an example of a **loop**
- But what if a loop is used to **traverse a collection**?



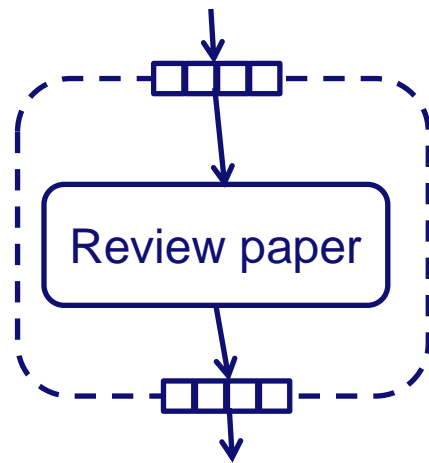
```
void reviewSubmissions(Collection<Submission> c) {  
    for (Submission s : c)  
        s.review();  
    ...  
}
```

- Upper pins: collection of submitted papers
- Lower pins: collection of reviewed papers

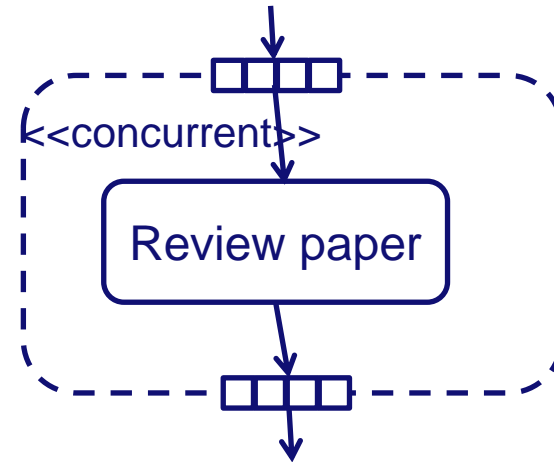


# Expansion regions

- How do we **traverse** the collection?



a) Sequentially

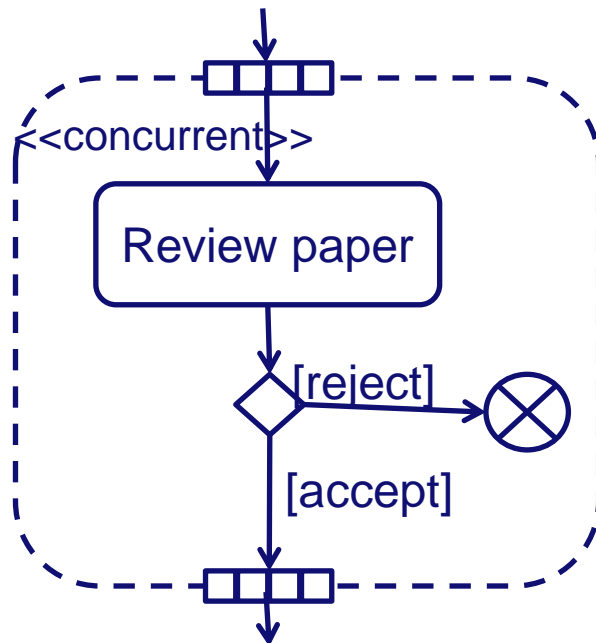


b) Concurrently

- Reviewing scientific papers is done concurrently, so we chose solution b)

# Expansion regions

- Each paper is reviewed and either **rejected** or **accepted**.
  - If paper is rejected no further processing is needed
  - If paper is accepted it is included in the proceedings volume



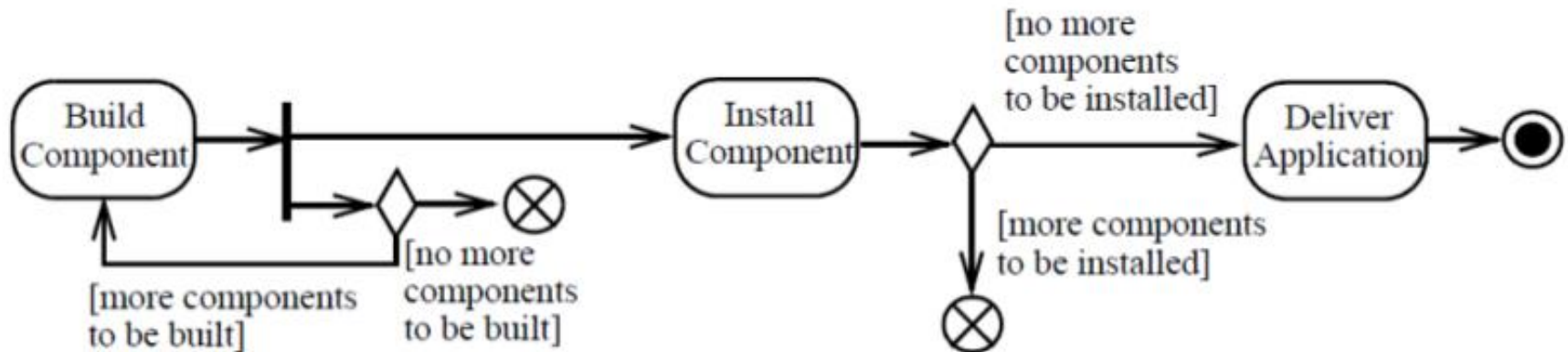
# Activity final vs Flow final



ActivityFinal:  
Termination of the  
entire activity



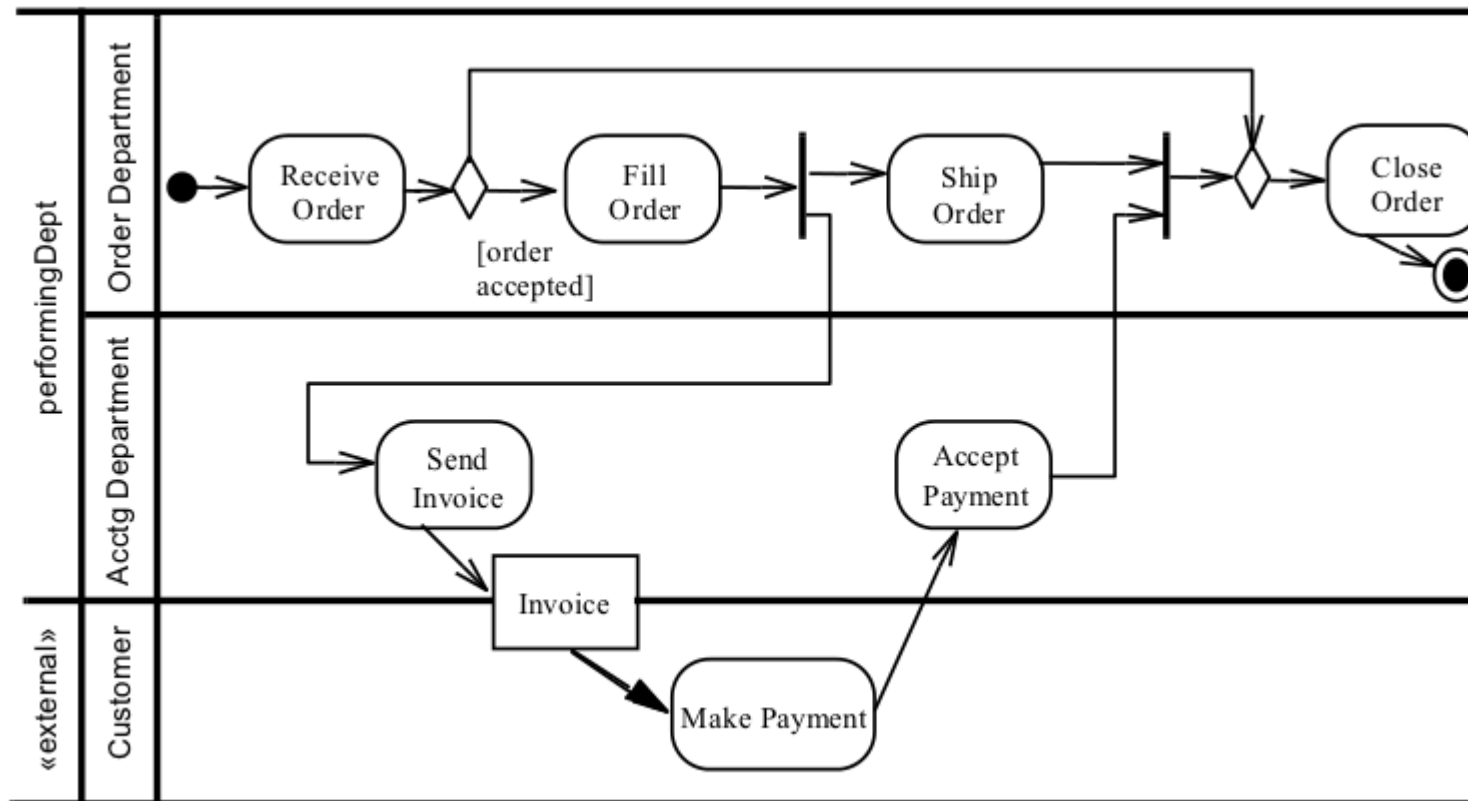
FlowFinal:  
Termination of one of the  
parallel flows, e.g., in the  
expansion region





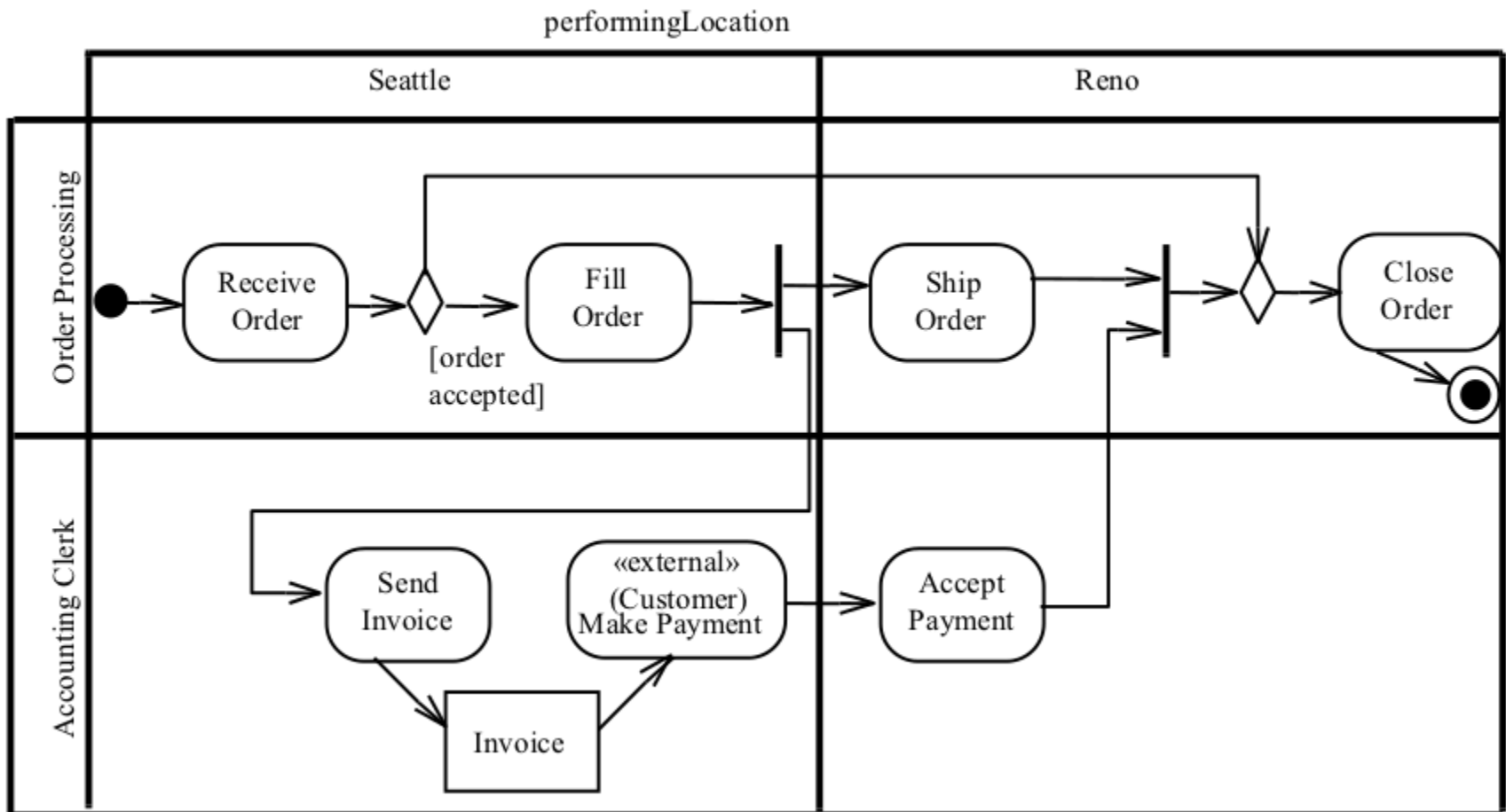
# Still, who is responsible for what?

