Brief Introduction to UML 2.0 *II*

– Activity & State Machine Modeling

(for SEG seminar)

Tian Zhang

Nanjing University,China

October 2005

Outline

- Foreword
  - OO visual modeling and formal methods
  - Specification of UML2.0 language
- Activity Modeling
- State Machine Modeling

OO建模的图形化和形式化方法

- Based on graphic mode, *e.g. UML, MSC/SDL*
  - easy to use
  - hard to translate, execute
  - 对需求、设计的结果缺乏进行分析、验证的手段
- Based on formalization, e.g. *Z, VDM*; *logic, algebra, Petri nets, state machine*
  - 理论上可以解决上述问题，但复杂的形式使其难以推广
- 如何把直观自然的OO图形建模法和形式化技术相结合，是目前普遍关注的问题*（如何折中！）*

常见的几种动态行为语义模型

- 基于逻辑的形式化方法
  - 一阶线性/分支时序逻辑、模态μ演算、分布时序逻辑、TLA以及线性逻辑
    - 用来描述系统的生命期约束性质,如系统的安全性、活性、公平性及不变性等
    - 在逻辑的形式公理系统上,还可以进行系统的*特性推理*
- 基于刻画系统并发行为的模型和方法
  - CCS,CSP,ACP等基于进程代数
  - I/O自动机、Büchi自动机等自动机模型
  - Petri nets
  - Actor模型

关于并发的形式语义

- 并发性是指一个系统内部发生的两个事件之间不存在因果关系。
  - 因果关系不等同于*先后关系*，有因果必有先后，但反之则未必
  - 在一个存在并发性行为的系统中没有统一的时钟
  - 用“*不存在因果关系*”来描述计算机网络的并发性是最合适的
- 对并发性的另外一种理解：
  - 并发性是指一个系统内部发生的两个事件之间的*先后次序*不确定
    - 即，如果有多个进程在活动，那么这些进程中全体事件的集合一定可以按照执行时间的先后排成一个*全序*
  - Milner认为这样做在数学处理上比较简单
  - 基于这种观点描述的语义称为*交叠式语义 (interleaving semantics)*;
    - CCS和CSP等进程代数的演算系统就是基于interleaving模型描述并发
  - 基于上一个观点描述的语义成为*真并发语义*。

再谈 Interleaving

- 用*Interleaving*的观点来理解并发，实际上是用*非确定性*特性代替系统的并发特征。能够这样做的理由：
  - 保持动作的原子性；
  - 在一定程度上对问题进行简化，以得到良好的代数性质。
  - 目前的*进程代数系统*就是采用Interleaving模型描述并发，典型的代表有CCS和CSP
- 基于Interleaving的模型描述并发的缺点是很多现象不能描述
  - 如组件对象内部多线程的并发执行
  - 毫无疑问，非确定性并不能代替并发
- 从*CCS, CSP*看*Petri nets*
  - Petri nets是描述*真并发*的典型系统，它没有CCS那样漂亮的演算系统，但它可以刻画更多并发现象，如并发、矛盾及接触等等

Petri nets

- defined in 1962 by Carl Adam Petri, extend *state machines* with a notion of *concurrency*
- *Execution (transition)* of Petri nets is *nondeterministic*, since multiple transitions can be enabled at the same time.
- Petri nets只给出模型，而没有演算。但是Petri nets是一种真并发模型，可以刻划更多的并发现象，如并发、矛盾及接触等。
- Petri nets描述的系统中不存在一个统一的时钟，这一点也是网论的本质。
- 总之：
  - *[并发]* Petri nets是一种真并发模型，用它可以描述更多的并发现象。针对分布式对象系统中的多输入流并发现象，可考虑采用Petri nets
  - *[交互]* Petri nets只有模型而没有没演算，利用它来验证系统的计算和语义特性较为困难。不过，可以将Petri nets描述并发的思想应用到刻画系统组件的交互之中。

CCS和CSP

- CCS和CSP是进程代数方法的代表,是描述*通信*和*并发*的演算系统
  - 它们均以进程为计算单位，进程的基本组成是原子性动作
- CCS和CSP都是既有模型又有演算，两者的不同点是：
  - CCS用同步树(synchronizationtree)表示进程,或称为状态转移图，而CSP用失败集(failureset)表示进程
  - CCS中,使用操作语义(互模拟语义)解释进程(的等价性);而在CSP中，则使用指称语义(失败语义)解释进程(的等价性)
- CCS和CSP等进程代数的演算系统均采用*Interleaving*模型描述并发
- 标准的CCS和CSP均以同步通信的方式描述进程间的交互
  - 有人在CCS的基础上进行一定的改进(从语法上,或从语义上),得到描述异步通信的演算系统

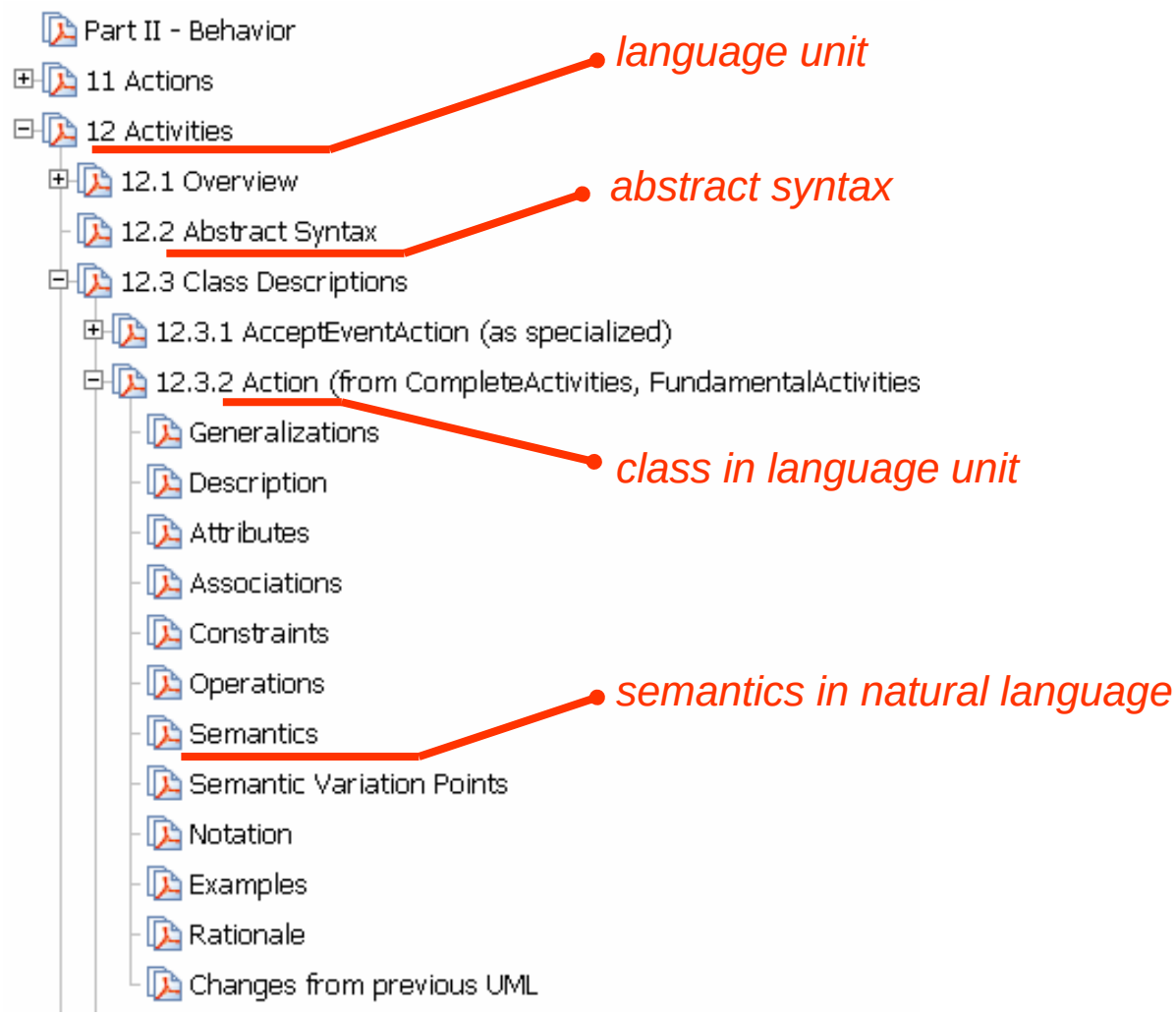
Specification of Language *(common)*

- A *common* technique for specification of languages
  - firstly, define the *syntax* of the language;
    - syntax defines what constructs exist in the language, and
    - how the constructs are built up in terms of other constructs.
  - secondly, describe its static and dynamic *semantics*.
    - *static semantics* define *how* an instance of a construct should be connected to other instances to be meaningful
    - *dynamic semantics* define the meaning of a *well formed* construct
      - *well formed* - fulfills the rules defined in the *static semantics*
      - the meaning of a description written in the language is defined only if the description is *well formed*

Specification of Language *(graphic, such as UML)*

- A *special* technique for specification of graphic languages
  - if the language has a *graphic syntax*, it is important to define the syntax in a *notation independent* way
  - firstly, to define the *abstract syntax* of the language
    - UML使用元模型给出抽象语法
  - secondly, to define *concrete syntax* by mapping the *notation* onto the *abstract syntax*
    - 给出具体的Notation，可以是图形化的也可以是文本
  - thirdly, to describe its semantics
    - UML中的语义是使用自然语言给出的
    - UML中constructs的静态语义和动态语义是同时给出的

Example: ***Action*** in *Activities* Language Unit



Abstract Syntax

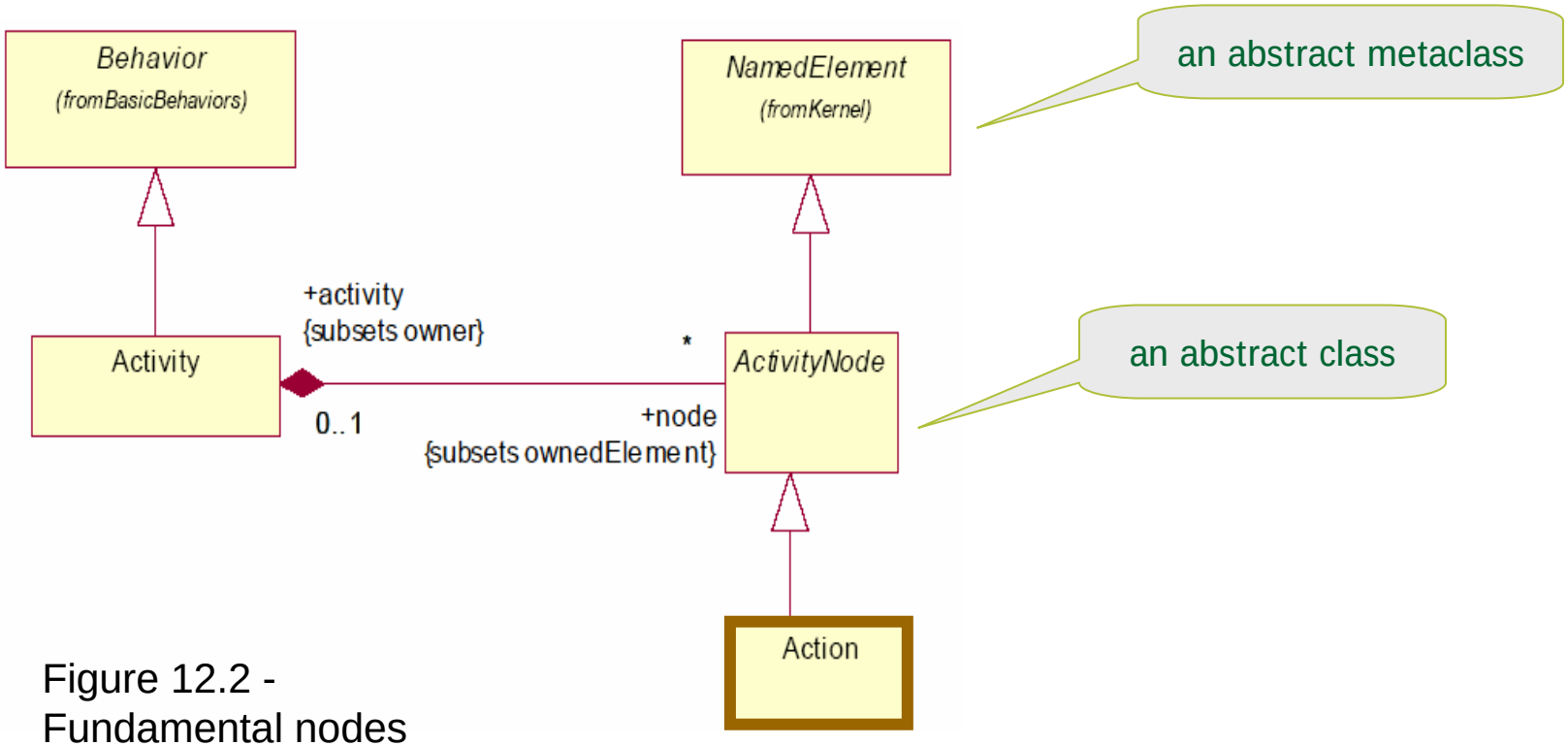
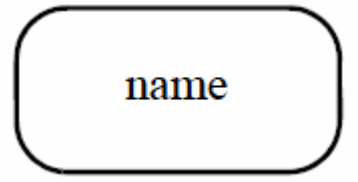


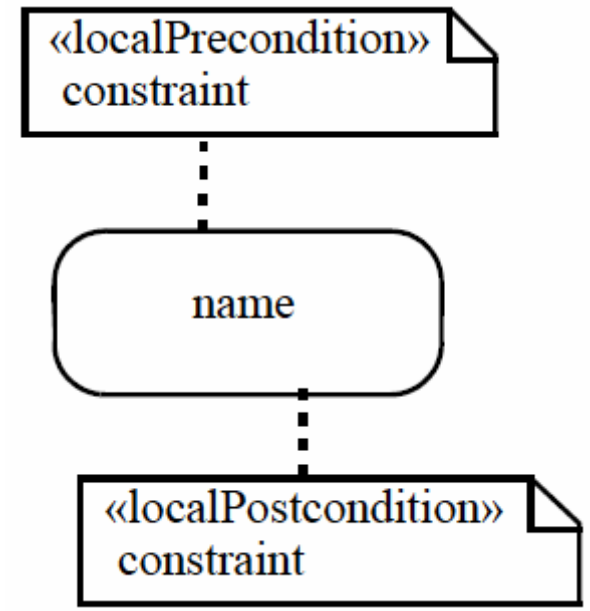
Figure 12.2 -
Fundamental nodes

- 使用元模型给出Fundamental nodes的抽象语法
  - 该语法描述中同时包含了Action的抽象语法
  - Action的抽象语法由多张类似的语法图定义

Concrete Syntax

- Use of action and activity notation is optional. A textual notation may be used instead.
- Actions are notated as round-cornered rectangles. The name of the action or other description of it may appear in the symbol.





Semantics

The sequencing of actions are controlled by control edges and object flow edges within activities, which carry control and object tokens respectively (see Activity). Alternatively, the sequencing of actions is controlled by structured nodes, or by a combination of structured nodes and edges. Except where noted, an action can only begin execution when all incoming control edges have tokens, and all input pins have object tokens. The action begins execution by taking tokens from its incoming control edges and input pins. When the execution of an action is complete, it offers tokens in its outgoing control edges and output pins, where they are accessible to other actions.

The steps of executing an action with control and data flow are as follows:

[1] An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). Exceptions to this are listed below. The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions. This ensures that multiple action executions competing for tokens do not accept only some of the tokens they need to begin, causing deadlock as each execution waits for tokens that are already taken by others.

[2] An action execution consumes the input control and object tokens and removes them from the sources of control edges and from input pins. The action execution is now enabled and may begin execution. If multiple control tokens are available on a single edge, they are all consumed.

[3] An action continues executing until it has completed. Most actions operate only on their inputs. Some give access to a wider context, such as variables in the containing structured activity node, or the self object, which is the object owning the activity containing the executing action. The detailed semantic of execution an action and definition of completion depends on the particular subclass of action.

[4] When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges (implicit fork), and it terminates. Exceptions to this are listed below. The output tokens are now available to satisfy the control or object flow prerequisites for other action executions.

[5] After an action execution has terminated, its resources may be reclaimed by an implementation, but the details of resource management are not part of this specification and are properly part of an implementation profile.

See ValuePin and Parameter for exceptions to rule for starting action execution.

If a behavior is not reentrant, then no more than one execution of it will exist at any given time. An invocation of a non-reentrant behavior does not start the behavior when the behavior is already executing. In this case, tokens control tokens are discarded, and data tokens collect at the input pins of the invocation action, if their upper bound is greater than one, or upstream otherwise. An invocation of a reentrant behavior will start a new execution of the behavior with newly arrived tokens, even if the behavior is already executing from tokens arriving at the invocation earlier.

Package ExtraStructuredActivities

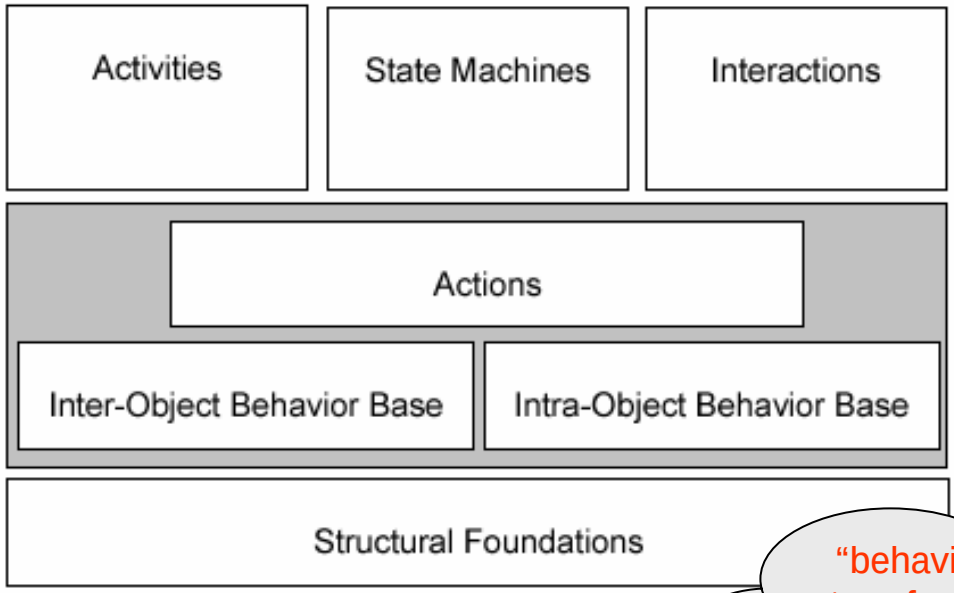
If an exception occurs during the execution of an action, the execution of the action is abandoned and no regular output is generated by this action. If the action has an exception handler, it receives the exception object as a token. If the action has no exception handler, the exception propagates to the enclosing node and so on until it is caught by one of them. If an exception propagates out of a nested node (action, structured activity node, or activity), all tokens in the nested node are terminated. The data describing an exception is represented as an object of any class.

Package CompleteActivities

Streaming allows an action execution to take inputs and provide outputs while it is executing. During one execution, the action may consume multiple tokens on each streaming input and produce multiple tokens on each streaming output. See Parameter.

Local preconditions and postconditions are constraints that should hold when the execution starts and completes, respectively. They hold only at the point in the flow that they are specified, not globally for other invocations of the behavior at other places in the flow or on other diagrams. Compare to pre and postconditions on Behavior (in Activities). See semantic variations below for their effect on flow.

The Semantics Architecture



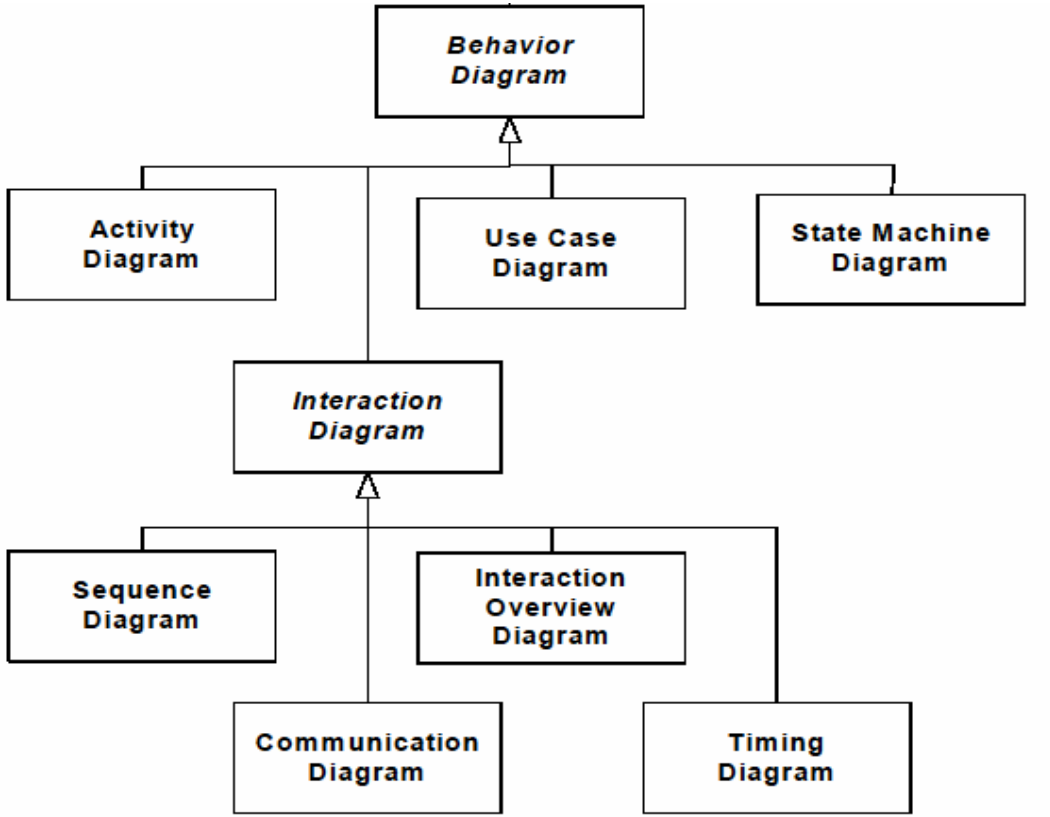
High-level behavioral formalisms, other behavioral formalisms may be added to this layer in the future.

provides the foundation for the semantic description of all the higher-level *behavioral formalisms*

"behavioral formalism" refers to a formalized framework for describing behavior, such as state machines, Petri nets, data flow graphs, etc.

A schematic of rum-time sem

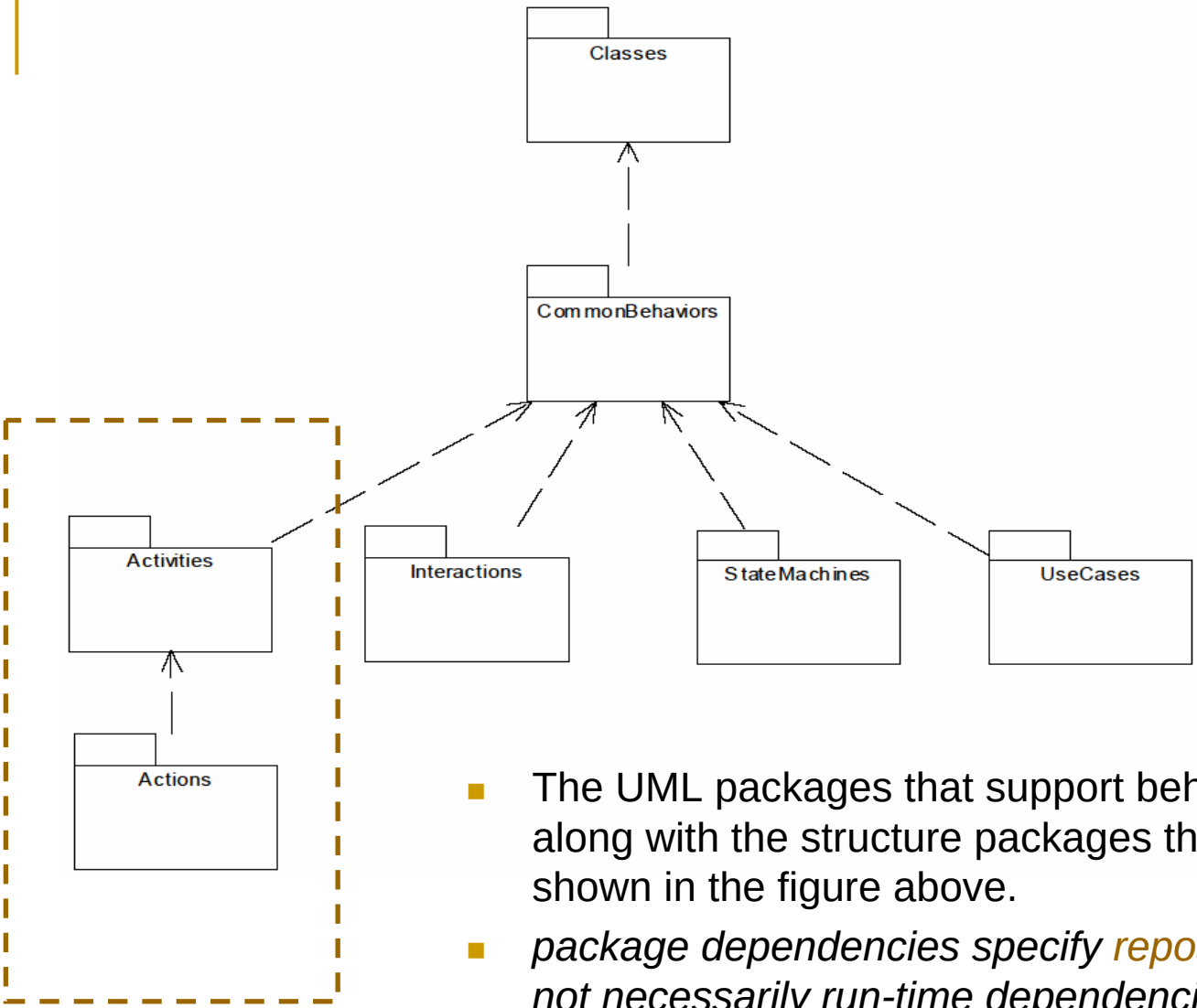
Review : Behavior Diagrams



Part II: Behavior

11. Actions
12. Activities
13. Common Behaviors
14. Interactions
15. State Machines
16. Use Cases

*(cite in **Table of Contents**)*



how a UML model behaves in a model repository

- The UML packages that support behavioral modeling, along with the structure packages they depend upon are shown in the figure above.
- *package dependencies specify repository dependencies not necessarily run-time dependencies*

Activity Modeling

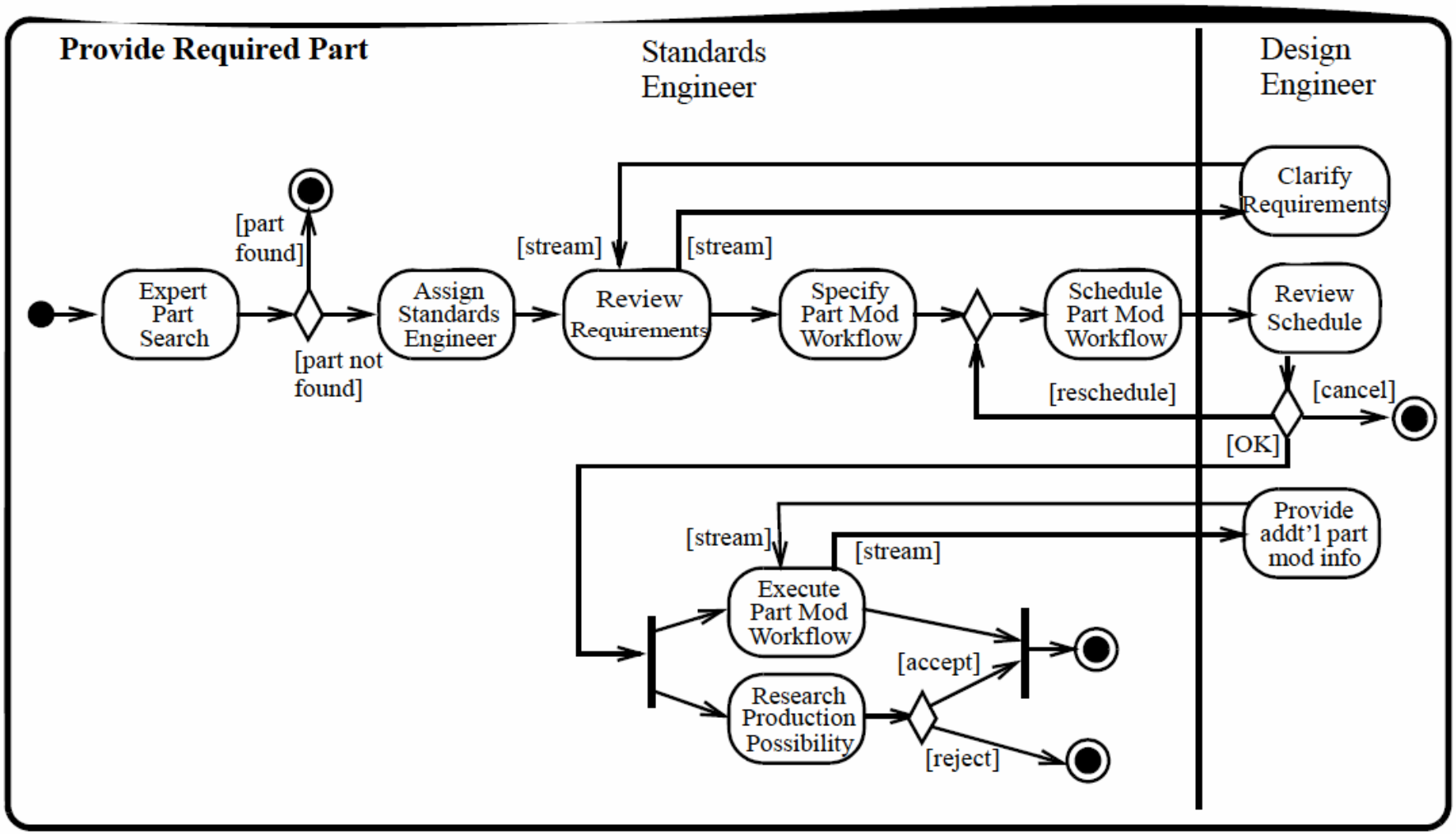
- Overview
  - Activity Diagrams
  - Change Summary
  - Example
- The core constructs and concepts in Activity
  - Decomposition, Partition, Flows and Edge, Signal, Token, Data Store Note, Pin, Expansion Region, Flow Final, Join
- Discussion about concurrency in Activity
  - compared with UML 1.5 *Interleaving* and *true concurrency*

Activity Diagrams

- Activity diagrams are a technique to describe *procedural logic*, *business process*, and *work flow*.
- Activity diagrams play a role similar to *flowcharts*
  - the principal difference between them and flowchart notation is that they support *parallel behavior*
- Activity diagrams are redesigned radically
  - significant syntax modification
  - the main difference is switching from State Machine based semantics to the token flow (Petri net like) semantics

Changes Summary

- *Activity* replaces *ActivityGraph* in UML 1.5.
- Activities are redesigned to use a Petri-like semantics instead of state machines
  - this widens the number of flows that can be modeled, especially those that have parallel flows
- *Activity* also replaces *procedures* in UML 1.5, as well as the other control and sequencing aspects, including *composite* and *collection* actions.


Provide Required Part
Standards Engineer
Design Engineer
Expert Part Search
[part found]
[part not found]
Assign Standards Engineer
[stream]
[stream]
Review Requirements
Specify Part Mod Workflow
Schedule Part Mod Workflow
Clarify Requirements
Review Schedule
[reschedule]
[cancel]
[OK]
Provide addt'l part mod info
[stream]
[stream]
Execute Part Mod Workflow
[accept]
Research Production Possibility
[reject]

Decomposing an Action

- *Actions can be decomposed into subactivities(M. Flower)*.
- Note that Action is the basic unit and can not be decomposed, so we must think it as:
  - *Decomposition provide the flexibility so that user can substitute an action with subactivity at the further design steps.*
- *Actions have no further decomposition in the activity containing them.* [page 306]
  - However, the execution of a single action may *induce* the execution of many other actions.
  - For example, a call action invokes an operation that is implemented by an activity containing actions that execute before the call action completes.