- Indicate who is **responsible** for each group of activities
- Solution 1a: hierarchical swimlanes (UML 1)



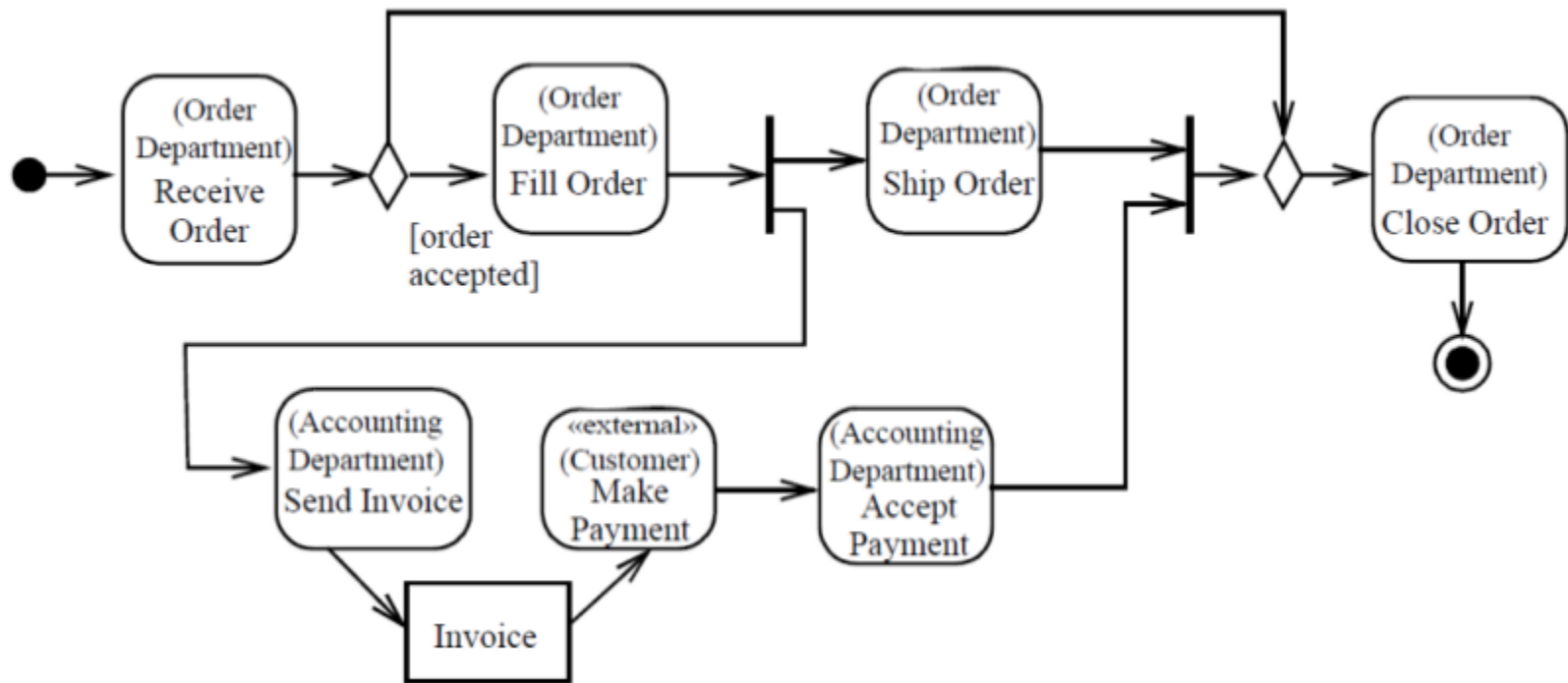
# Partitions

- Indicate who is **responsible** for each group of activities
- Solution 1b: multidimensional swimlanes (UML 2)



# Partitions

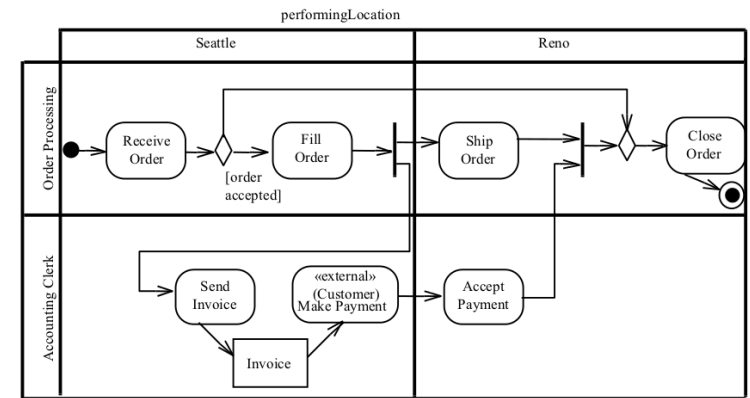
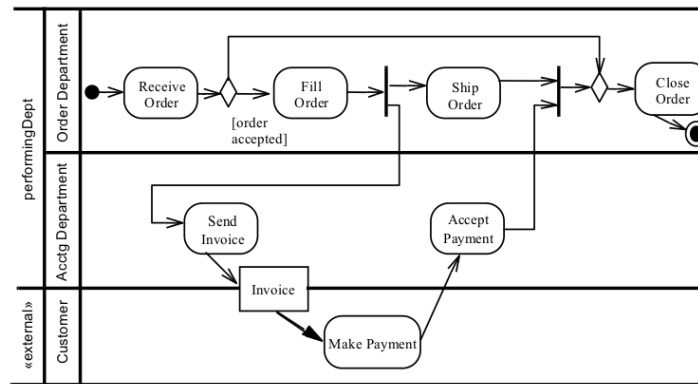
- Indicate who is **responsible** for each group of activities
- Solution 2: annotations (UML 2)



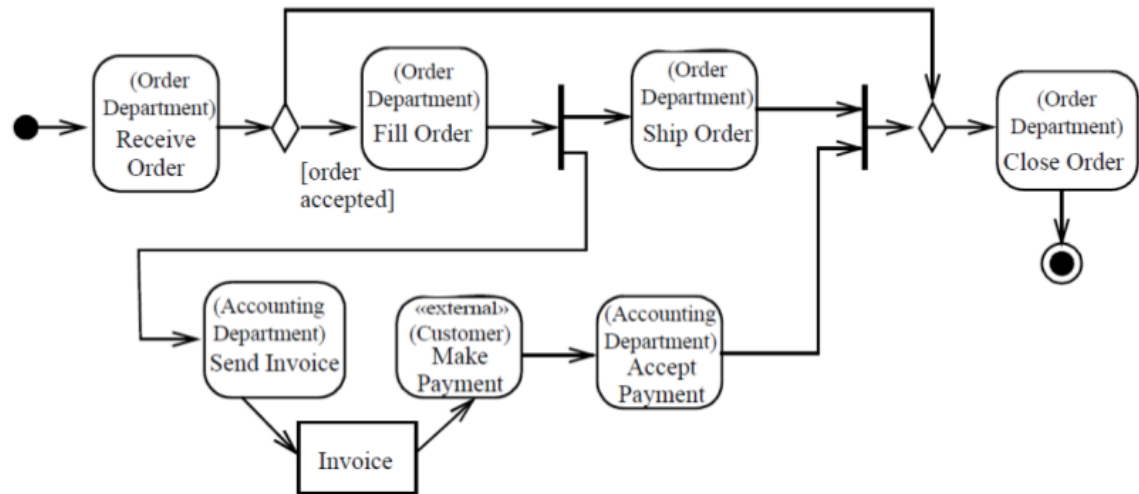
# Partitions

- Indicate who is **responsible** for each group of activities
- Advantages / disadvantages?

## Solution 1: swimlanes



## Solution 2: annotations



# Partitions

- Indicate who is **responsible** for each group of activities
- Advantages / disadvantages?

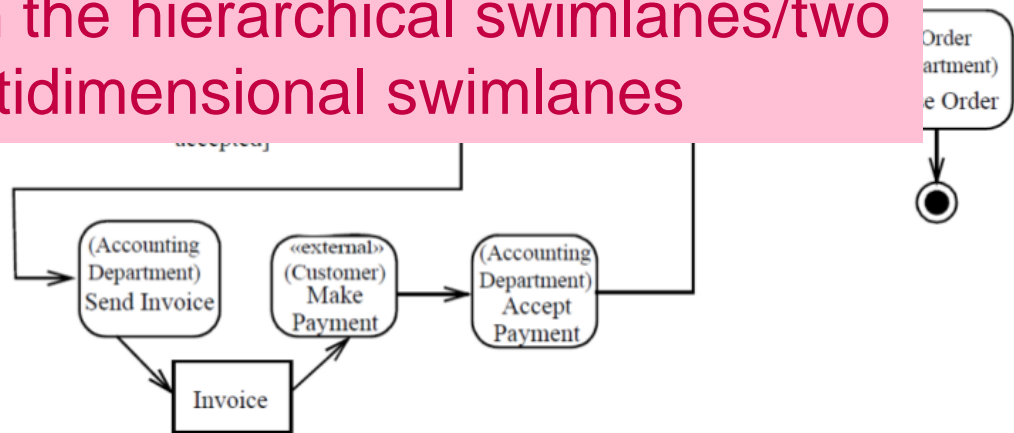
## Solution 1: swimlanes



**Advantage** of swimlanes: easy to identify which activities belong together

**Disadvantage:** we can have only one axis in the hierarchical swimlanes/two in multidimensional swimlanes

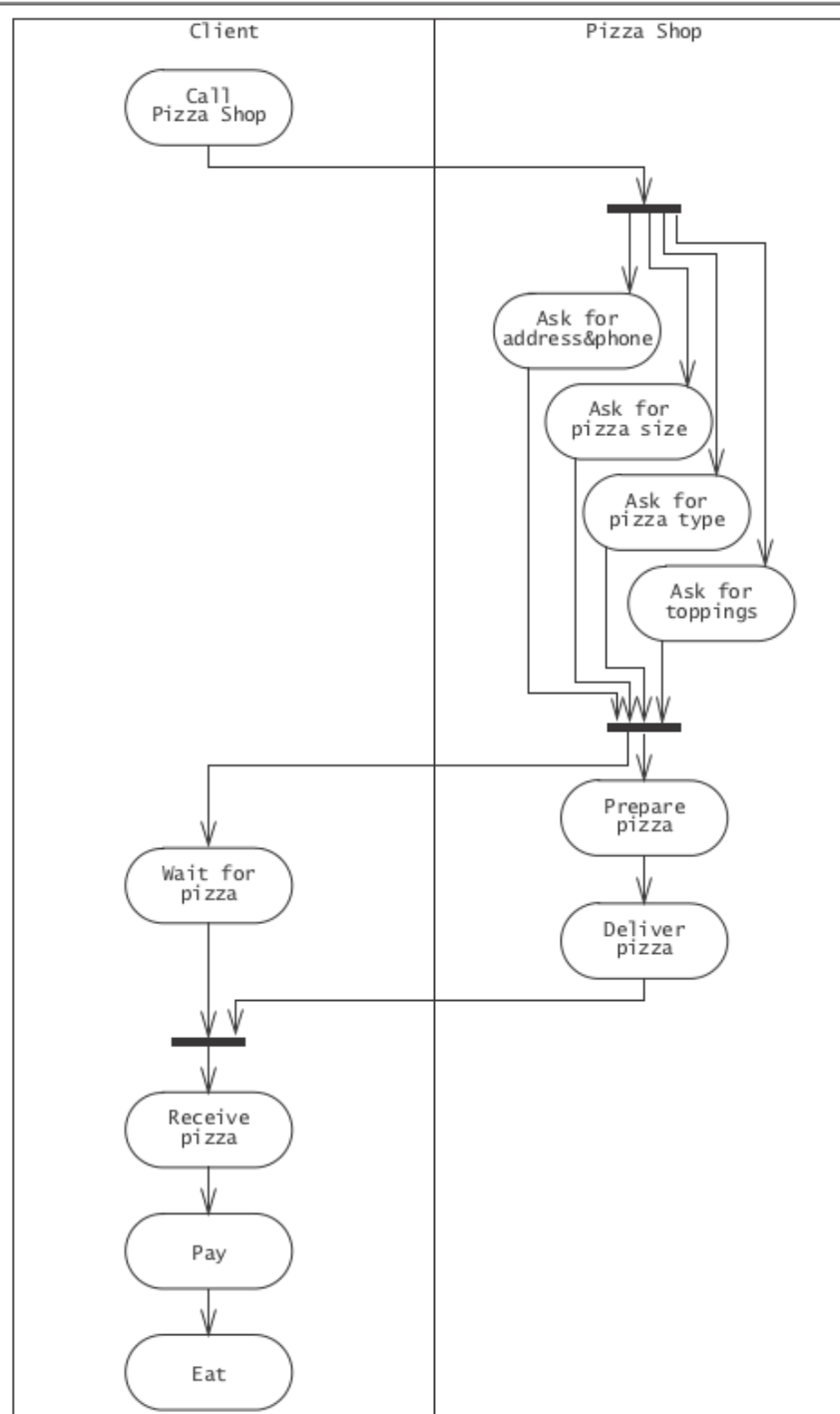
## Solution 2: annotations



# Exercise [*Bernd Bruegge, Allen H. Dutoit*]

- Draw an activity diagram representing each step of the pizza ordering process, from the moment you pick up the phone to the point where you start eating the pizza.
  - Do not represent any exceptions.
  - Include activities that others need to perform.
    - Use an appropriate partition mechanism

# Pizza



# Summary of activity diagram elements

	Graphical representation	Description
Action		action with three inputs
Control flow		start / stop markers
		decision, merge
		fork / join
Signals		incoming (accept), outgoing (send), time-based
Interrupts		interruptible activity region, interrupting edge
Subactivity		activity with input/output parameters, activity invocation
Collection		expansion region



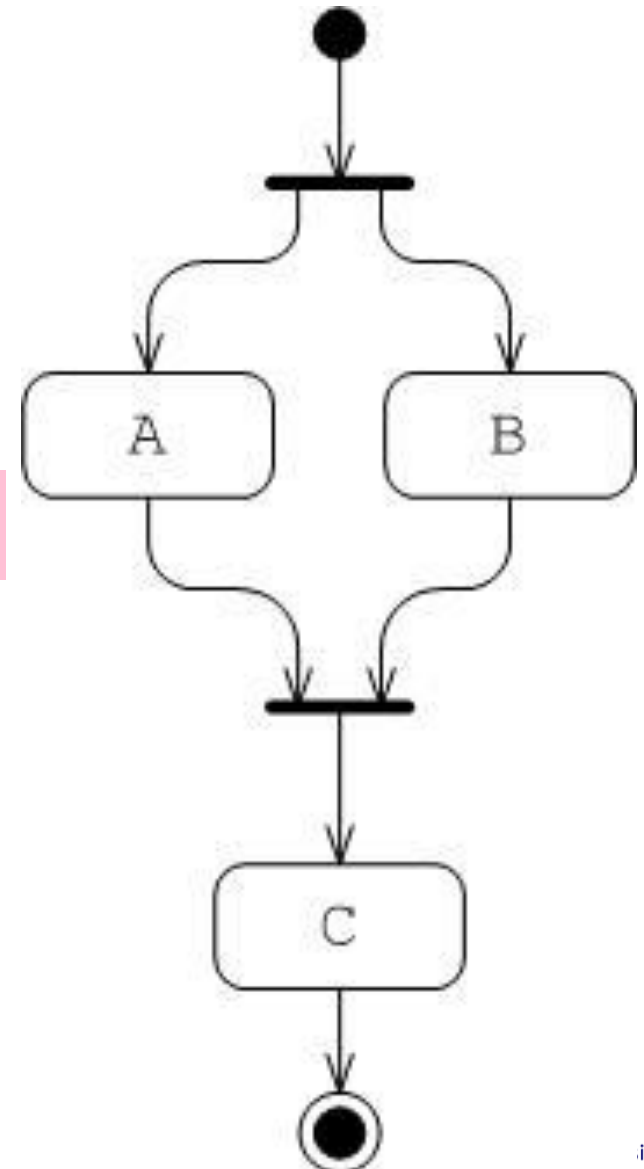
# Which one of the executions is impossible?

a) A,B,C

b) B,A,C

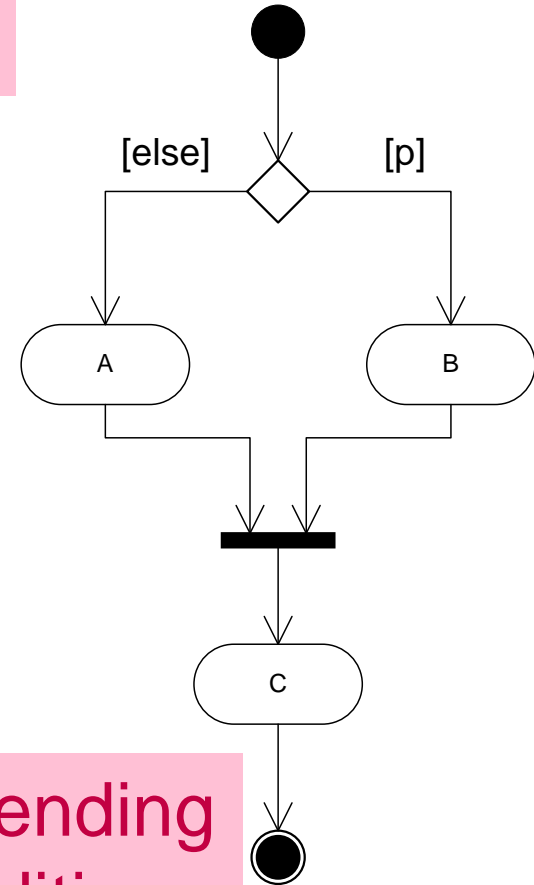
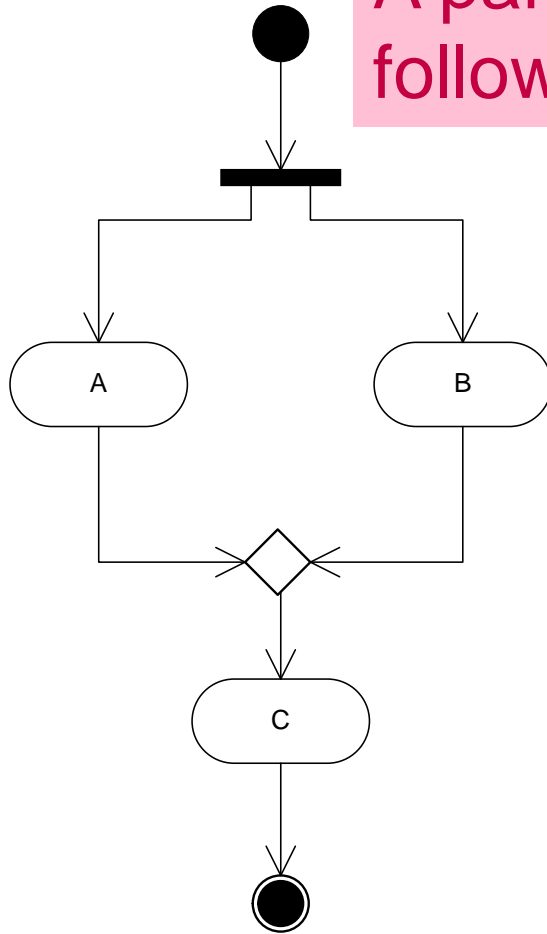
c) A simultaneously with C, B ✓

d) B simultaneously with A, C



# What would be the execution order?

A parallel with B,  
followed by C (twice)



A or B depending  
on the condition,  
then blocked

# Activity diagrams as a specification technique?

*Unambiguous?*

*Realistic?*

*Verifiable?*

*Evolvable?*

# Activity diagrams as a specification technique?

## *Unambiguous?*

- Becomes better with more recent versions of UML (formal semantics via Petri nets)
- Still, we might like to verify the activity diagram itself!

## *Realistic?*

- Yes (up to some level)

## *Verifiable?*

- Parallelism might be difficult to observe

## *Evolvable?*

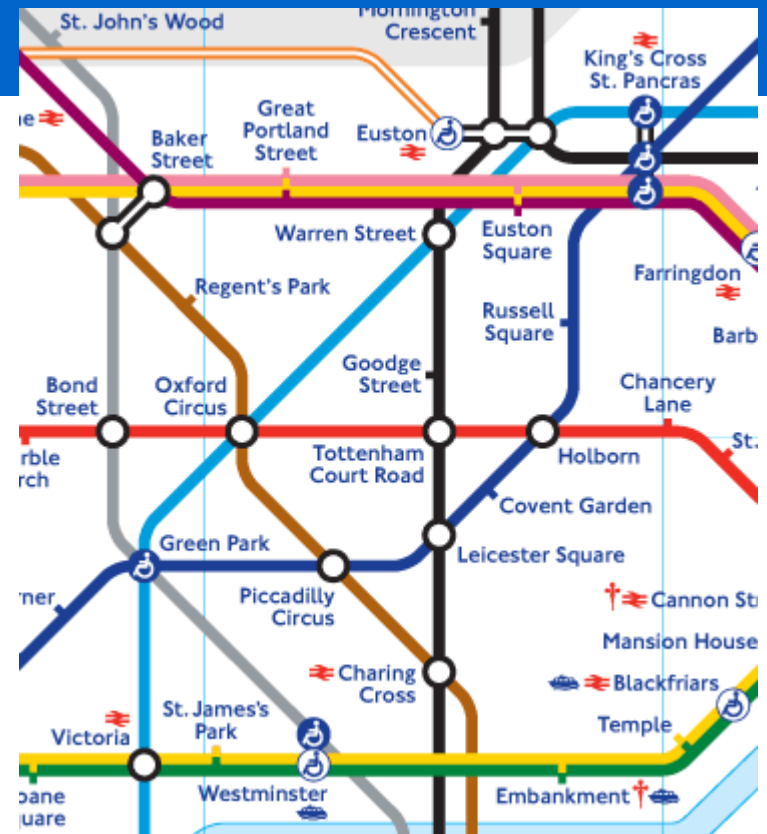
- See discussion of the diagram complexity

# State machines

- **State machines** – a different way of modeling behavior in UML
- Activity diagrams focus on actions (stories, narratives)
  - first A, then B, ...
- State machines focus on state of the object and transitions between them (description)
  - H<sub>2</sub>O:
    - States: water, vapor, ice
    - Transitions: melting, freezing, vaporization, condensation, ...
- Can give complementary views on the system

# Activities or states

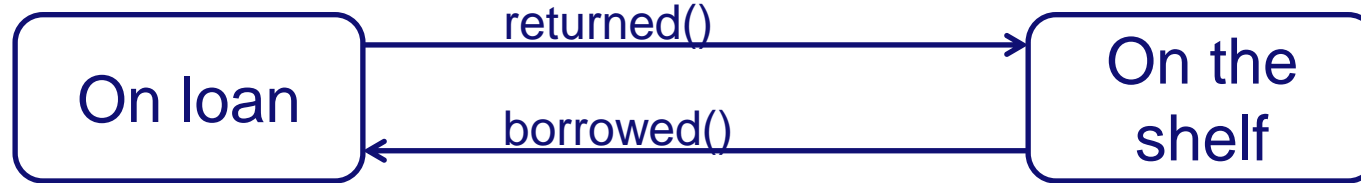
- Activities:
  - How to get to **Covent Garden** from **St. James's park**?
- States (example):
  - States = stations
  - Transitions = direct train connection



Fragment of the London's tube map

# Library

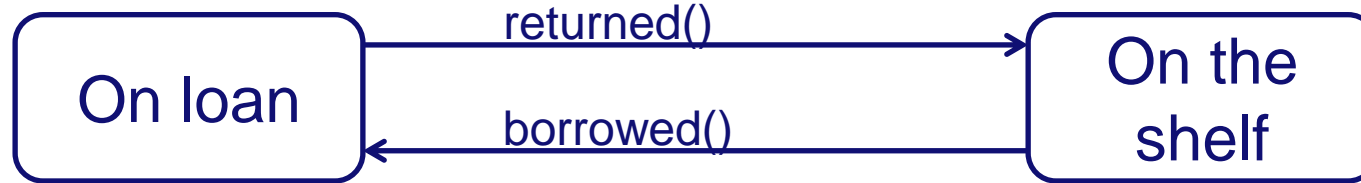
- A UML state machine diagram shows how the messages that an object receives change its **state**
  - The state is determined by the values in its attributes



- Object: Book
  - Boolean attribute: onShelf
  - **States:** “On the shelf” (onShelf=true) and “On loan” (onShelf=false)
  - **Transition:** change of state
  - **Event:** message received by an object, causing change of state

# Library

- A UML state machine diagram shows how the messages that an object receives change its **state**
  - The state is determined by the values in its attributes

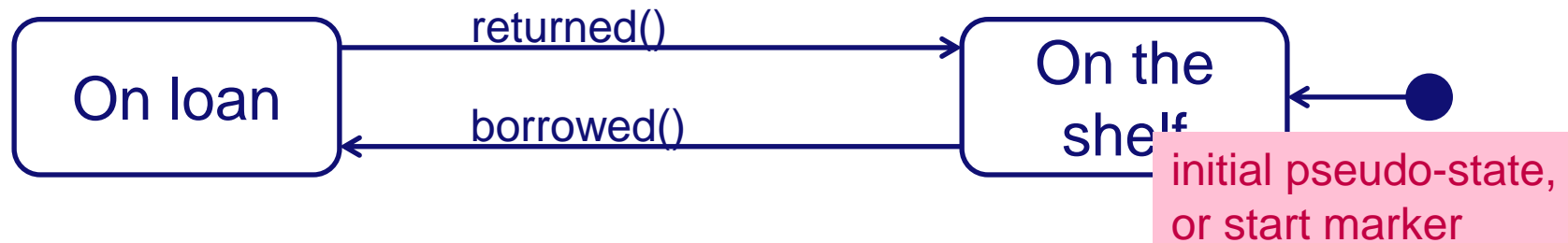


- This diagram does not indicate what happens if “returned()” is received when the book is already on the shelf.
  - Probably, an error should be reported.