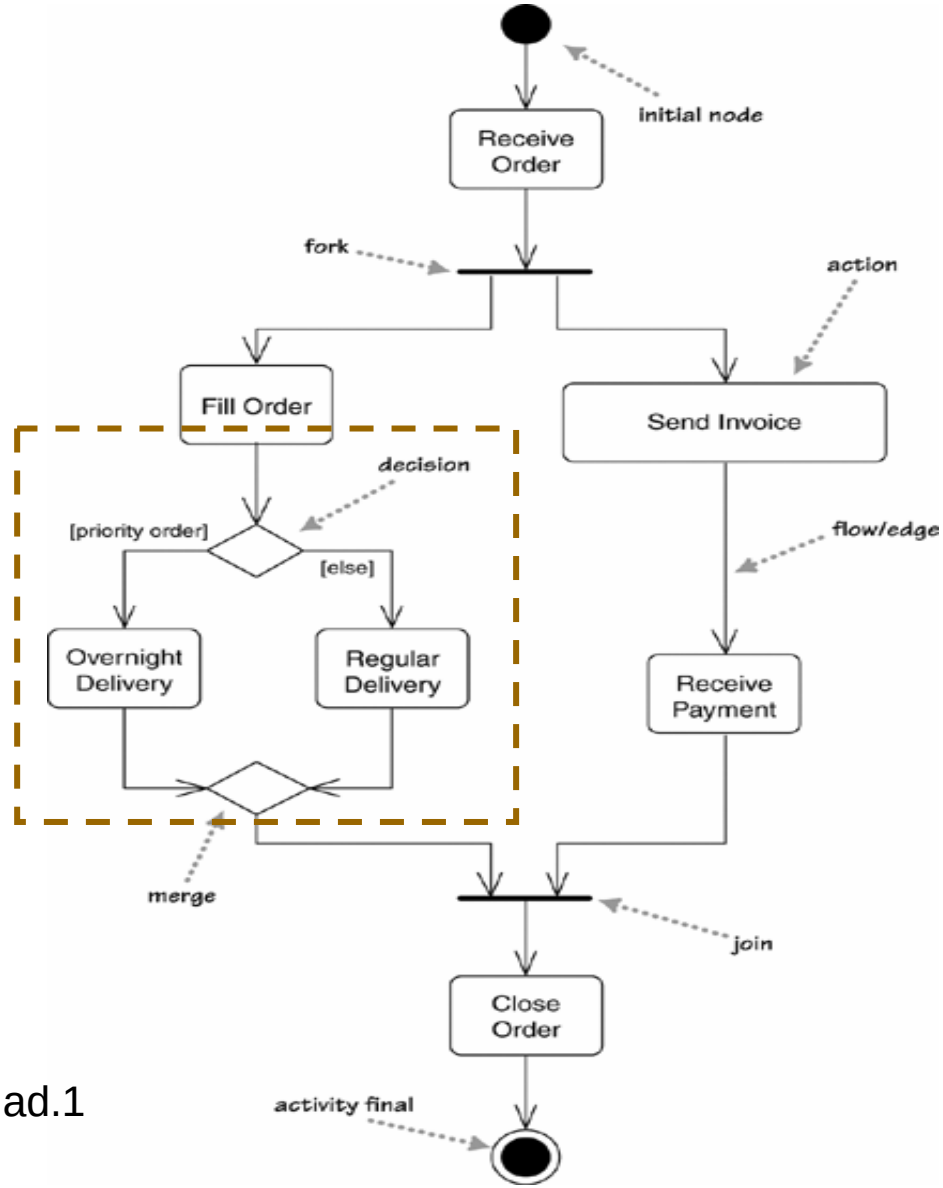
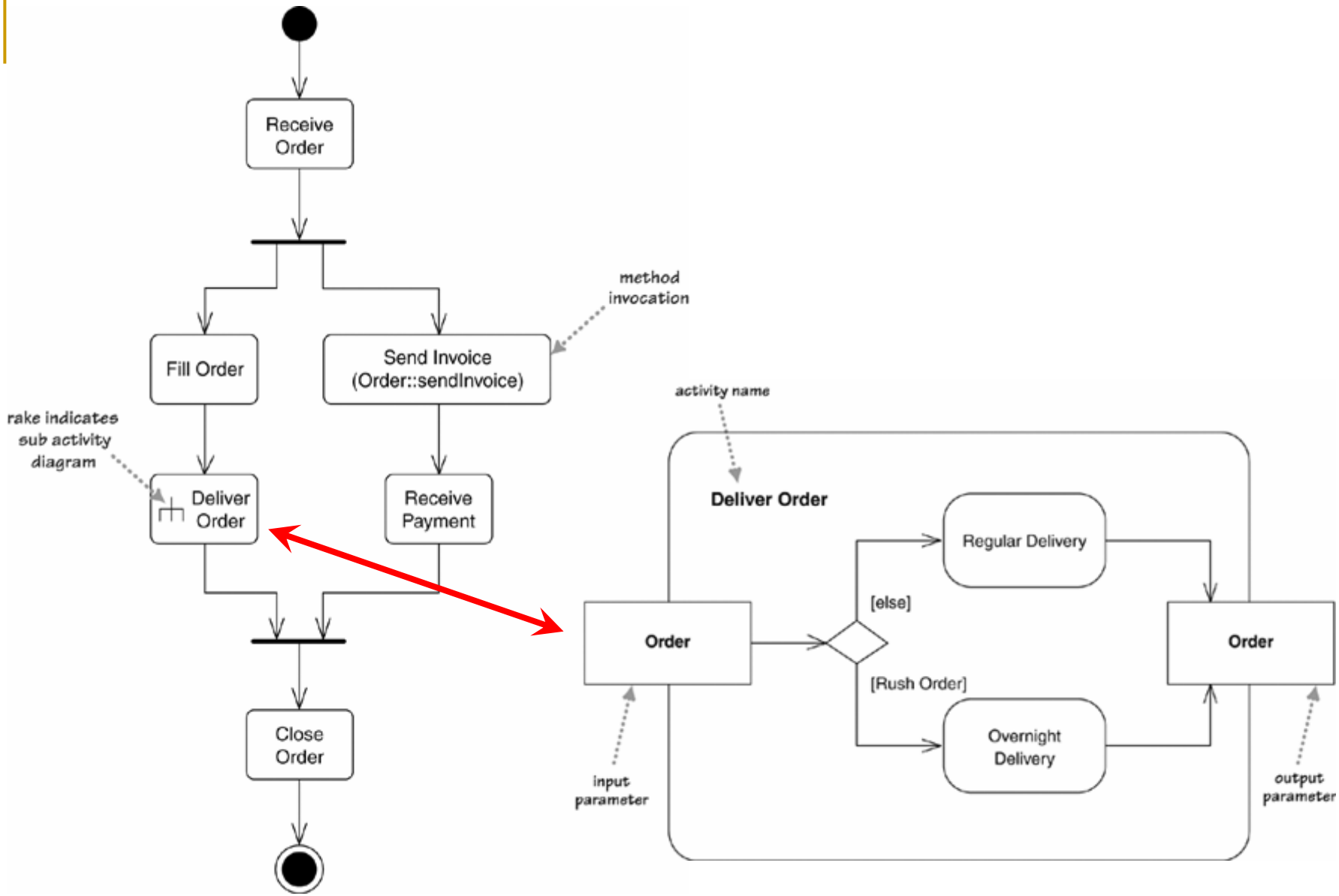

initial node
Receive Order
fork
action
Fill Order
Send Invoice
decision
[priority order]
[else]
flow/edge
Overnight Delivery
Regular Delivery
Receive Payment
merge
join
Close Order
activity final

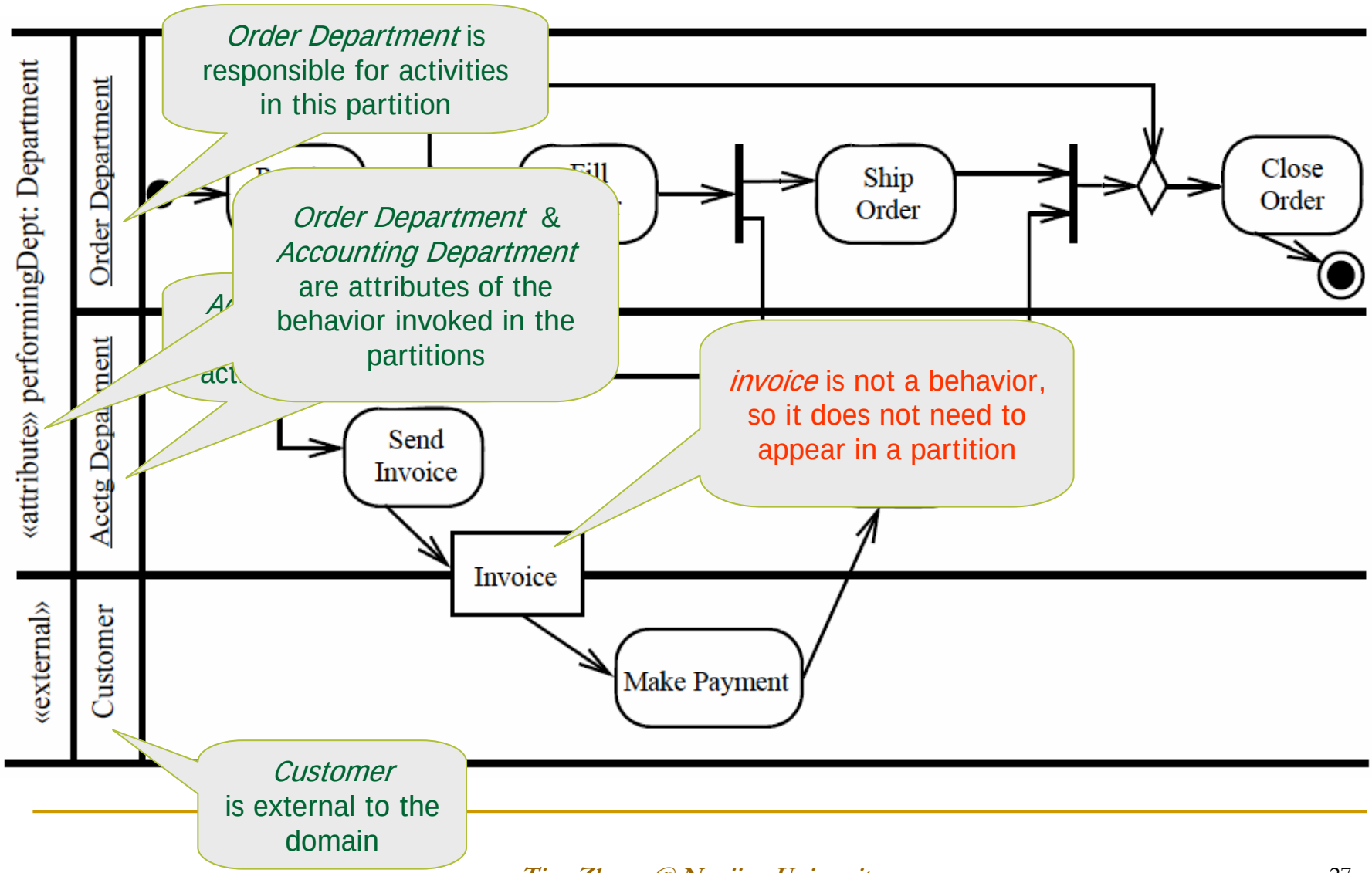
Figure ad.1


Receive Order
Fill Order
Send Invoice (Order::sendInvoice)
method invocation
rake indicates sub activity diagram
Deliver Order
Receive Payment
Close Order
activity name
Deliver Order
Regular Delivery
[else]
Order
[Rush Order]
Overnight Delivery
Order
input parameter
output parameter

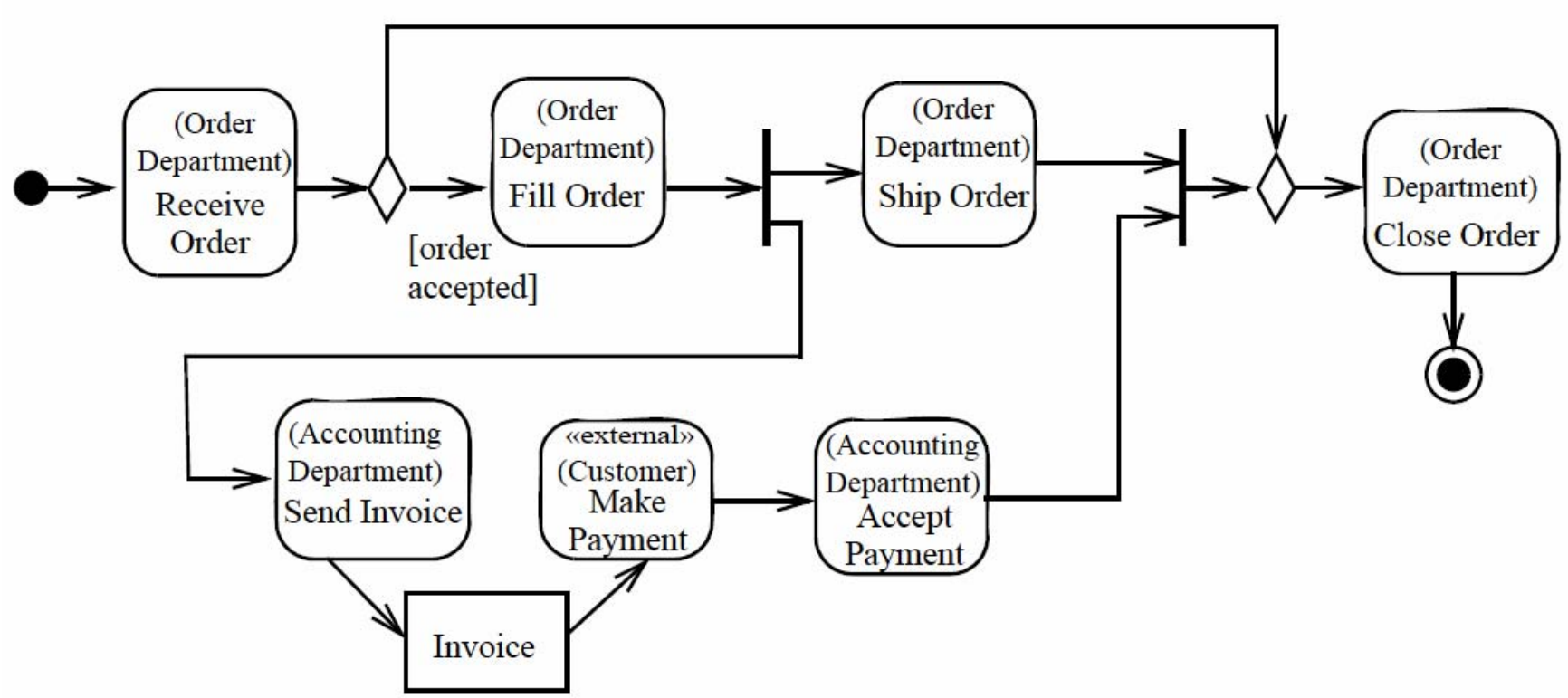
Partitions

- If you want to show who does what, you can divide an activity diagram into *partitions*
  - partitions show which actions one class or organization unit carries out.
- In UML1.x, there are only *one-dimensional* partitioning;
- In UML 2, you can use a *two-dimensional* grid.