# Library

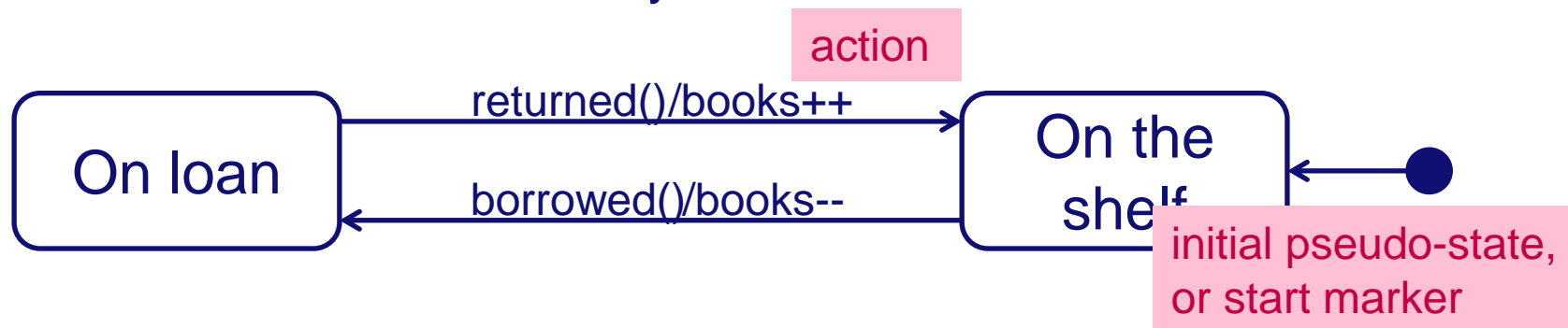
- A UML state machine diagram shows how the messages that an object receives change its **state**
  - The state is determined by the values in its attributes



- When a new book is being purchased by the library, it should be “On the shelf”

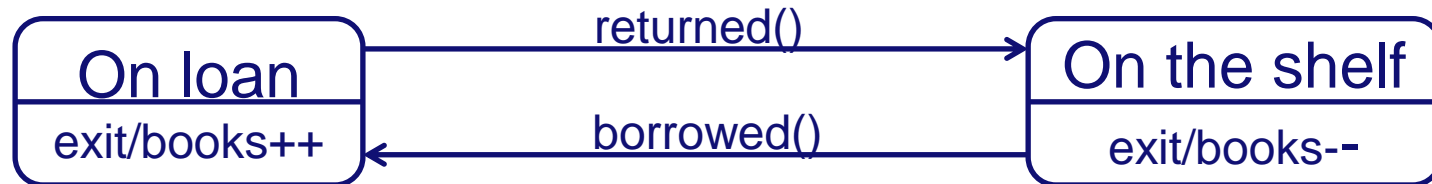
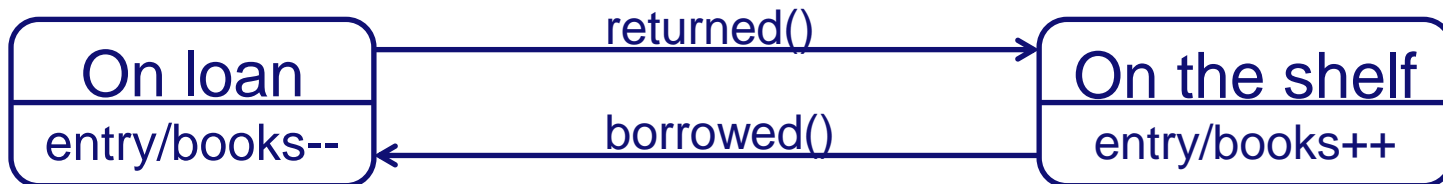
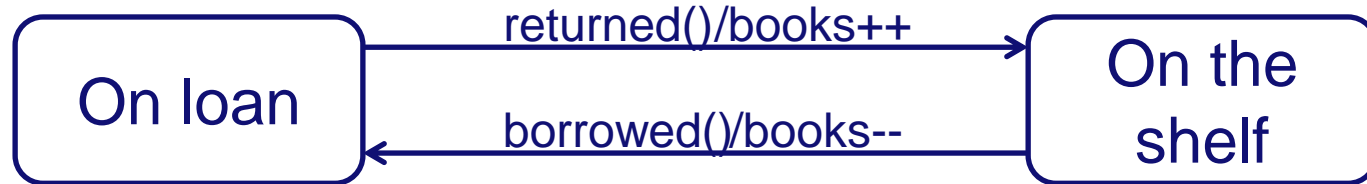
# Library

- A UML state machine diagram shows how the messages that an object receives change its **state**
  - The state is determined by the values in its attributes



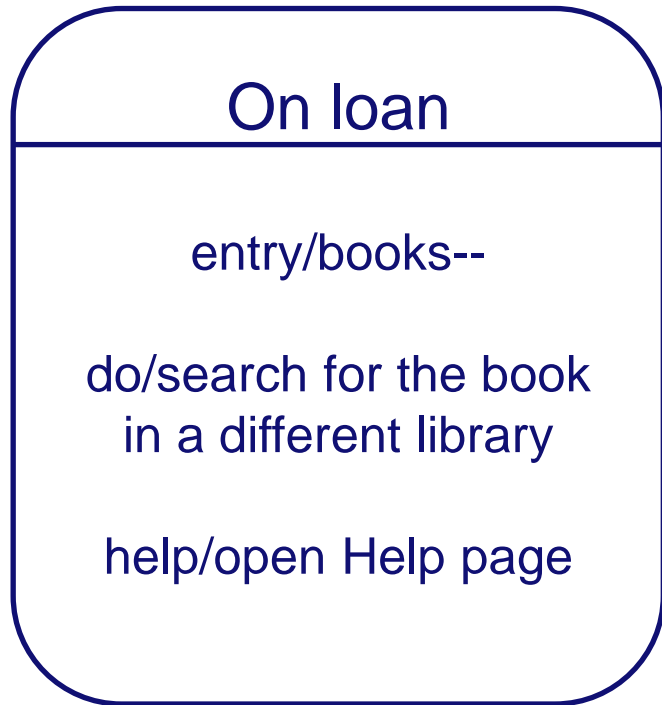
- When a new book is being purchased by the library, it should be “On the shelf”
- *Recall*: **event** – message received by an object, causing change of state.
- **Action**: message sent by an object, when changing a state

# Entry and exit actions



- Three equivalent representations
- When would you prefer to use entry/exit actions instead of actions on arrows, and why?

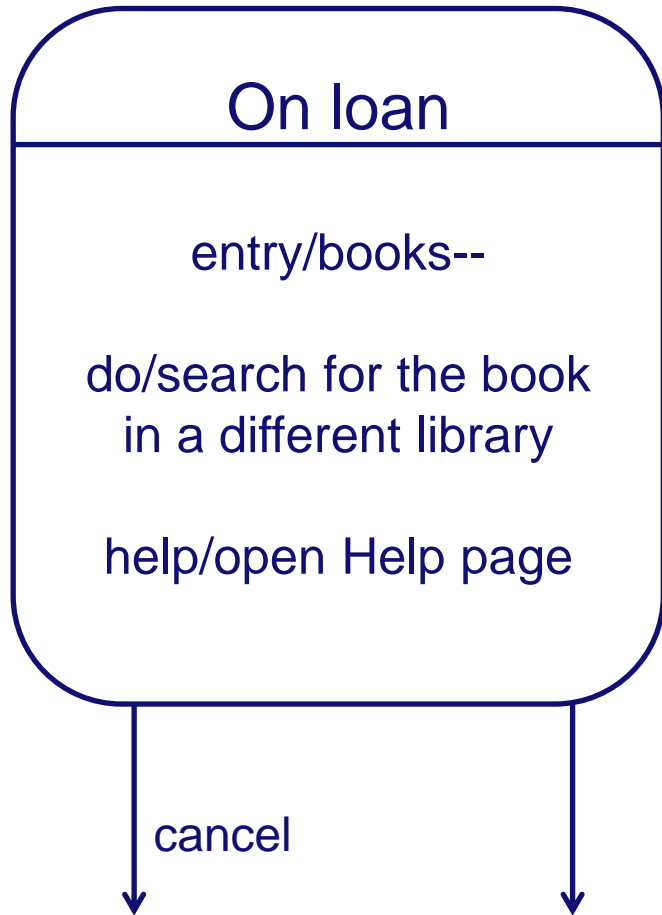
# Closer look at a state



- **entry**/behavior performed upon entry to the state
- **do**/ongoing behavior, performed as long as the element is in the state
- **exit**/behavior performed upon exit from the state
- behaviors can be added

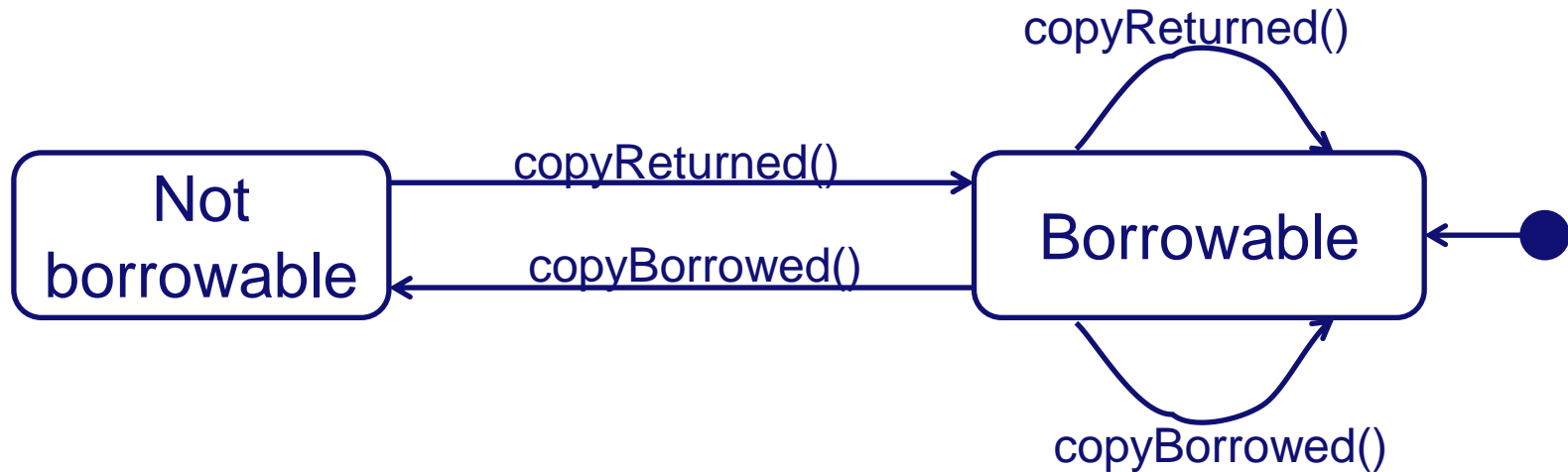
*do and internal behaviors do not change state!*

# Closer look at a state



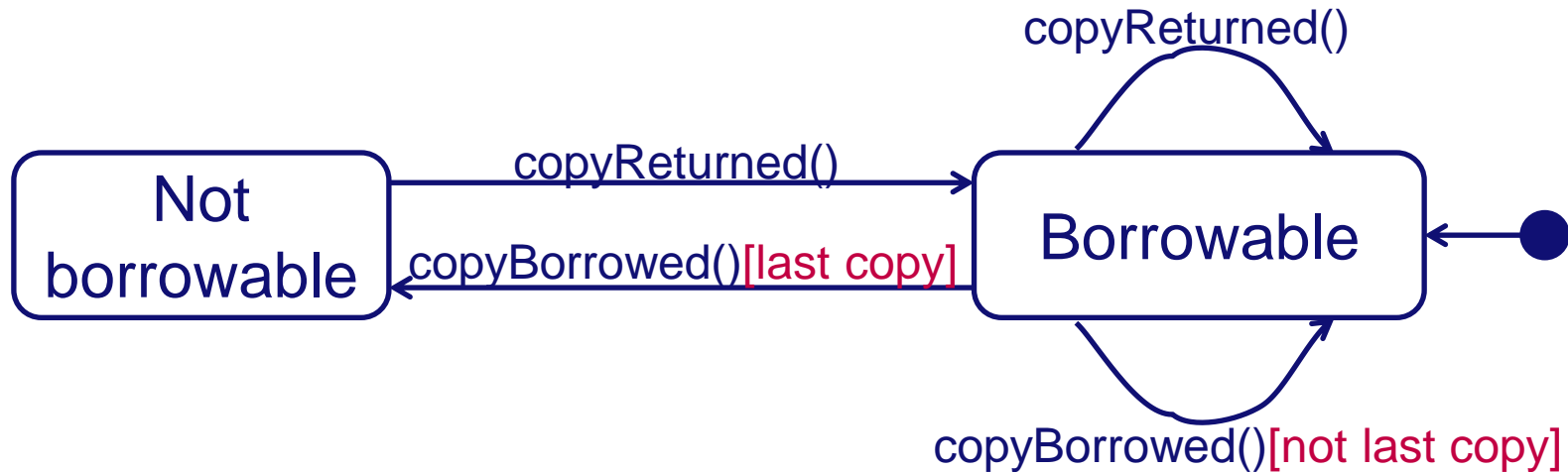
- Once do-activity completed, transition without a label is taken
- If “cancel” event occurs during the do-activity, then it is aborted
- Do-activities can be interrupted but regular activities (actions) cannot

# Library revisited



- A book is **borrowable** if at least one copy is available
- How can we know whether `copyBorrowed()` should change the state or not?
  - If the copy borrowed was the last one...
  - We need a mechanism to distinguish between the two situations

# Library revisited



- A book is **borrowable** if at least one copy is available
- How can we know whether `copyBorrowed()` should change the state or not?
  - If the copy borrowed was the last one...
  - We need a mechanism to distinguish between the two situations
- **Guard**: any unambiguous boolean expression

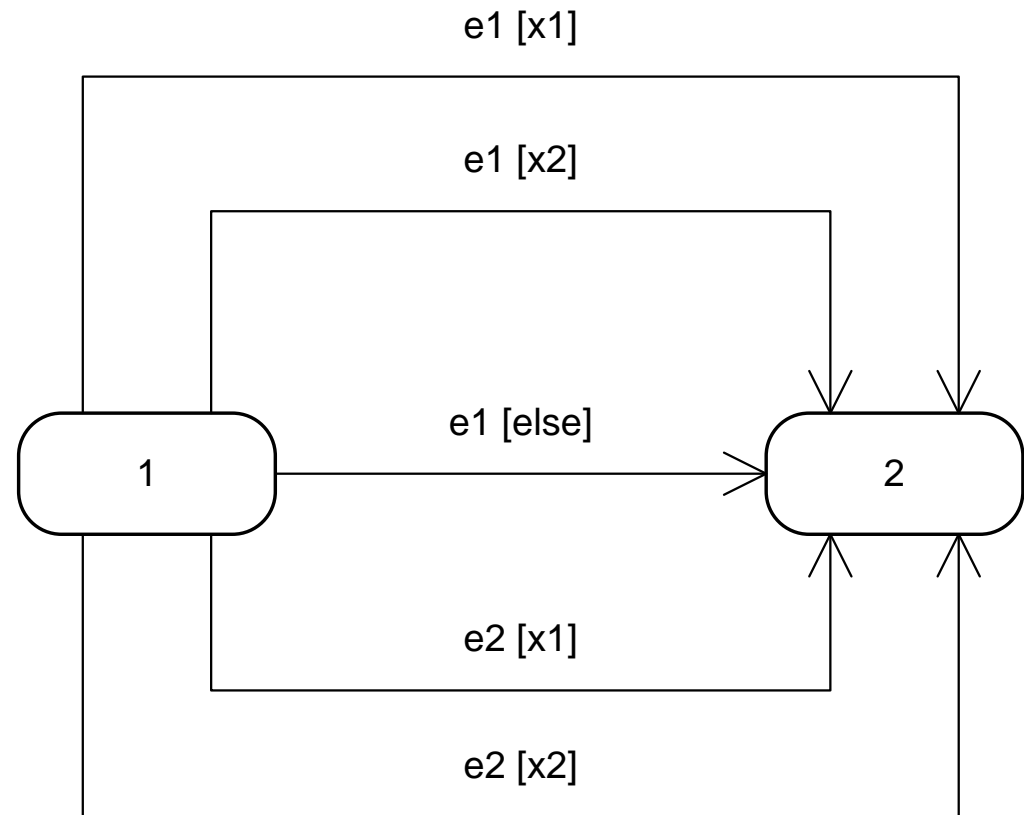
# What would happen if...

- **Current state: 1**

a) Event: e1, both x1 and x2 hold

b) Event: e1, neither x1 nor x2 holds

c) Event: e2, neither x1 nor x2 holds





# What would happen if...

- **Current state: 1**

a) Event: e1, both x1 and x2 hold

e1 [x1]

The state changes to 2 but it is unpredictable which transition will be taken

b) Event: e1, neither x1 nor x2 holds

The state changes to 2.

c) Event: e2, neither x1 nor x2 holds

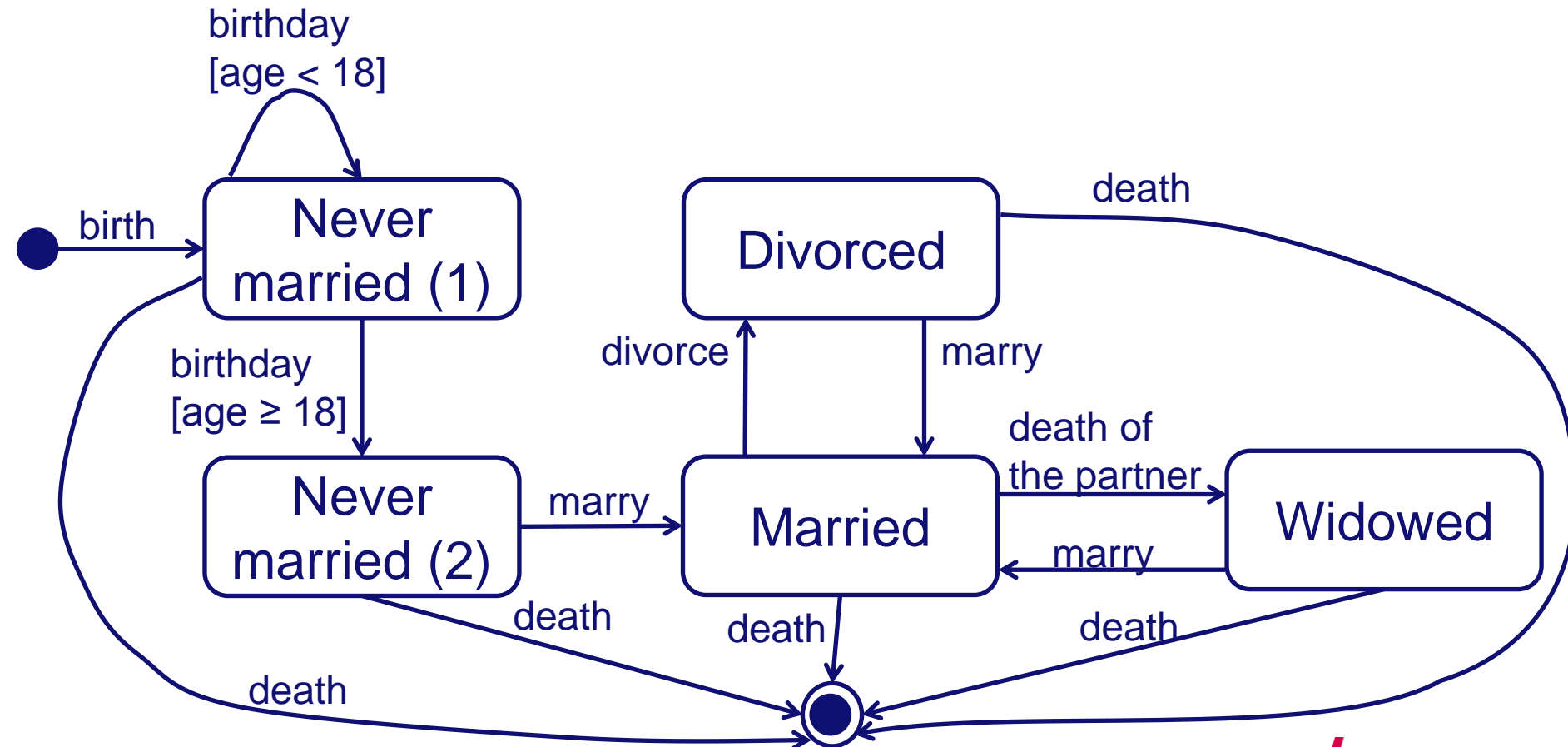
Transitions that have a guard which evaluates to false are disabled. State does not change.

# Exercise

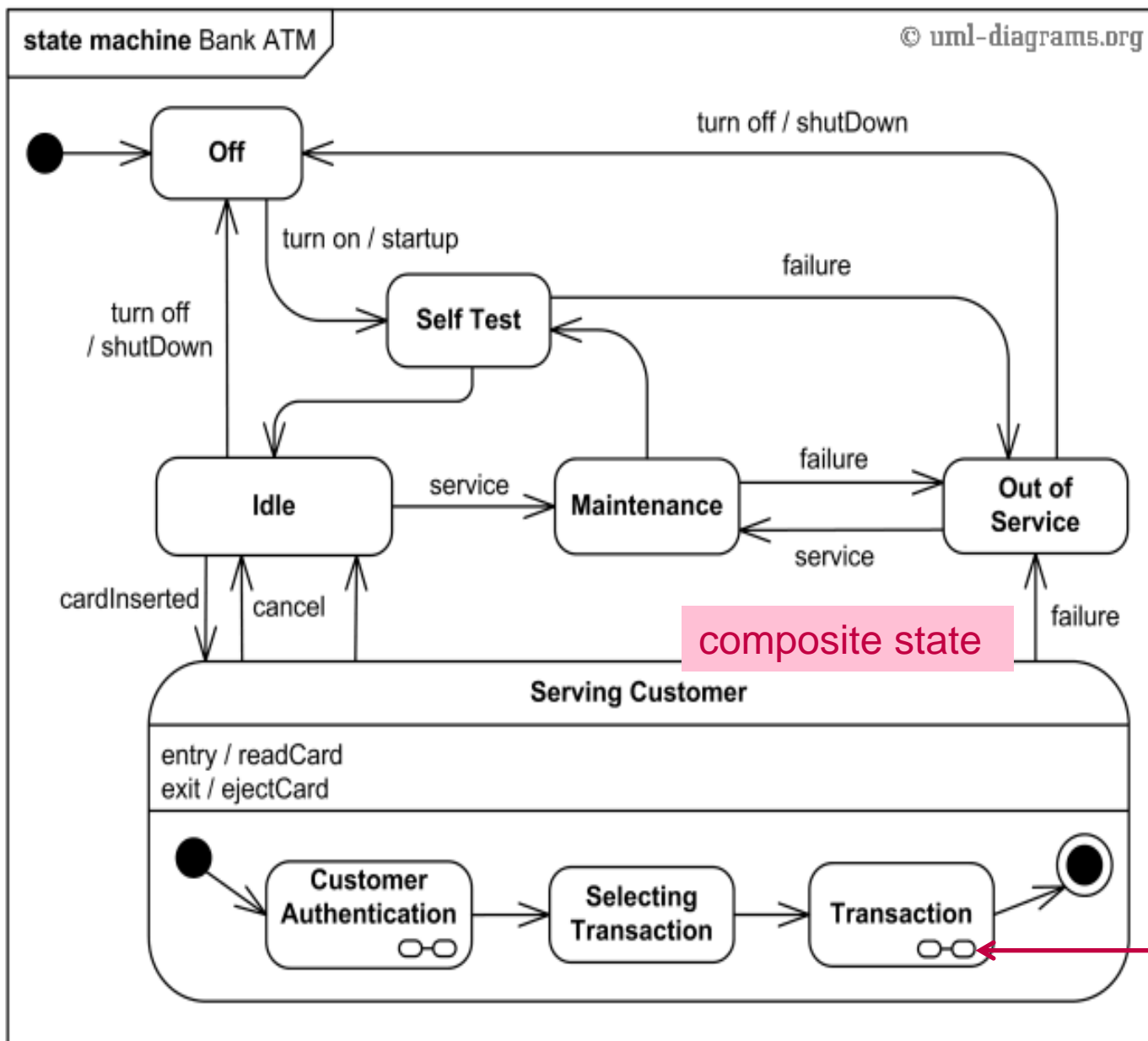
- Dutch statistics office recognizes the following marital statuses: *never married, married, widowed, divorced*
- Describe human life from birth to death