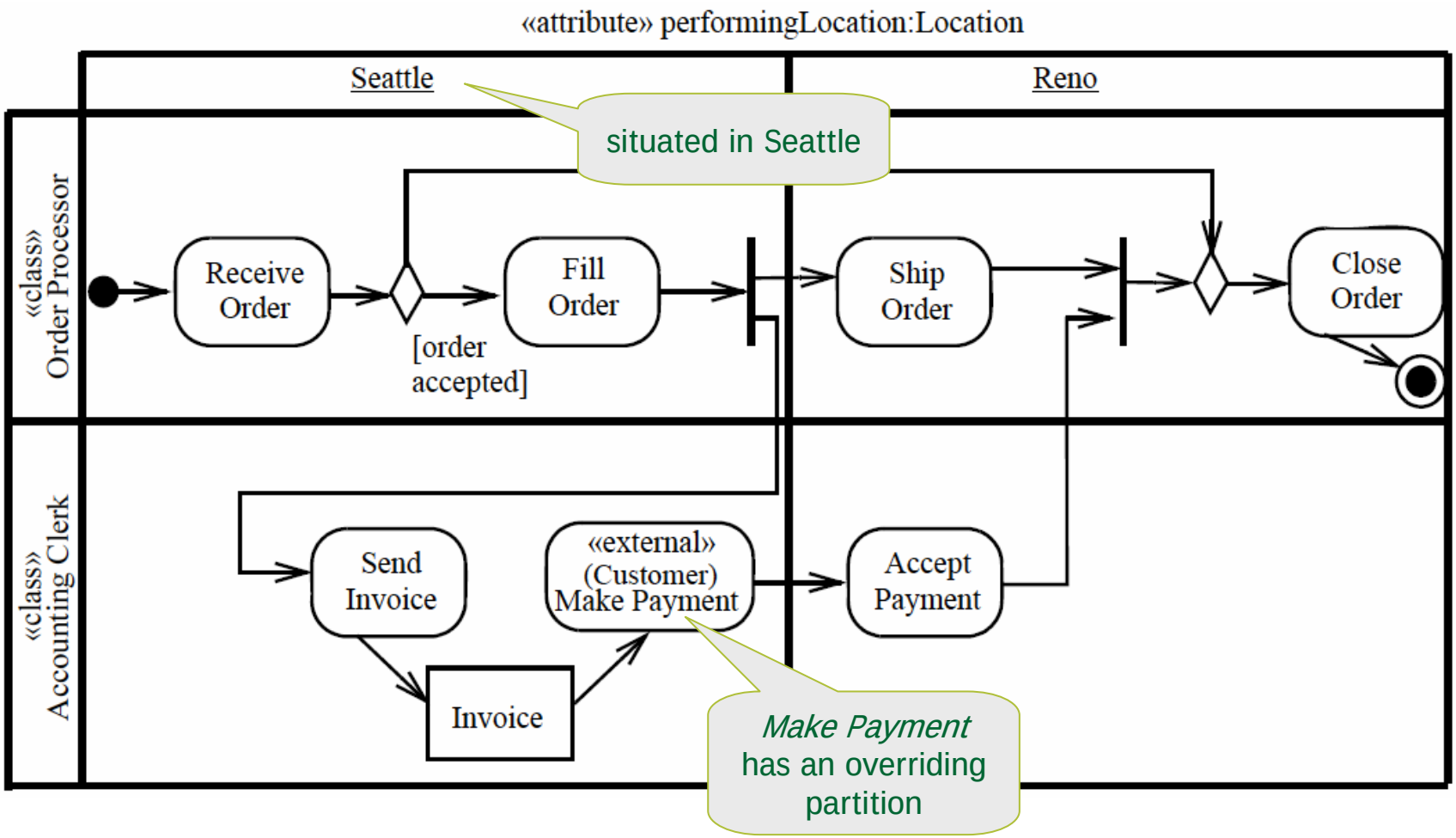
Partitions


«attribute» performingDept: Department
Order Department
Acctg Department
«external»
Customer
Ship Order
Close Order
Send Invoice
Invoice
Make Payment
Order Department is responsible for activities in this partition
Order Department & Accounting Department are attributes of the behavior invoked in the partitions
invoice is not a behavior, so it does not need to appear in a partition
Customer is external to the domain

Activity partition using annotation

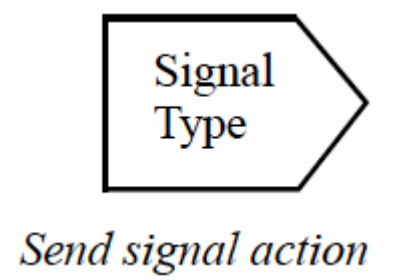


Multidimensional swim lanes

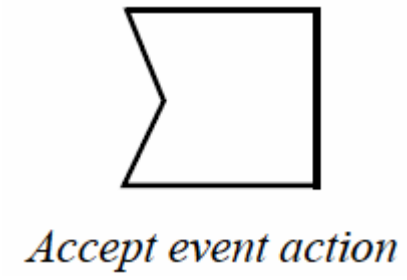


Signals

- A signal indicates that the activity receives an event from an outside process
- Signal is defined and organized in *Communications* unit
- There are two main actions defined in *Actions* unit
  - AcceptEventAction
  - SendSignalAction



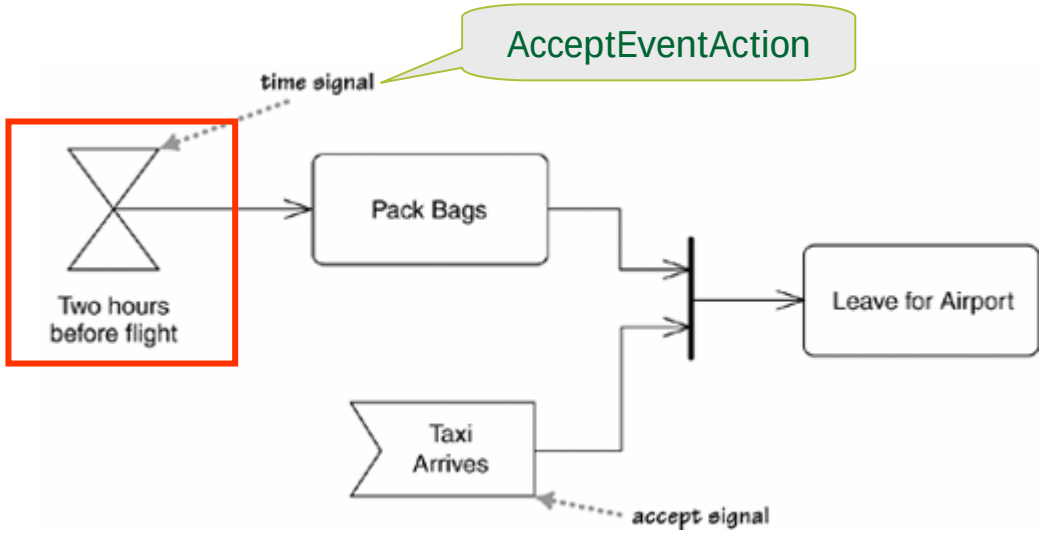
Send signal action



Accept event action

Accept time event action

Signals on an activity diagram



- In the case of above, 2 hours before my flight leaves, I need to start packing my bags.
  - If I'm quick to pack them, I still cannot leave until the taxi arrives;
  - If the taxi arrives before my bags are packed, it has to wait for me to finish before we go.

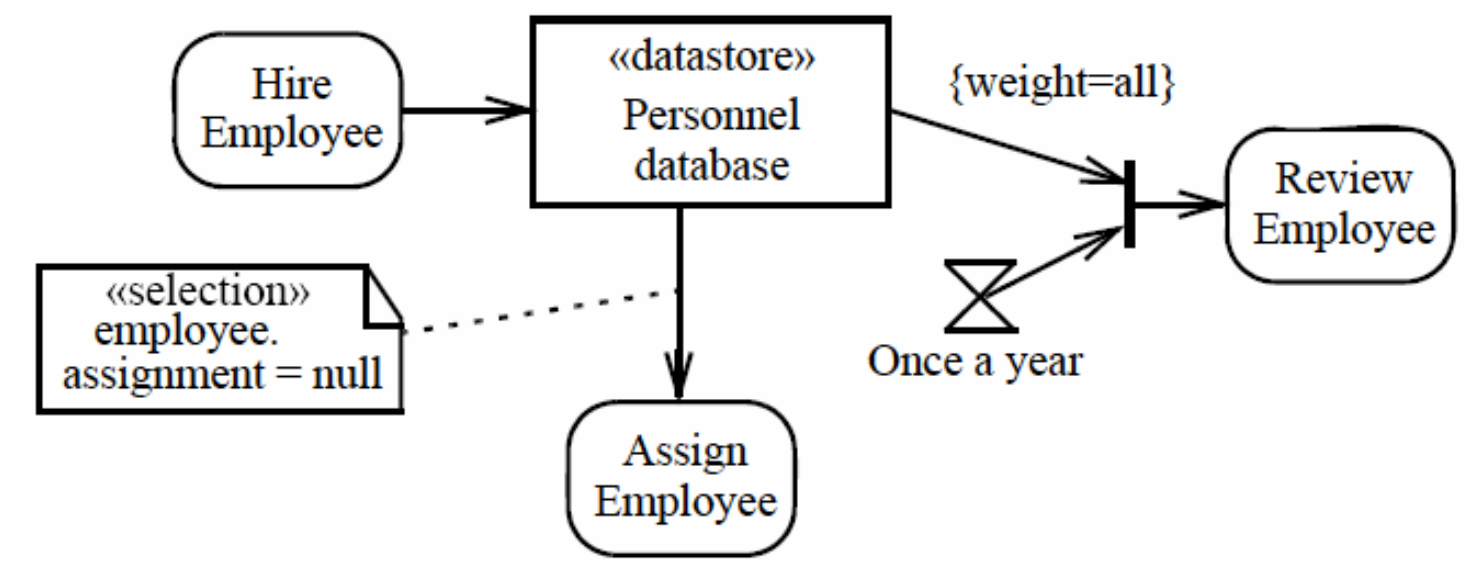
Token

- Token is introduced in UML2.0 from Petri nets
- There are many descriptions about *token* and their *production and consumption*, such as:
  - The *initial node* creates a token, which then passes to the next action, which executes and then passes the token to the next.
  - At a *fork*, one token comes in, and the fork produces a token on each of its outward flows.
  - Conversely, on a *join*, as each inbound token arrives, nothing happens until all the tokens appear at the join; then a token is produced on the outward flow.
- *Unfortunately, I haven't found separated conception describing token in the Specification.*

Data Store Node

- A data store node is a central buffer node for *non-transient* information.
- Semantics
  - Tokens chosen to move downstream are *copied* so that tokens appear to never leave the data store.
  - If a token containing an object is chosen to move into a data store, and there is a token containing that object already in the data store, then the chosen token replaces the existing one.

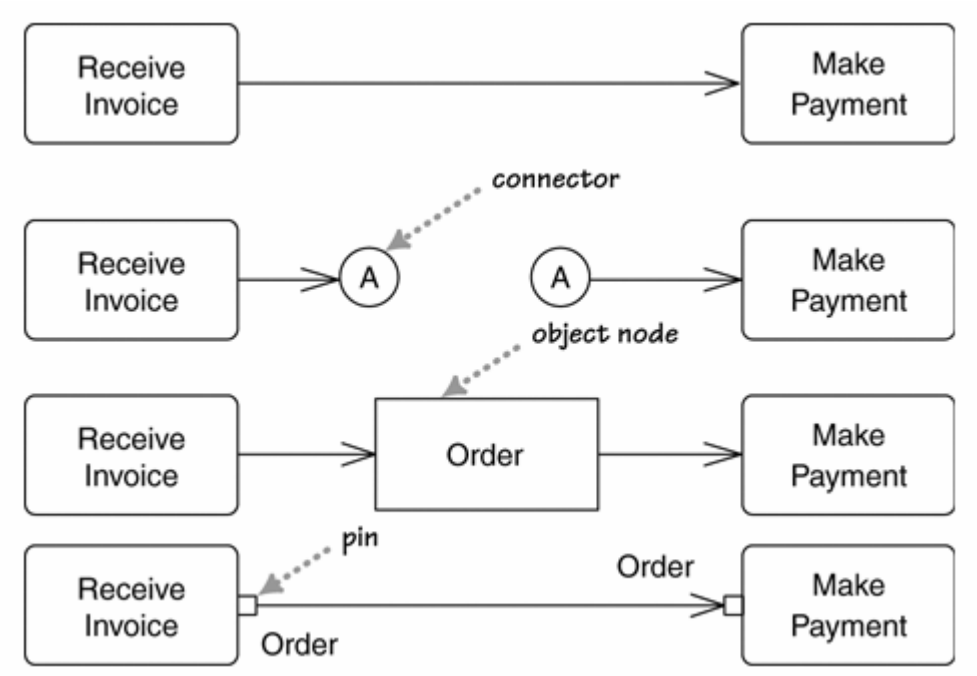
Example



Data store node example

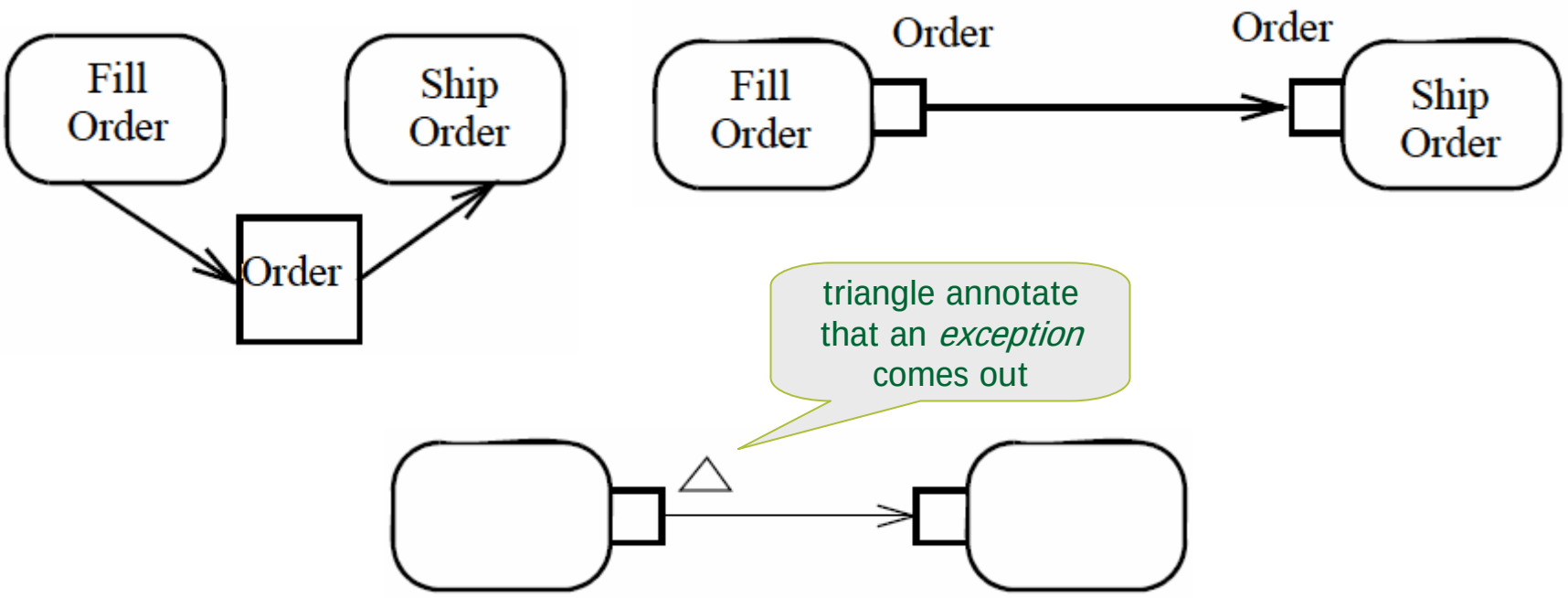
Flows and Edges

- UML 2 uses the terms flow and edge synonymously to describe the connections between two actions.
- Four ways of showing an edge:



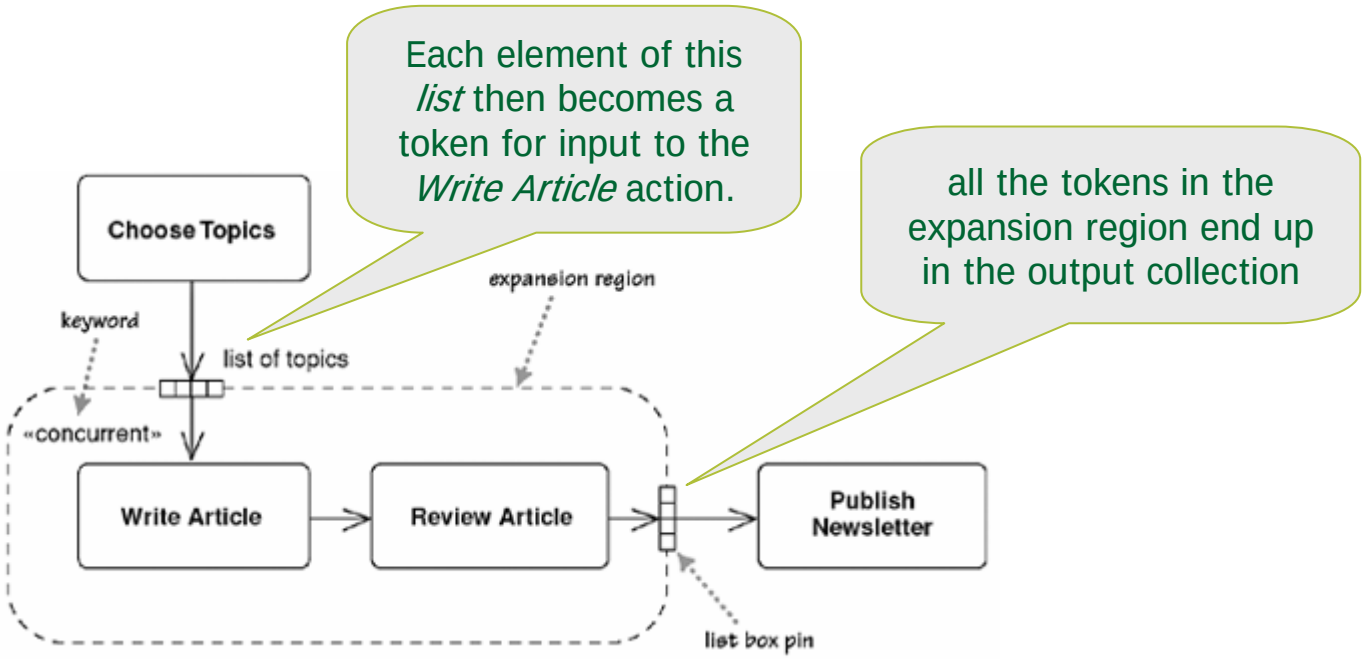
Pins

- A pin is an object node for inputs and outputs to actions.
- You can use pins to show information about *parameters* on the activity diagram

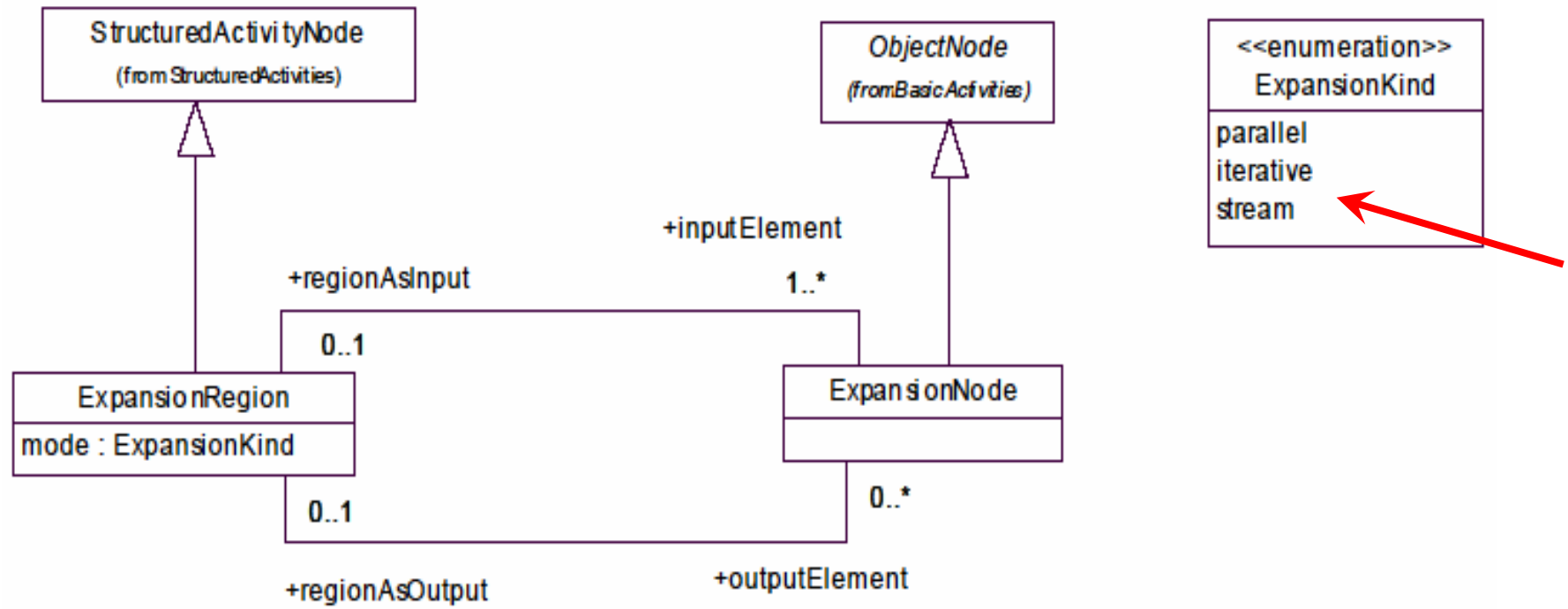


Expansion Regions

- An expansion region marks an activity diagram area where actions occur once for each item in a collection.

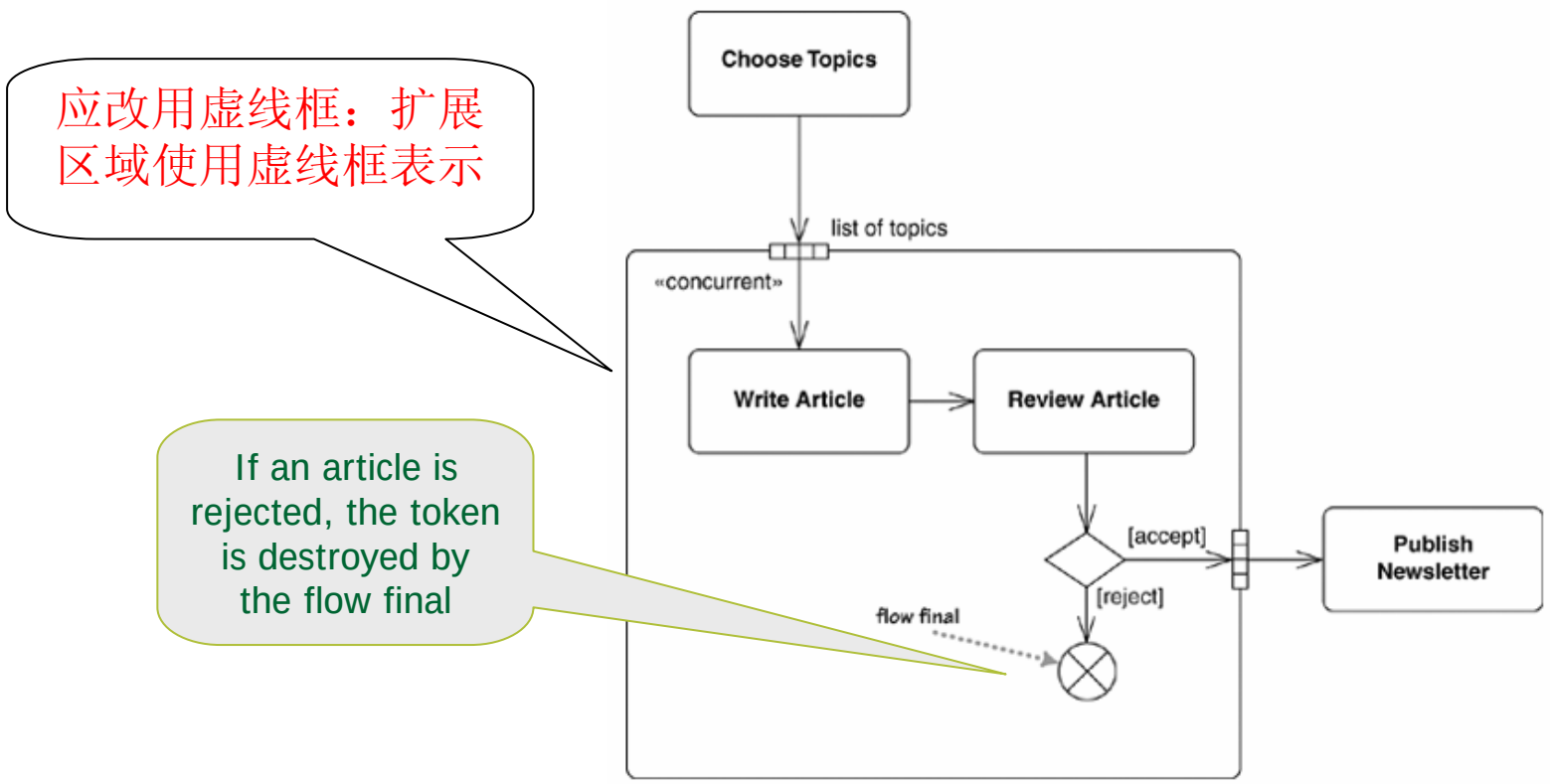


Abstract syntax of the *expansion regions*



Flow Final

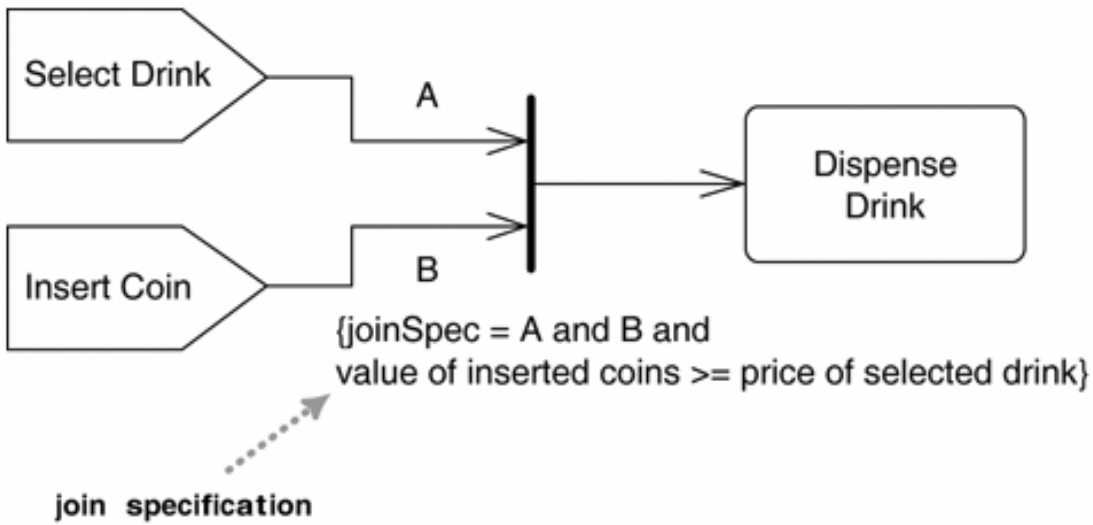
- A flow final indicates the end of one particular flow, without terminating the *whole activity*.



Join Specifications

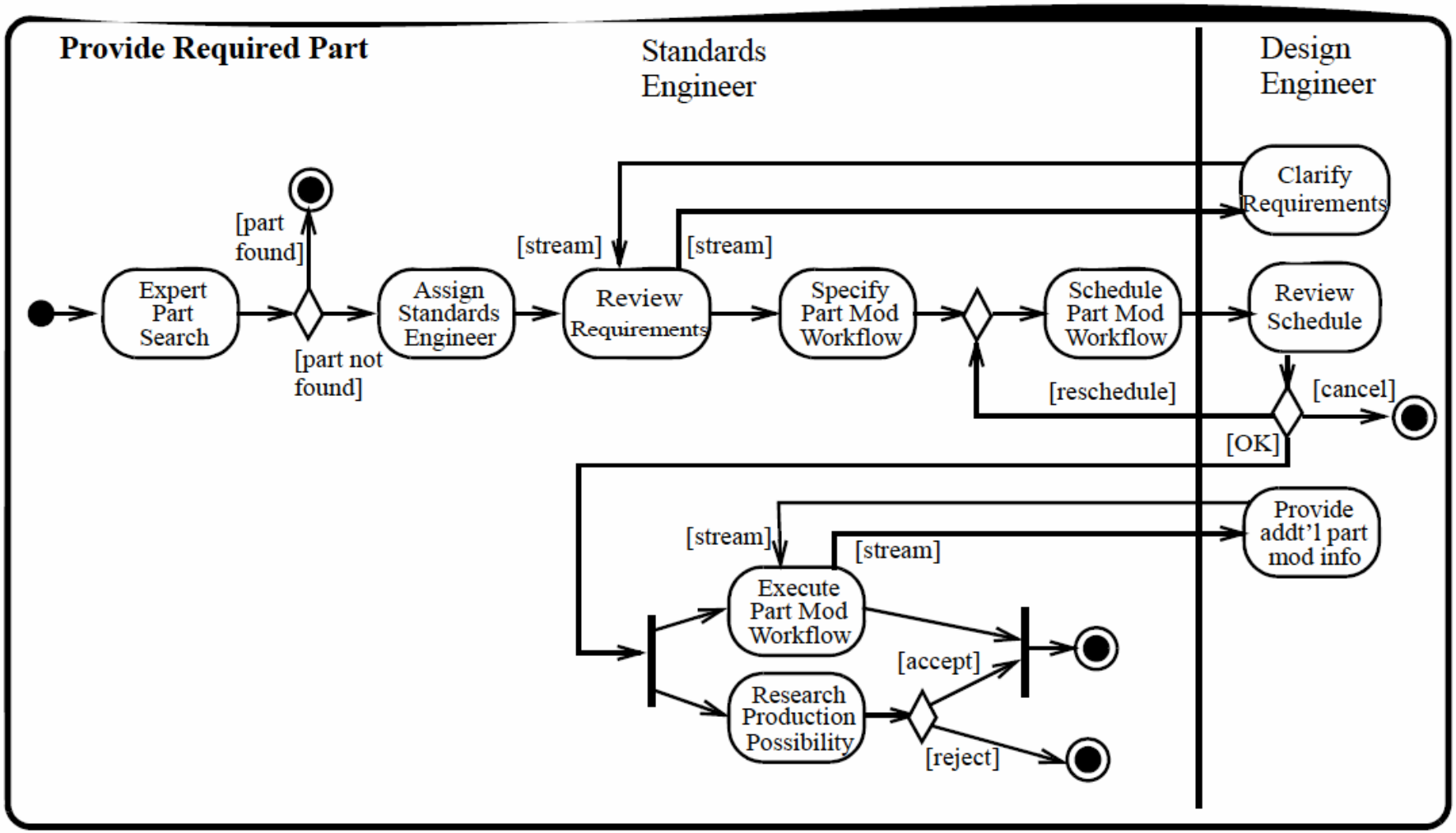
- *By default*, a join lets execution pass on its outward flow when all its input flows have arrived at the join.
- In some cases, particularly when you have a flow with *multiple tokens*, it's useful to have a more involved rule.
  - token不但可以控制并发性为，而且可以代表资源的数量
  - Activity Diagrams和Petri nets中的token含义非常相似
- A *join specification* is a Boolean expression attached to a join
  - Each time a token arrives at the join, the join specification is evaluated;
  - If true, an output token is emitted.