# Exercise

- Dutch statistics office recognizes the following marital statuses: *never married, married, widowed, divorced*



# Another example: ATM

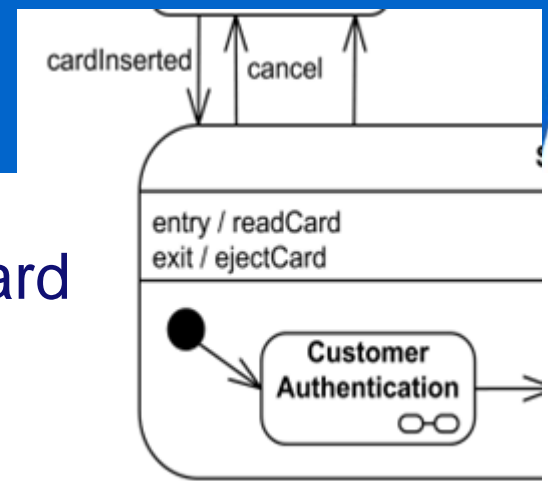


- **failure** and **cancel** are applicable to all states within **Serving Customer**
- **cardInserted** leads to the start marker
- transition without a label is followed by the stop marker

composite state with a hidden decomposition indicator icon

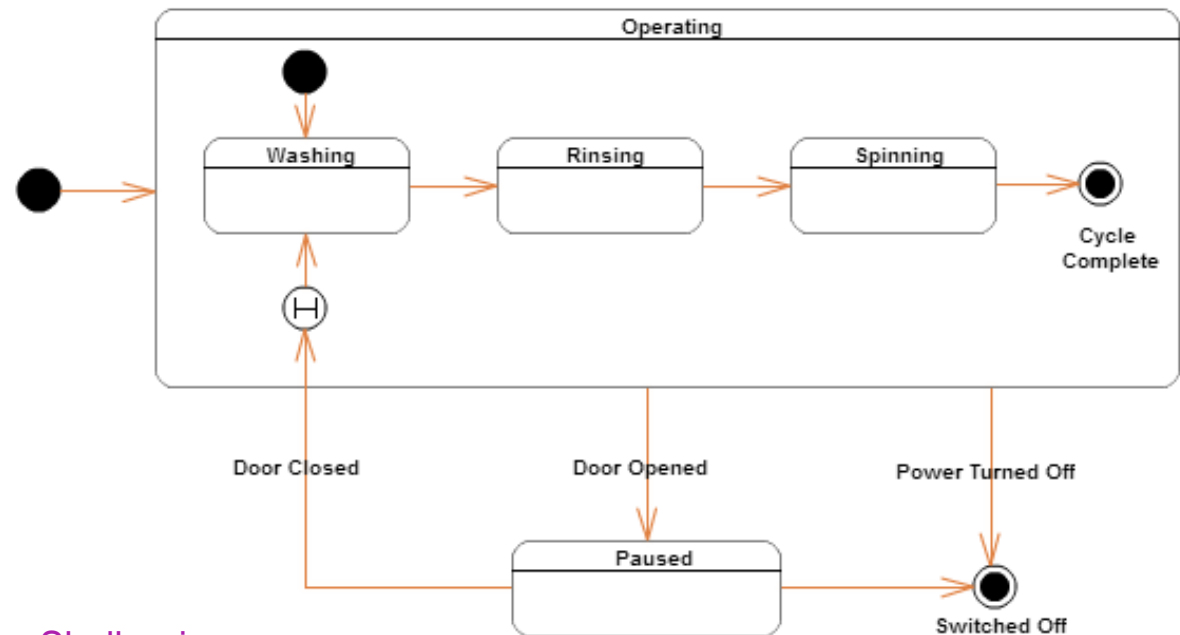
# ATM and more

- If an ATM operation is cancelled, the card is ejected and if it is reinserted, the “Serving Customer” should **restart**.
- Sometimes we would like to **resume** where we have stopped:
  - Closing the door of the washing machine



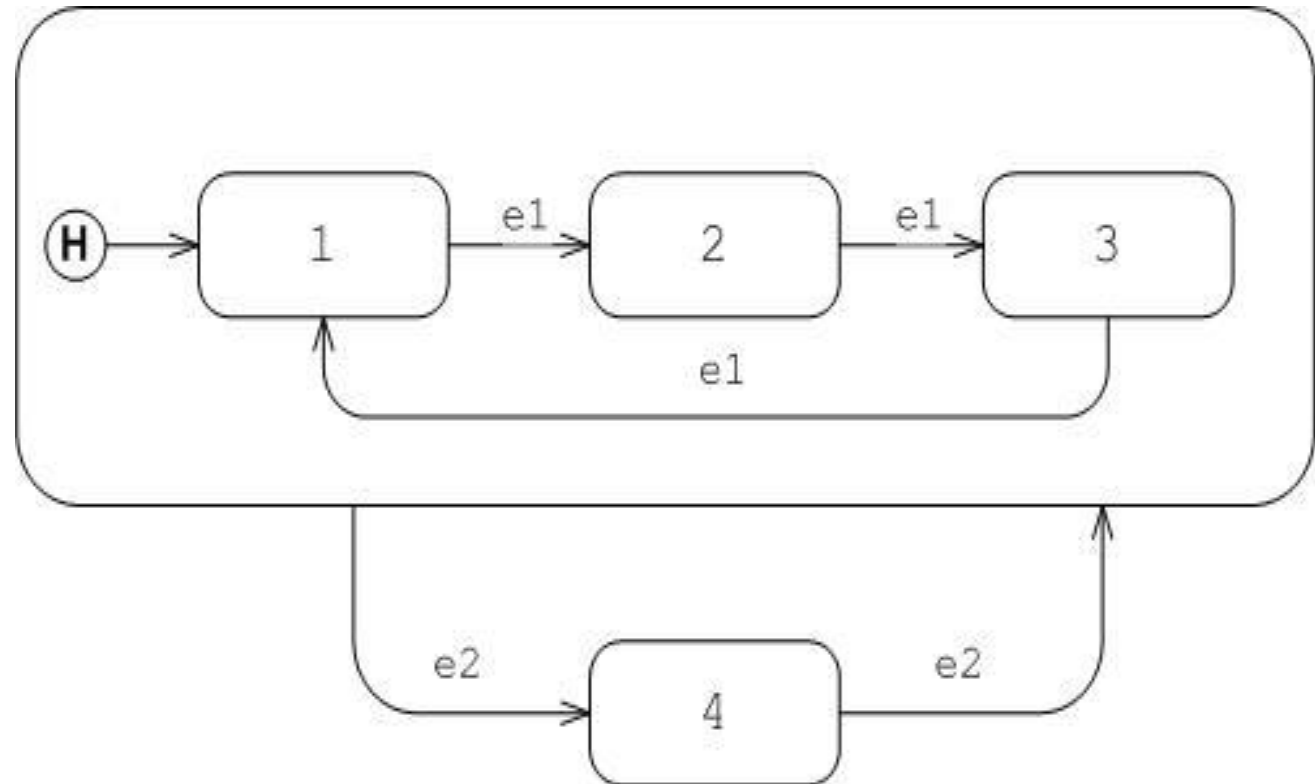
History state **H** remembers the state “Operating” was last time when it was exited.

Arrow from **H** indicates the state that should be entered if no previous history is available.



# What state would be reached from State 1 after

- a) e1, e2, e2, e1
- b) e1, e1, e2, e2
- c) e1, e1, e2, e2,  
e1, e1, e2, e2



# What state would be reached from State 1 after

a) e1, e2, e2, e1

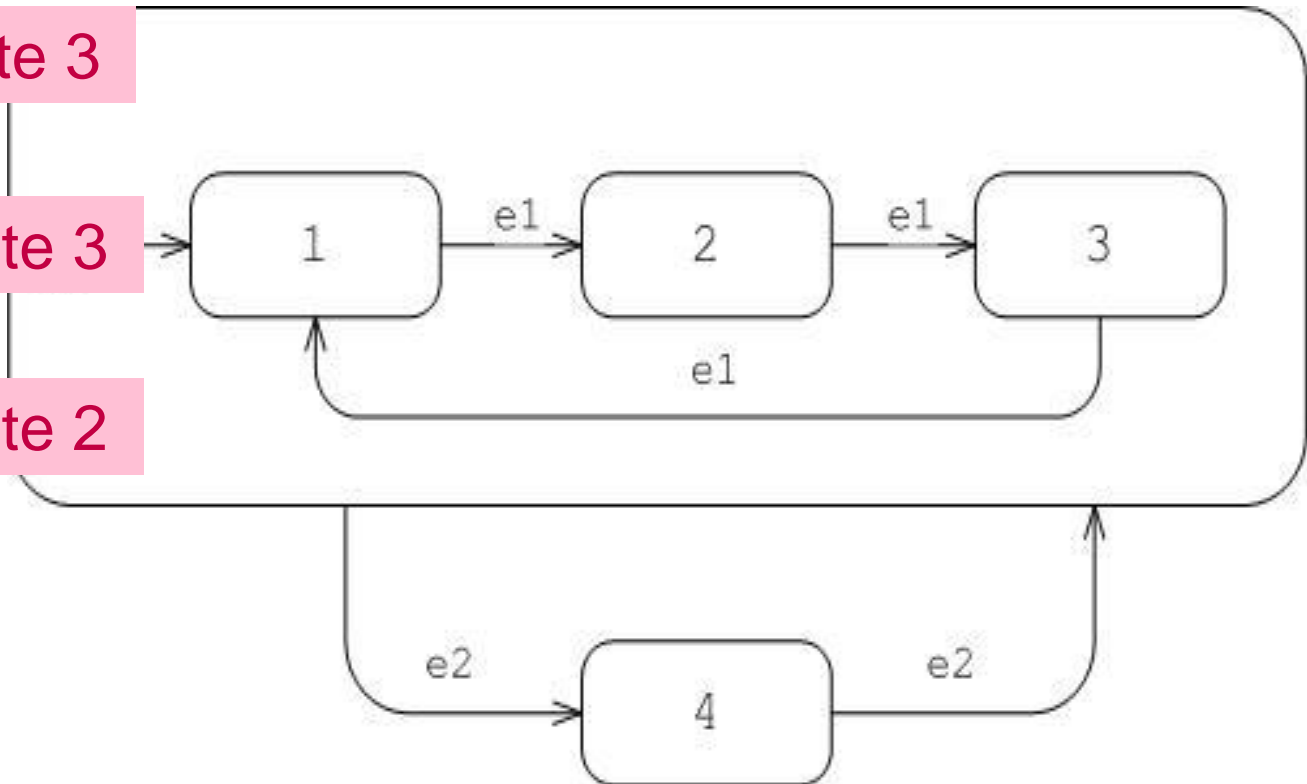
State 3

b) e1, e1, e2, e2

State 3

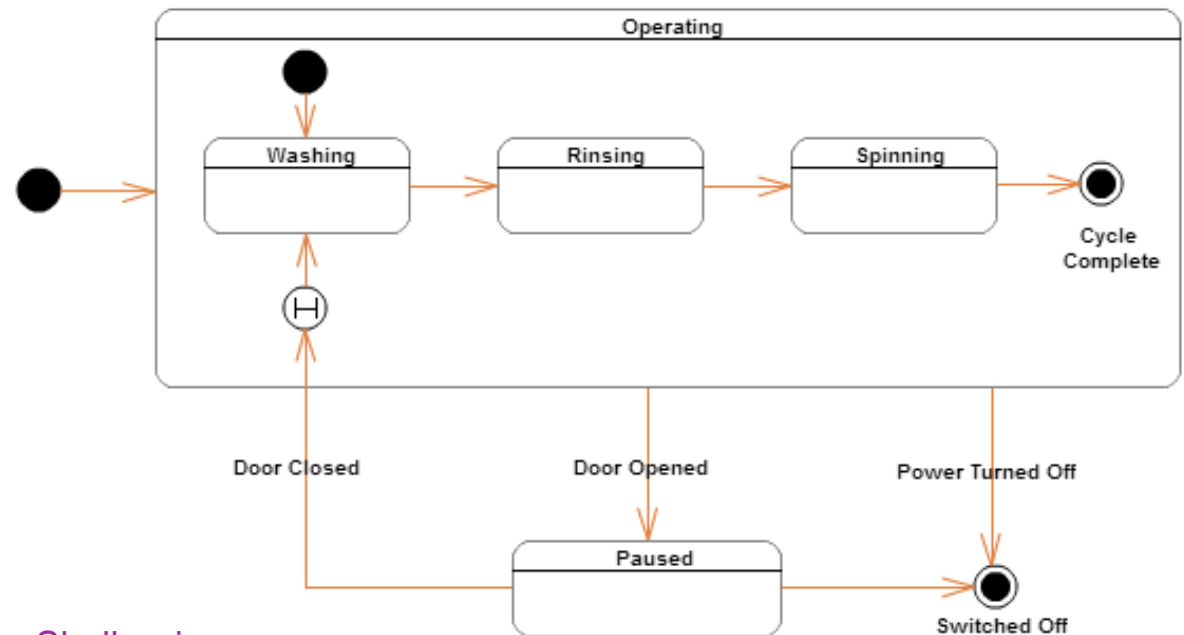
c) e1, e1, e2, e2,  
e1, e1, e2, e2

State 2



# History and nested states

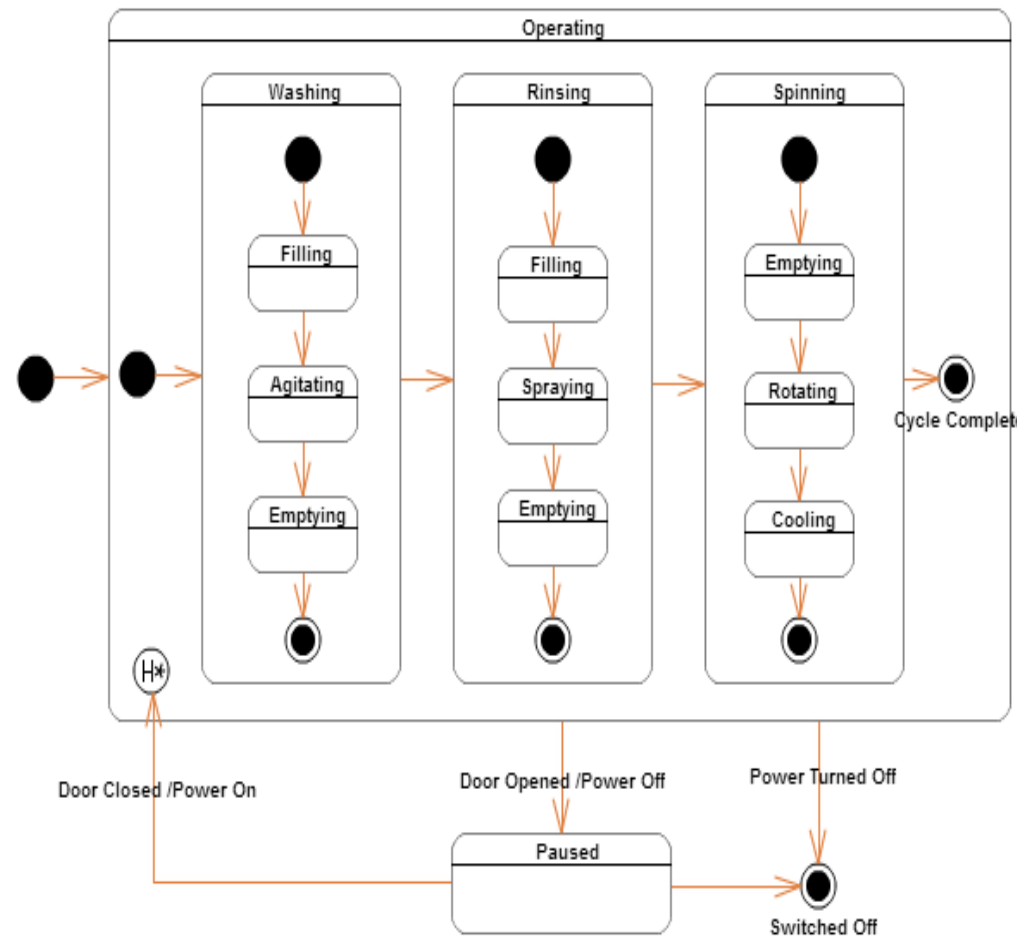
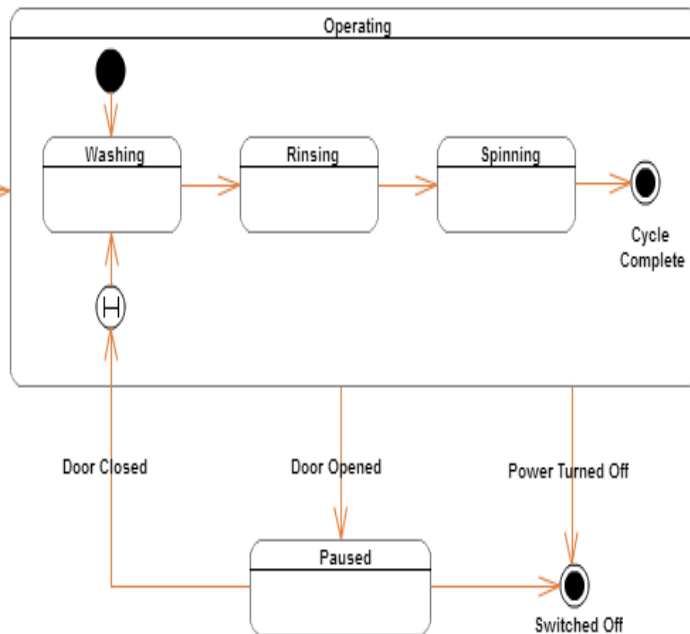
- Composite states can contain composite states that can contain composite states etc
- What if Washing, Rinsing, Spinning were composite states?
- In which of their substates should the execution resume?





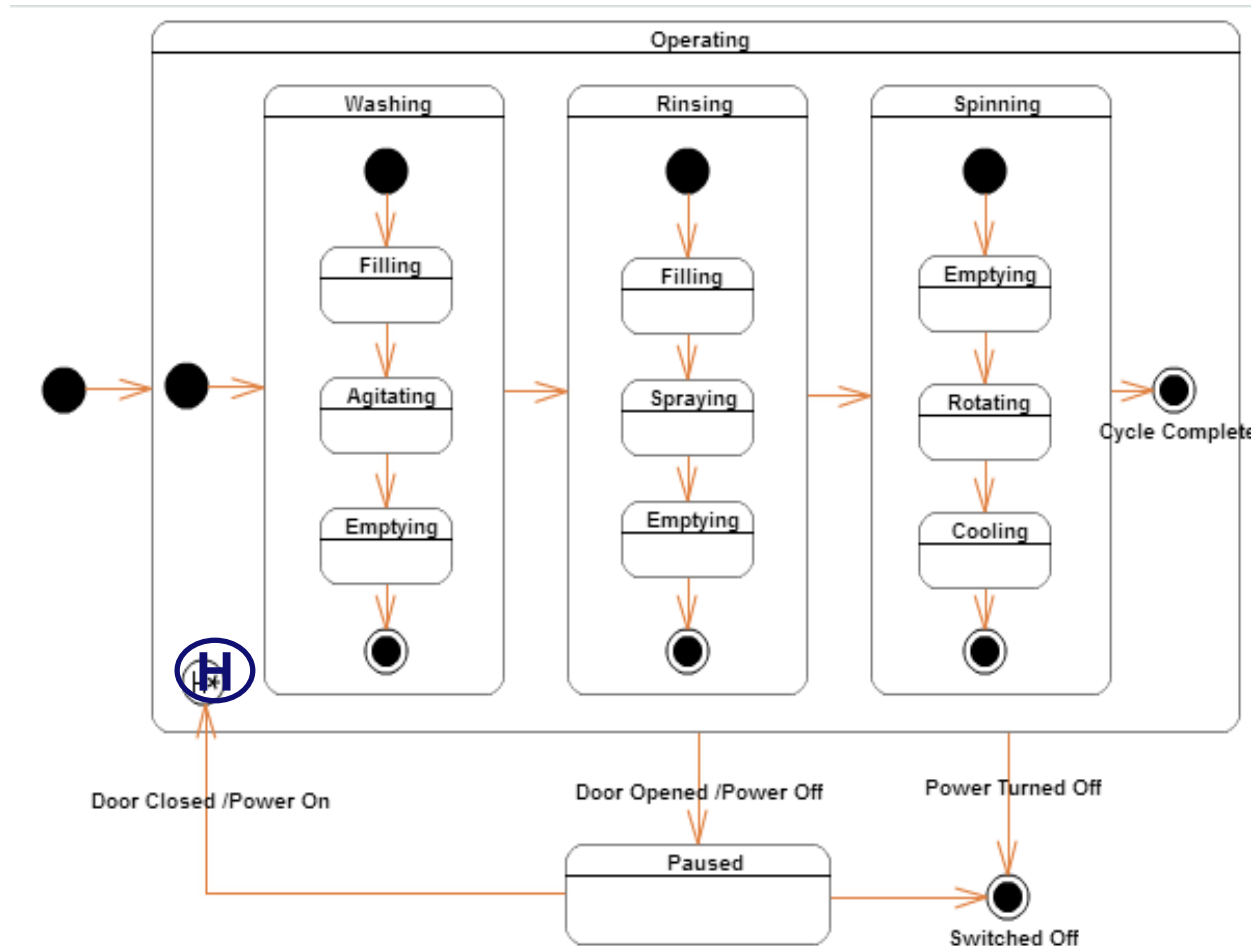
# Shallow and deep history

- **Shallow history H** remembers only one nesting level
- **Deep history H\*** remembers all nesting levels