Example



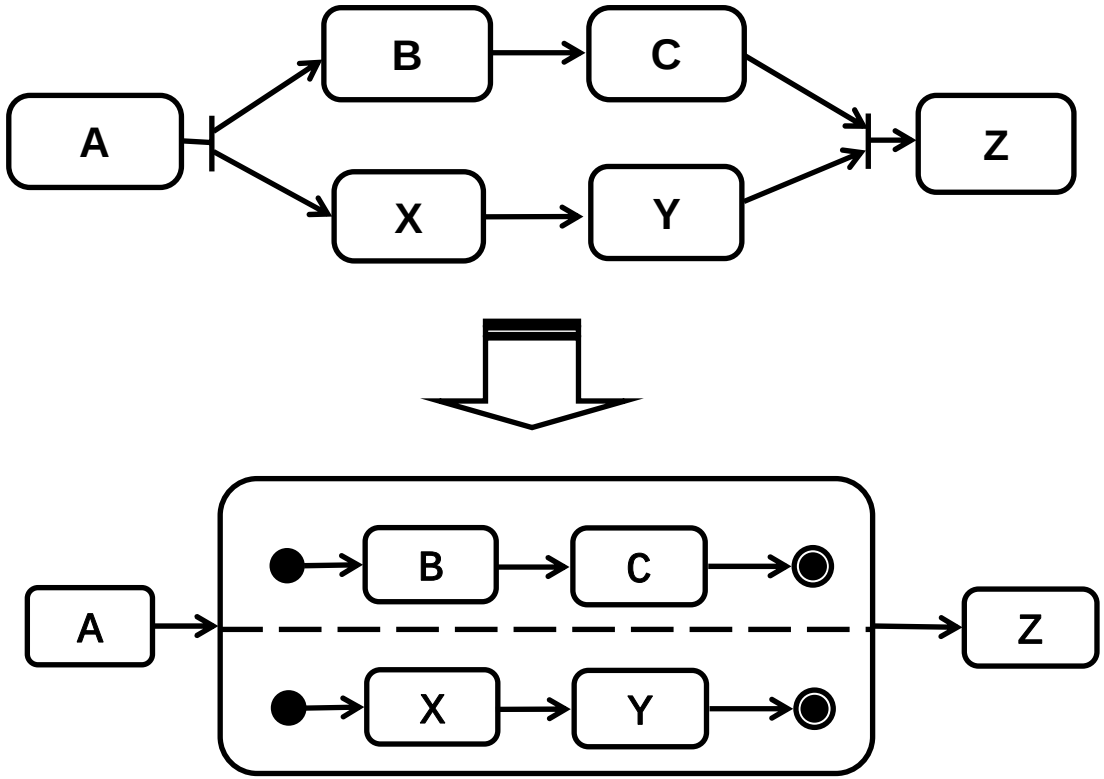
- whenever select a drink or insert a coin, the machine evaluates the join specification
- The machine slakes thirst only if putting in enough money
- This also a typical scenario for multiple tokens usage

Review


Provide Required Part
Standards Engineer
Design Engineer
Expert Part Search
[part found]
[part not found]
Assign Standards Engineer
[stream]
[stream]
Review Requirements
Specify Part Mod Workflow
Schedule Part Mod Workflow
Clarify Requirements
Review Schedule
[reschedule]
[cancel]
[OK]
Provide addt'l part mod info
[stream]
[stream]
Execute Part Mod Workflow
[accept]
Research Production Possibility
[reject]

Parallelism or Concurrency

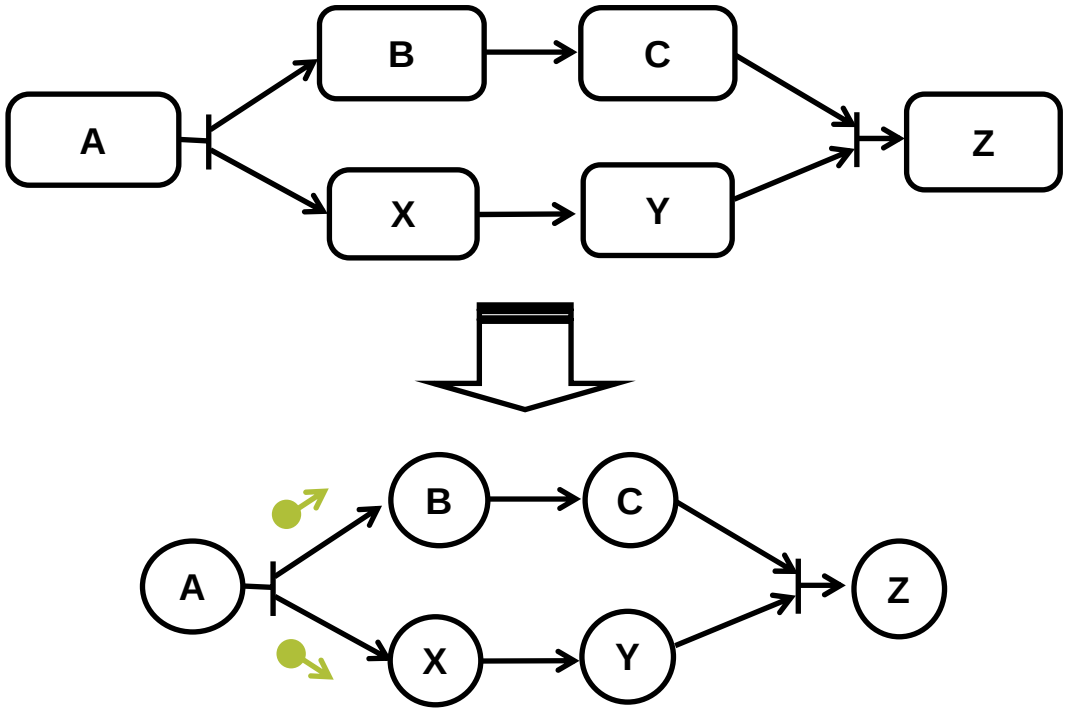
- Parallelism in UML1.5 activities:



Trace: A, B||X, C||Y, Z

Parallelism or Concurrency (2)

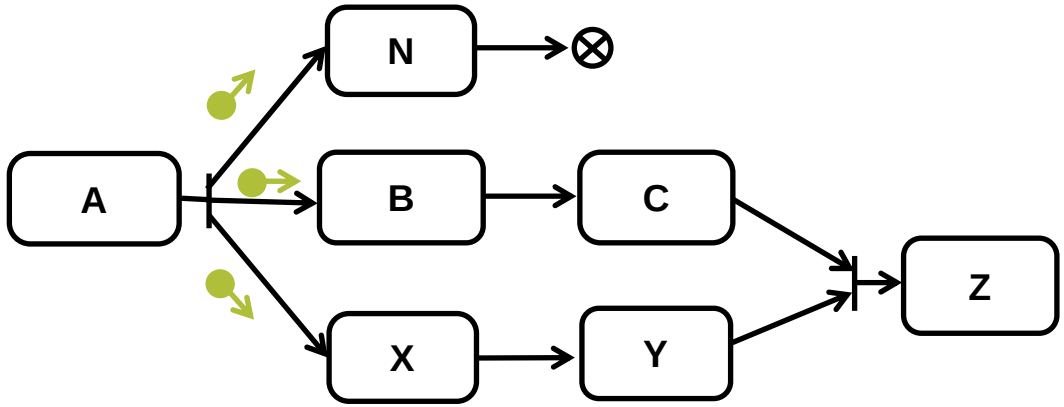
- Parallelism in UML2 activities:



Trace: A, (B,C) , Z
|| (X,Y)

Parallelism or Concurrency (3)

- Unrestricted flow patterns in UML2 activities *(U2P)* :



Trace: **A, (B,C) , Z**
|| (X,Y)
|| N

Parallelism or Concurrency (4)

- Changes from previous UML
  - Fork nodes replace the old concepts with fork kind in UML 1.5 activity modeling.
  - State machine forks in UML 1.5 required synchronization between parallel flows through the *state machine RTC* step.
  - *UML 2.0 activity forks model unrestricted parallelism*.

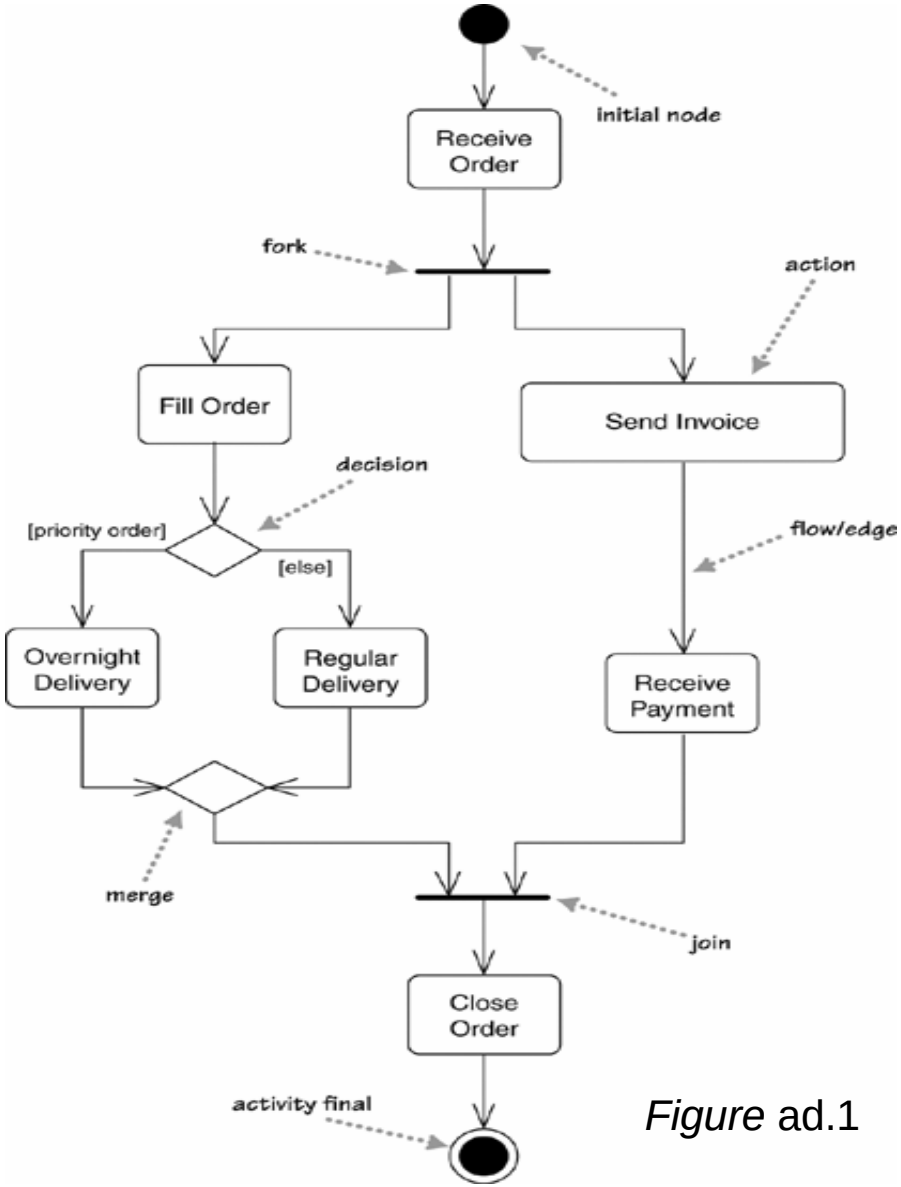


Figure ad.1

- Fill Order, Send Invoice, and the subsequent actions occur in parallel
- Essentially, this means that the sequence between them is irrelevant
  - I could fill the order, send the invoice, deliver, and then receive payment;
  - I could send the invoice, receive the payment, fill the order, and then deliver.

State Machine Modeling

- Overview
- A review of statecharts
  - conventional state machine modeling
    - two main limitations
- Harel Statecharts
  - original Harel Statecharts
  - object-based variant of Harel Statecharts
- State Machines in UML2.0
  - concepts and constructs
  - diagrams

Overview

- State Charts/Machines in both UML1.5 and 2.0 are kinds of object-based variant of *Harel statecharts*.
- State Machines are essentially *finite state-transition system* used for modeling *discrete behavior*.
- In addition to expressing the behavior of a part of the system, state machines can also be used to express the *usage protocol* of part of a system.

Review of statecharts

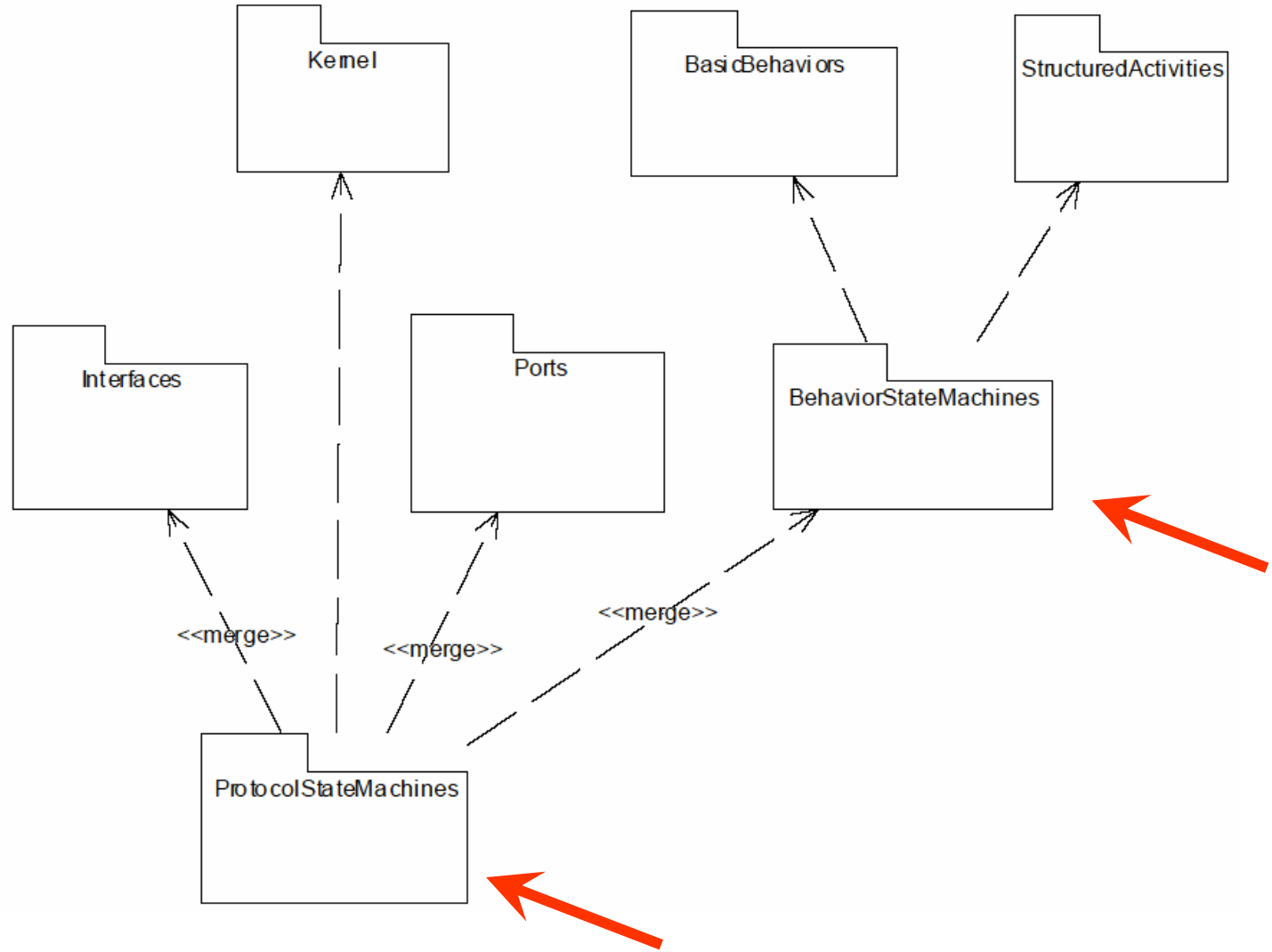
- Conventional state machine modeling techniques
  - design of discrete-event system, such as *reactive systems*
- Limitations
  - the complexity of the state diagram increases dramatically as the number of possible states increases
  - lack of support for concurrent constructs
- 可以这样理解所谓传统状态机建模技术，即在David Harel的Statecharts之前的那些状态机图，它们具有上述的两个主要的缺点。

Related work

- *finite-state machines* or *state transition diagrams*, for
  - the user interface of interactive software;
  - *the specification of data-processing systems;*
  - hardware system description;
  - *the specification of communication protocols;*
  - computer aided instruction.

State Machine Diagram

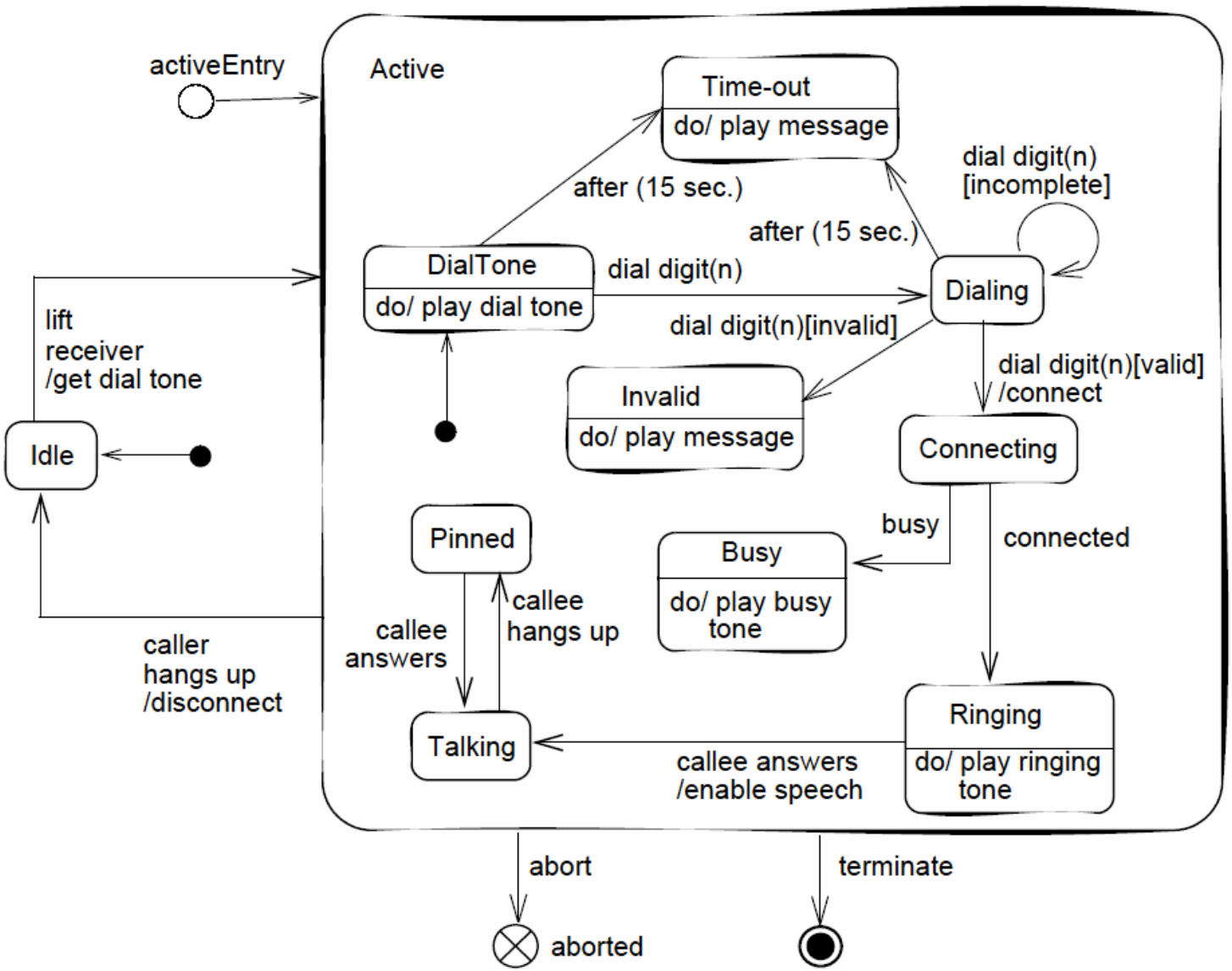
- Used for modeling discrete behavior through finite state transition systems
  - In addition to expressing the *behavior* of a part of the system, state machines can also be used to express the *usage protocol* of part of a system.
- Behavioral State Machines
  - State machines can be used to specify behavior of various model elements (e.g., class instances).
  - An object based variant of *Harel statecharts*.
- Protocol State Machines
  - Protocol state machines are used to express *usage protocols*.
  - Protocol state machines express the legal transitions that a classifier can trigger.



Package Dependencies

Change Summary

- Metamodel refactoring
- New core constructs added:
  - Fully encapsulated submachines (entry/exit points)
  - State machine specialization defined
  - State machine termination
  - Protocol State machines
    - Transitions with pre/post conditions
    - Protocol conformance between state machines
- Notational enhancements
  - Graphical notation for transitions
  - State lists


activeEntry
Active
Time-out
do/ play message
after (15 sec.)
dial digit(n)
[incomplete]
after (15 sec.)
DialTone
do/ play dial tone
dial digit(n)
Dialing
dial digit(n)[invalid]
lift
receiver
/get dial tone
Invalid
do/ play message
dial digit(n)[valid]
/connect
Idle
Connecting
Pinned
busy
connected
Busy
do/ play busy
tone
callee
hangs up
callee
answers
caller
hangs up
/disconnect
Ringing
do/ play ringing
tone
Talking
callee answers
/enable speech
abort
terminate
aborted

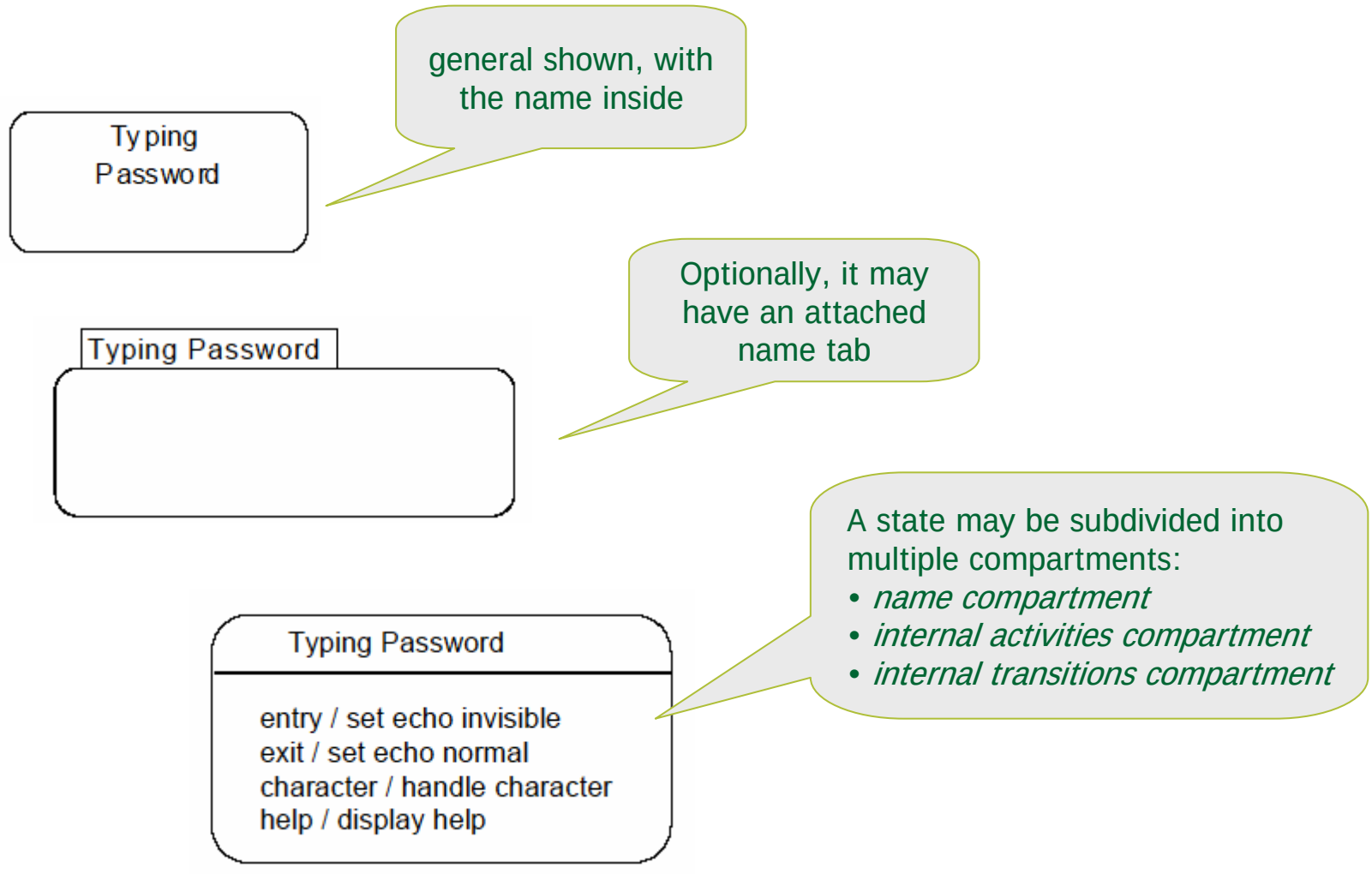
State

- Description
  - State in *Behavioral State machines*
    - A state models a situation during which some (usually implicit) invariant condition holds.
      - The invariant may represent a *static situation* such as an object waiting for some external event to occur.
      - However, it can also model *dynamic conditions* such as the process of performing some behavior
    - Three types: *simple state*, *composite state*, *submachine state*.
  - State in *Protocol State machines*
    - A protocol state represents an exposed stable situation of its context classifier
    - users can always know the state configuration.

Simple State

- Simple States are *states in general*, the following applies to states in general:
  - *Active states*
  - *State entry and exit*
  - *Behavior in state (do-activity)*
  - *Deferred events*
  - *State redefinition*

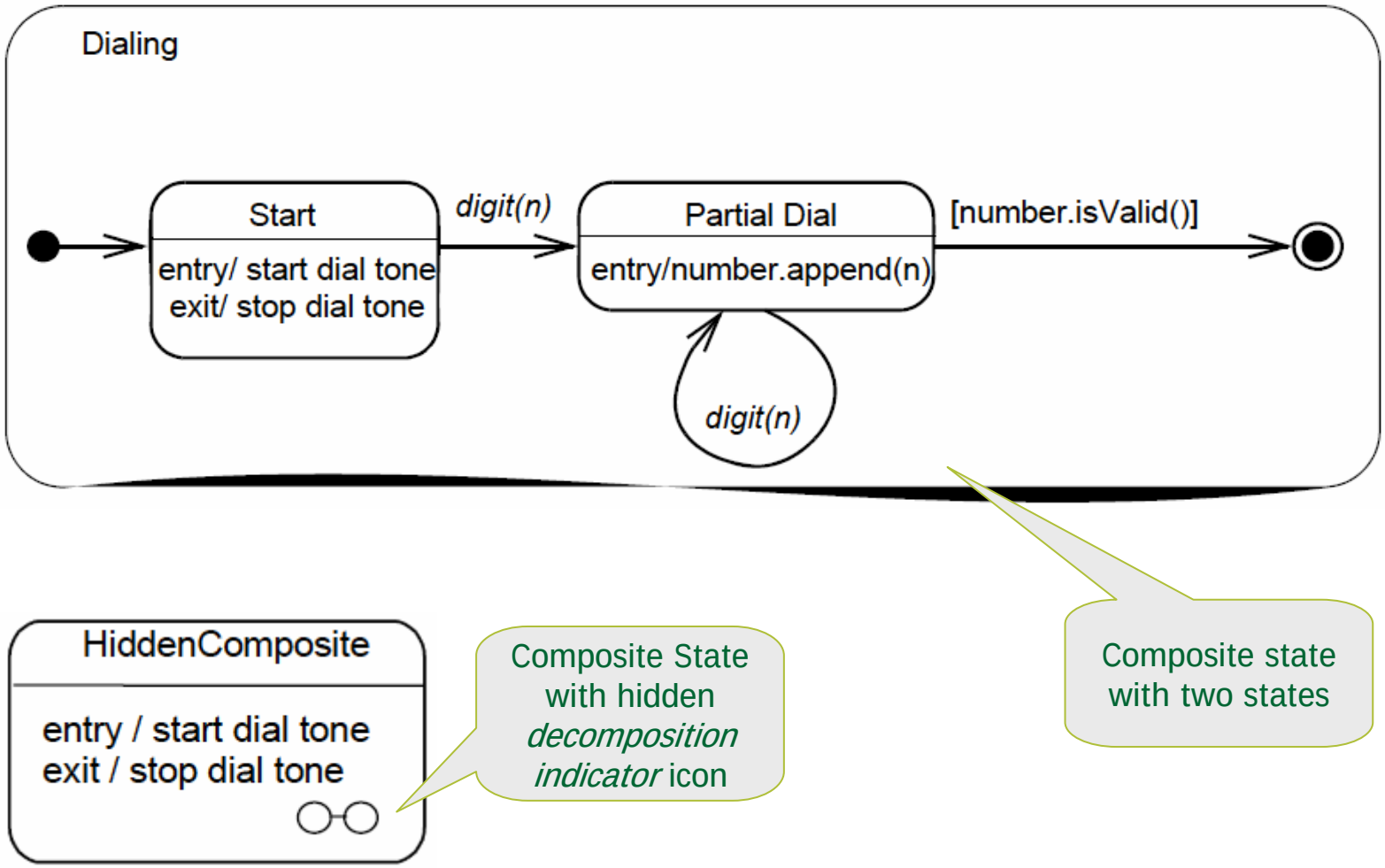
Notation

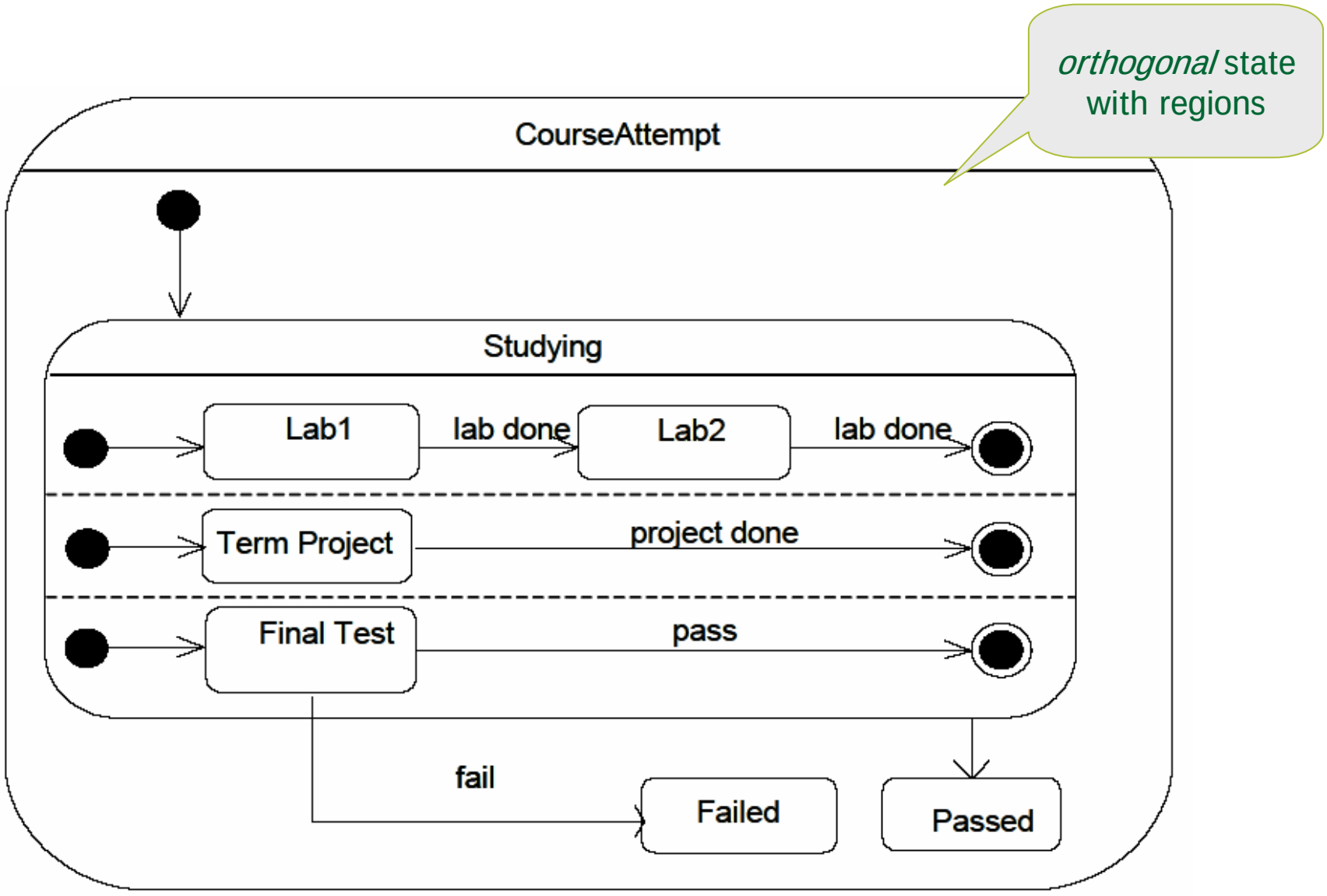

general shown, with the name inside
Typing Password
Optionally, it may have an attached name tab
Typing Password
A state may be subdivided into multiple compartments:
• name compartment
• internal activities compartment
• internal transitions compartment
Typing Password
entry / set echo invisible
exit / set echo normal
character / handle character
help / display help

Composite state

- 复合状态是对状态的一个归类，这就可以一次考虑和处理多个状态；
- 复合状态可以支持对系统行为（对象的生命期行为）进行层次化的描述；
- 复合状态可以支持逐步精化的设计思想，通过重定义进行状态拆分；
- 复合状态支持对并发性为的描述。