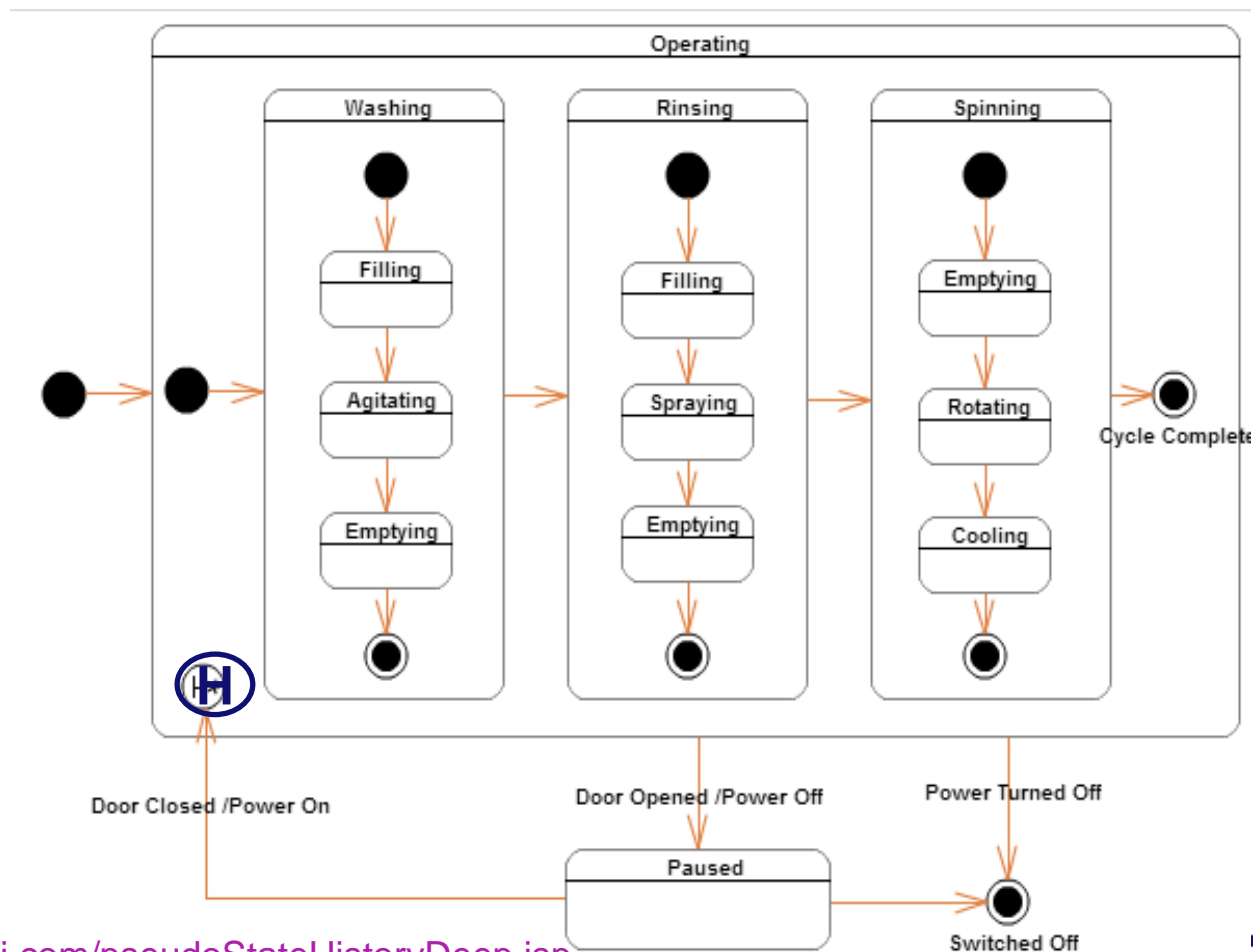
# Shallow and deep history

- What would happen if  $H^*$  would be replaced by  $H$ ?

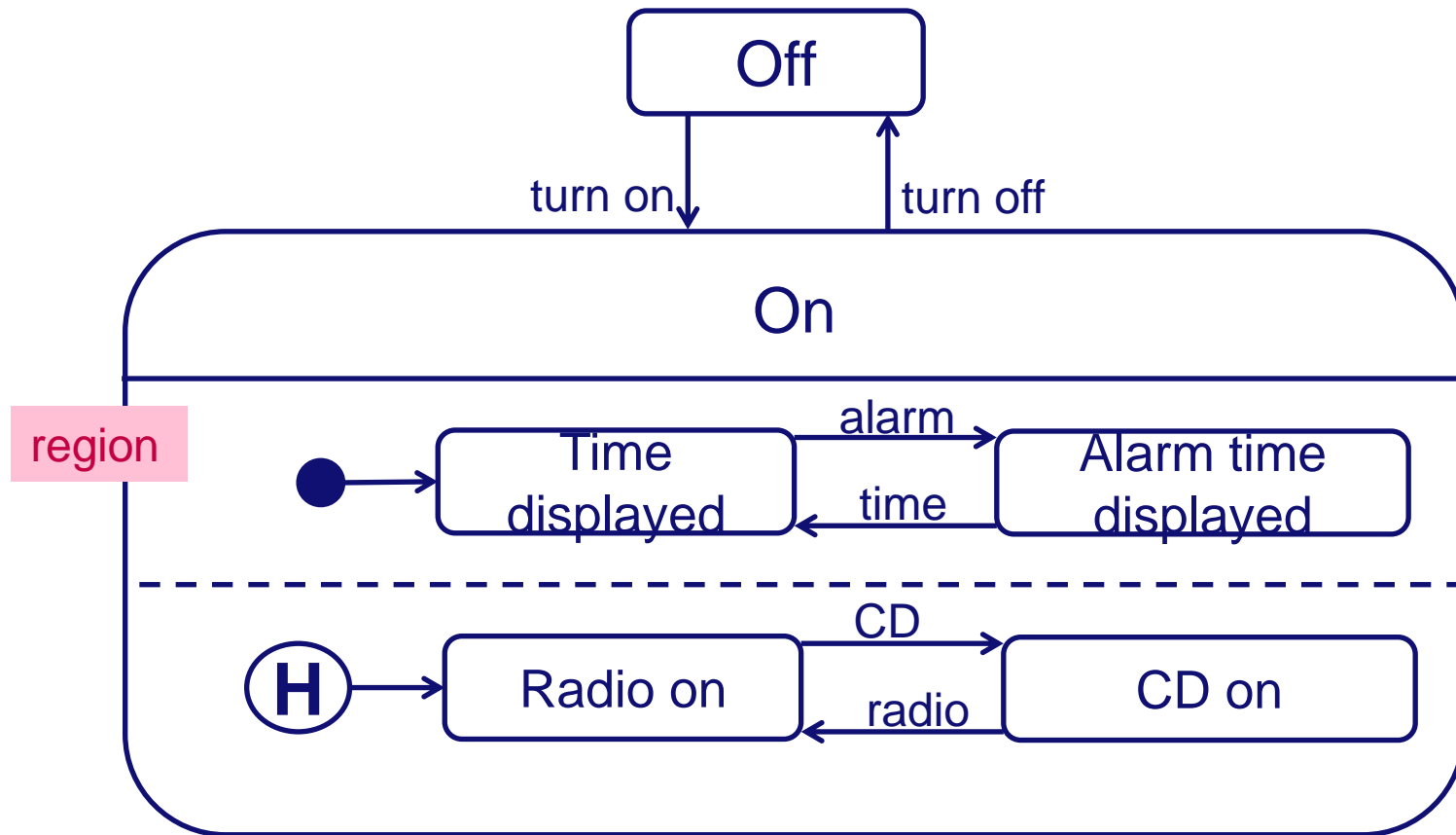


# Shallow and deep history

When door is closed the phase(s) [washing, rinsing, spinning] that have been completed will remain completed, while the current phase will be restarted from the beginning.



# Another example: alarm clock



- Orthogonal, **concurrent** state machine diagrams
  - Diagrams cannot share states
  - The choices CD/Radio and Time/Alarm time are orthogonal

# State machines: summary

- **State machine:** state, transition, guard/event/action
- **Composite state**
- **Regions and concurrent execution**
- **Shallow history vs. deep history**

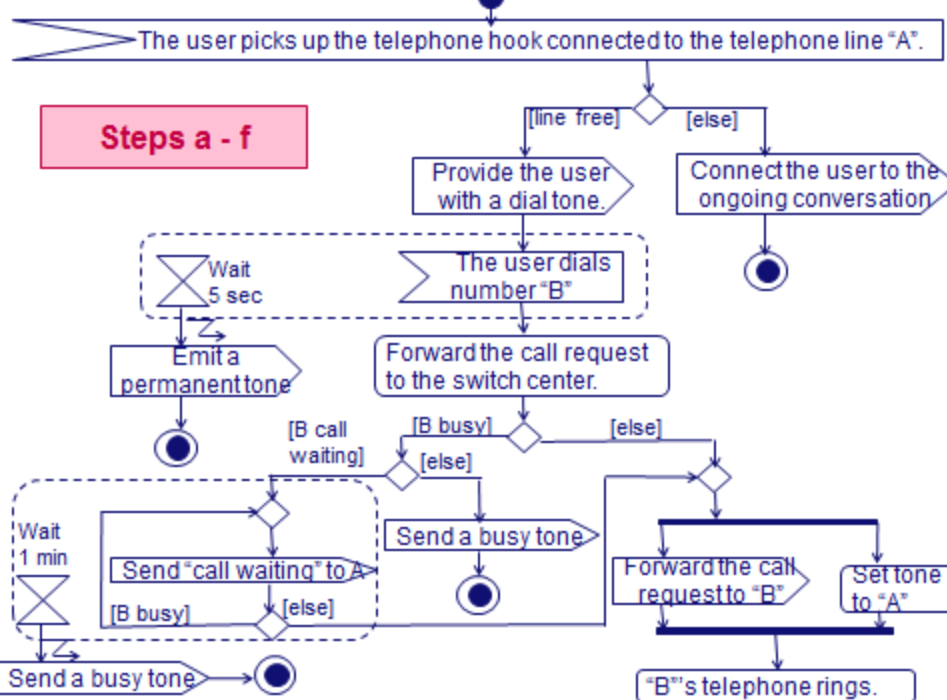
# State machines as a specification technique?

Similarly to activity diagrams.

## Summary of activity diagram elements

	Graphical representation	Description
Action		action with three inputs
Control flow		start / stop markers
		decision, merge
		fork / join
Signals		incoming (accept), outgoing (send), time-based
Interrupts		interruptible activity region, interrupting edge
Subactivity		activity with input/output parameters, activity invocation
Collection		expansion region

Steps a - f

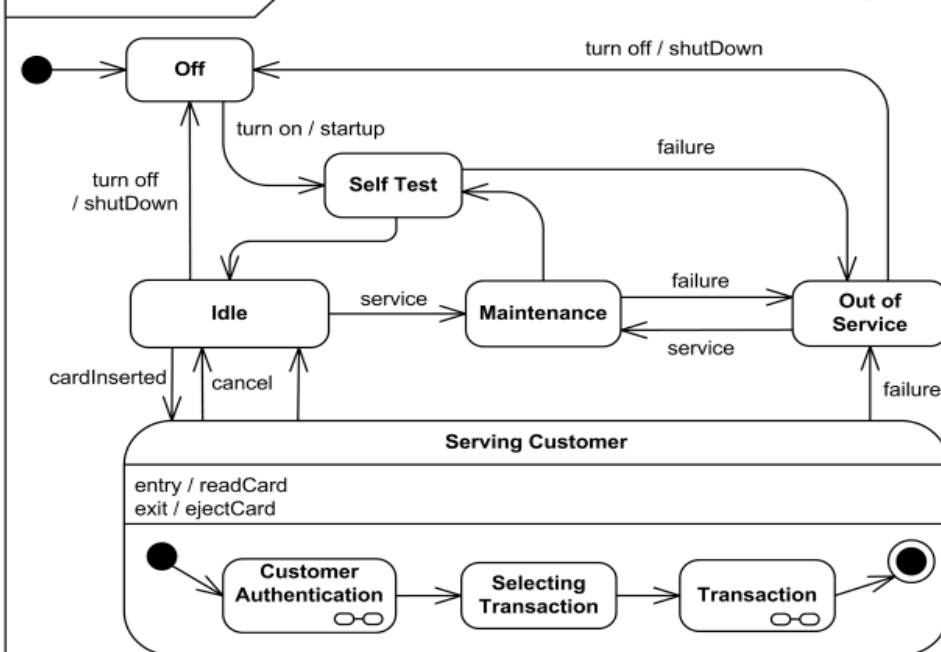


/ SET / WS/

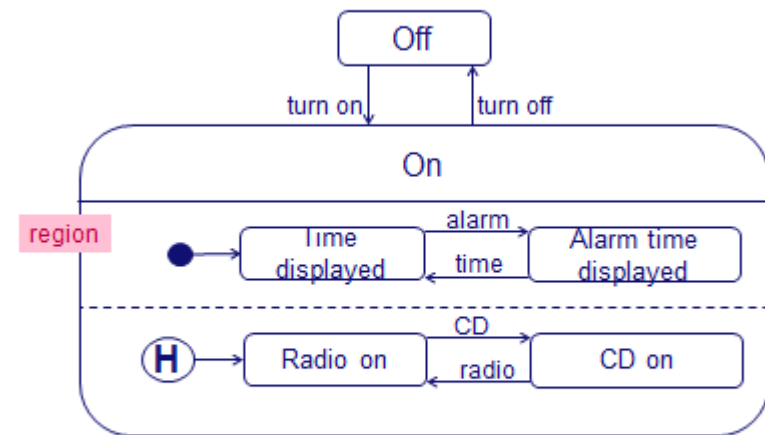
2-1-2014 PAGE 22

state machine Bank ATM

© uml-diagrams.org



## Another example: alarm clock



- Orthogonal, **concurrent** state machine diagrams
- Diagrams cannot share states
- The choices CD/Radio and Time/Alarm time are orthogonal

/ SET / WS/

7-2-2014 PAGE 41