Notation

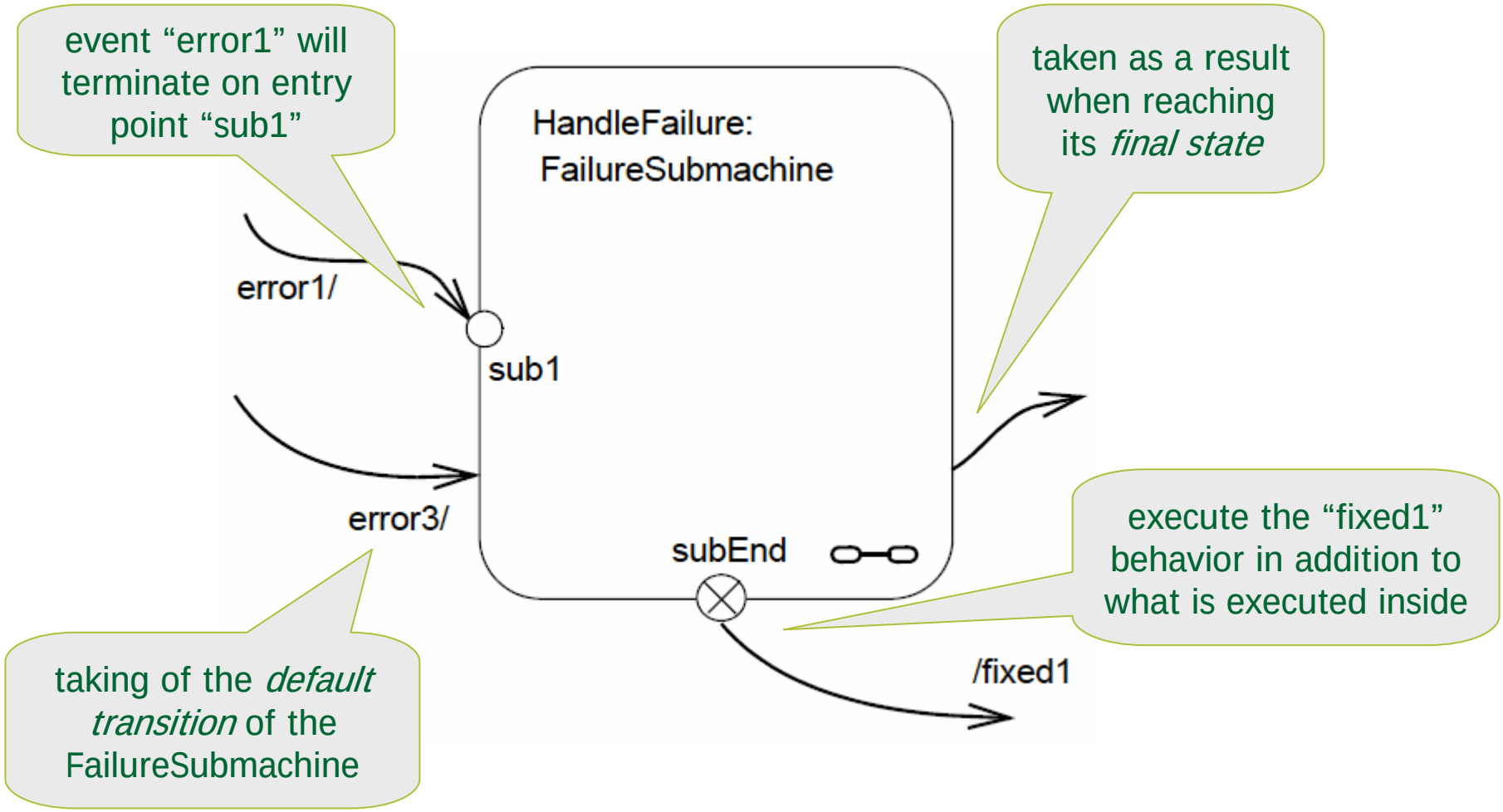

Dialing
Start
entry/ start dial tone
exit/ stop dial tone
digit(n)
Partial Dial
entry/number.append(n)
[number.isValid()]
digit(n)
HiddenComposite
entry / start dial tone
exit / stop dial tone
Composite State with hidden *decomposition indicator* icon
Composite state with two states


orthogonal state with regions
CourseAttempt
Studying
Lab1
lab done
Lab2
lab done
Term Project
project done
Final Test
pass
fail
Failed
Passed

Submachine state

- A submachine state is *semantically equivalent* to the composite state defined by the referenced state machine.
- What difference?
  - Entering and leaving this composite state is, in contrast to an ordinary composite state, via *entry* and *exit points*.
  - A submachine composite state machine can be entered via its default (initial) *pseudostate* or via any of its entry points
    - *pseudostate* has the same meaning as for ordinary composite states.

Notation


event “error1” will terminate on entry point “sub1”
taken as a result when reaching its *final state*
HandleFailure:
FailureSubmachine
error1/
sub1
error3/
subEnd
execute the “fixed1” behavior in addition to what is executed inside
/fixed1
taking of the *default transition* of the FailureSubmachine

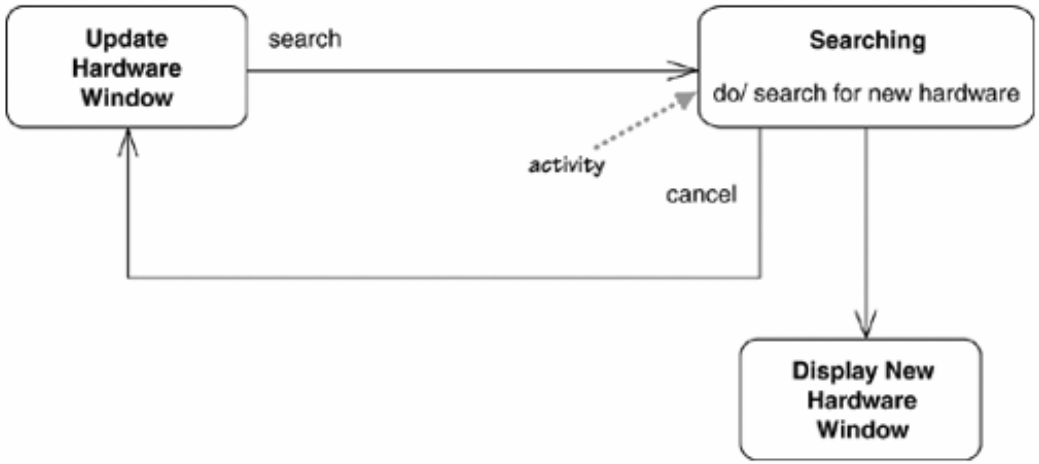
Internal Activities

- States can react to events without transition, using internal activities: putting the event, guard, and activity inside the state box itself
- An internal activity is similar to a *self-transition*
  - They are both transitions that loop back to the same state
  - However, internal activities do not trigger the entry and exit activities

Activity States

- Activity states are defined as *Internal activities*
- Activity states are such states in which the object is doing some ongoing work.
- The ongoing activity is marked with the ***do/*** ; hence the term *do-activity*.
  - ***do/*** identifies an ongoing behavior (“do activity”) that is performed as long as the modeled element is in the state or until the computation specified by the expression is completed

Example



- The Searching state in Figure is an activity state
  - Once the search is completed, any transitions without an activity, such as the one to display new hardware, are taken.
  - If the cancel event occurs during the activity, the do-activity is halted, and we go back to the Update Hardware Window state.

Pseudostate

- A pseudostate is an abstraction that encompasses different types of *transient vertices* in the state machine graph.
- Ten kinds of pseudostate:
  - An *initial* pseudostate represents a default vertex that is the source for a single transition to the *default* state of a composite state.
  - *deepHistory* represents the most recent active configuration of the composite state that directly contains this pseudostate.
  - *shallowHistory* represents the most recent active substate of its containing state (but not the substates of that substate).
  - *join* vertices serve to merge several transitions emanating from source vertices in different orthogonal regions.

Pseudostate (2)

- *fork* vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices.
- *junction* vertices are semantic-free vertices that are used to chain together multiple transitions.
- *choice* vertices which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions.
- An *entry point* pseudostate is an entry point of a state machine or composite state.
- An *exit point* pseudostate is an exit point of a state machine or composite state.
- Entering a *terminate* pseudostate implies that the execution of this state machine by means of its context object is terminated.

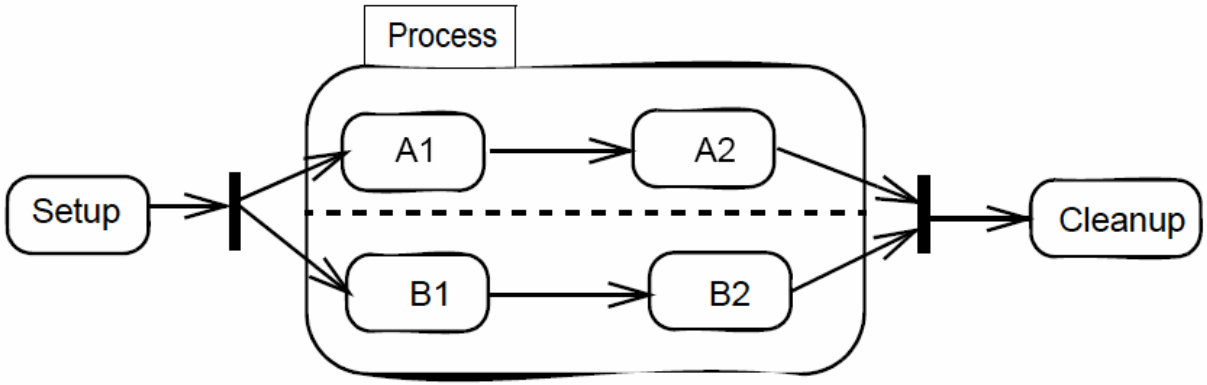
Notation

Initial Pseudo State

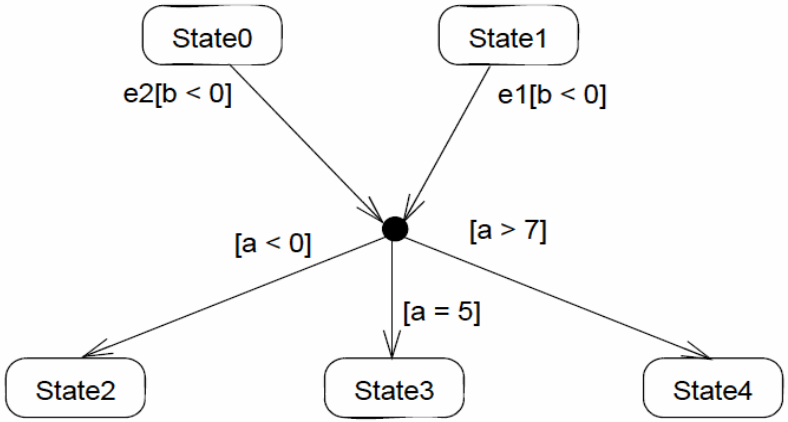
Shallow History

Deep History

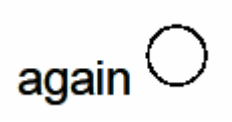
Terminate node



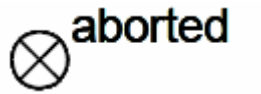
Fork and Join



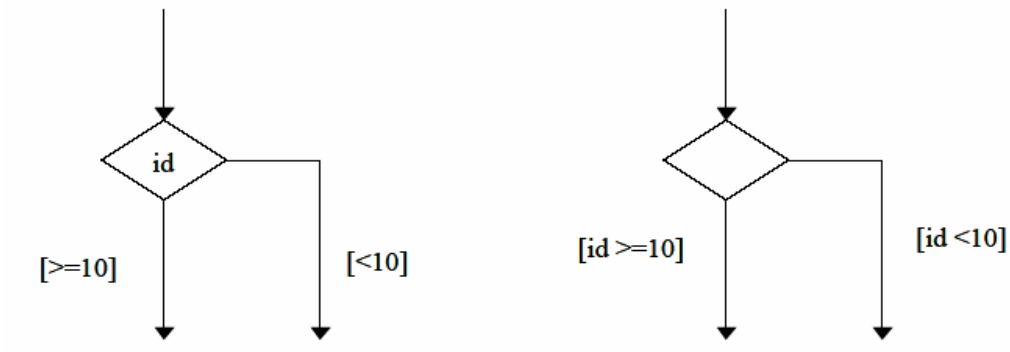
Junction Pseudo State



Entry point



Exit point



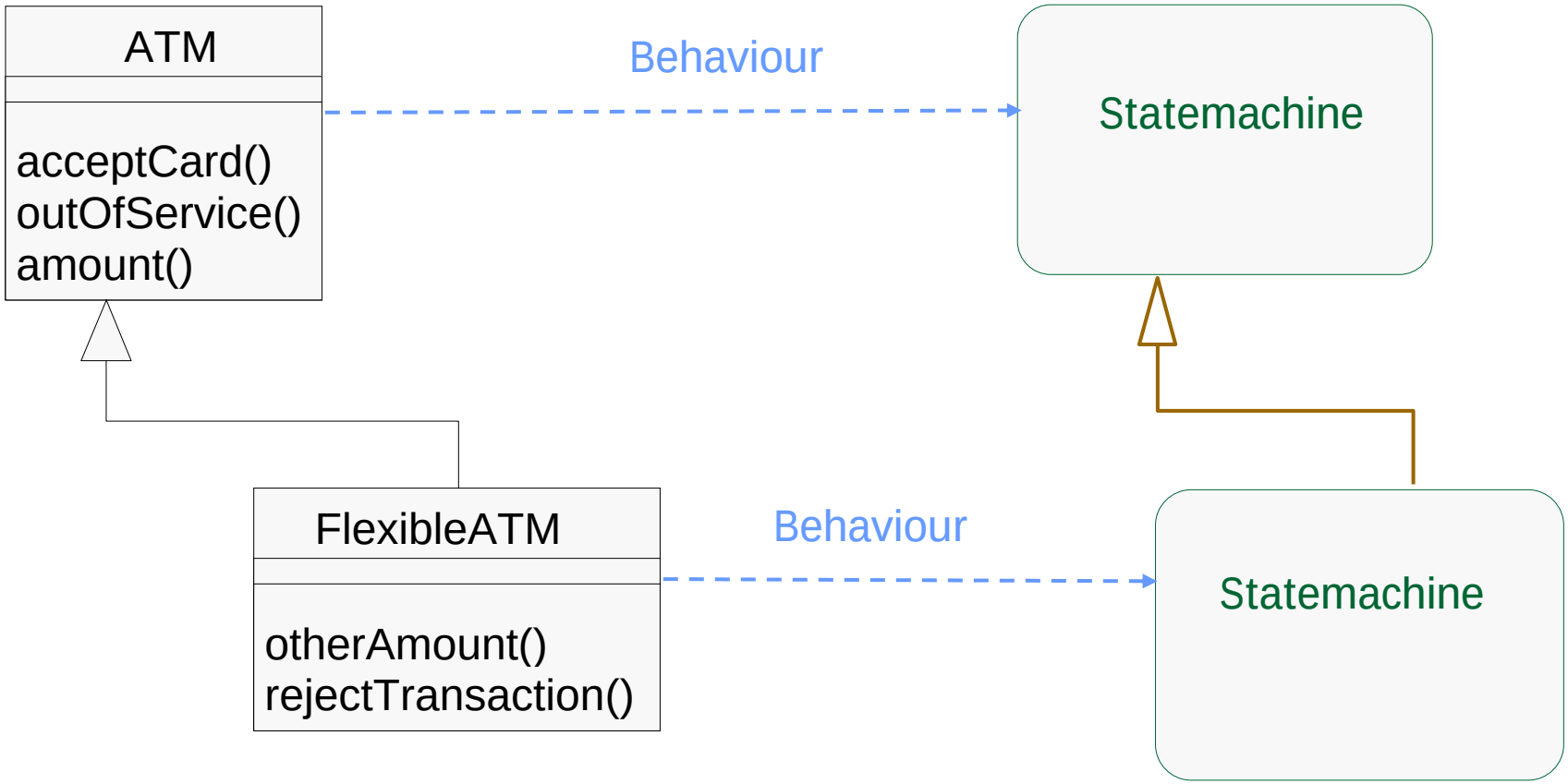
Choice Pseudo State

State redefinition

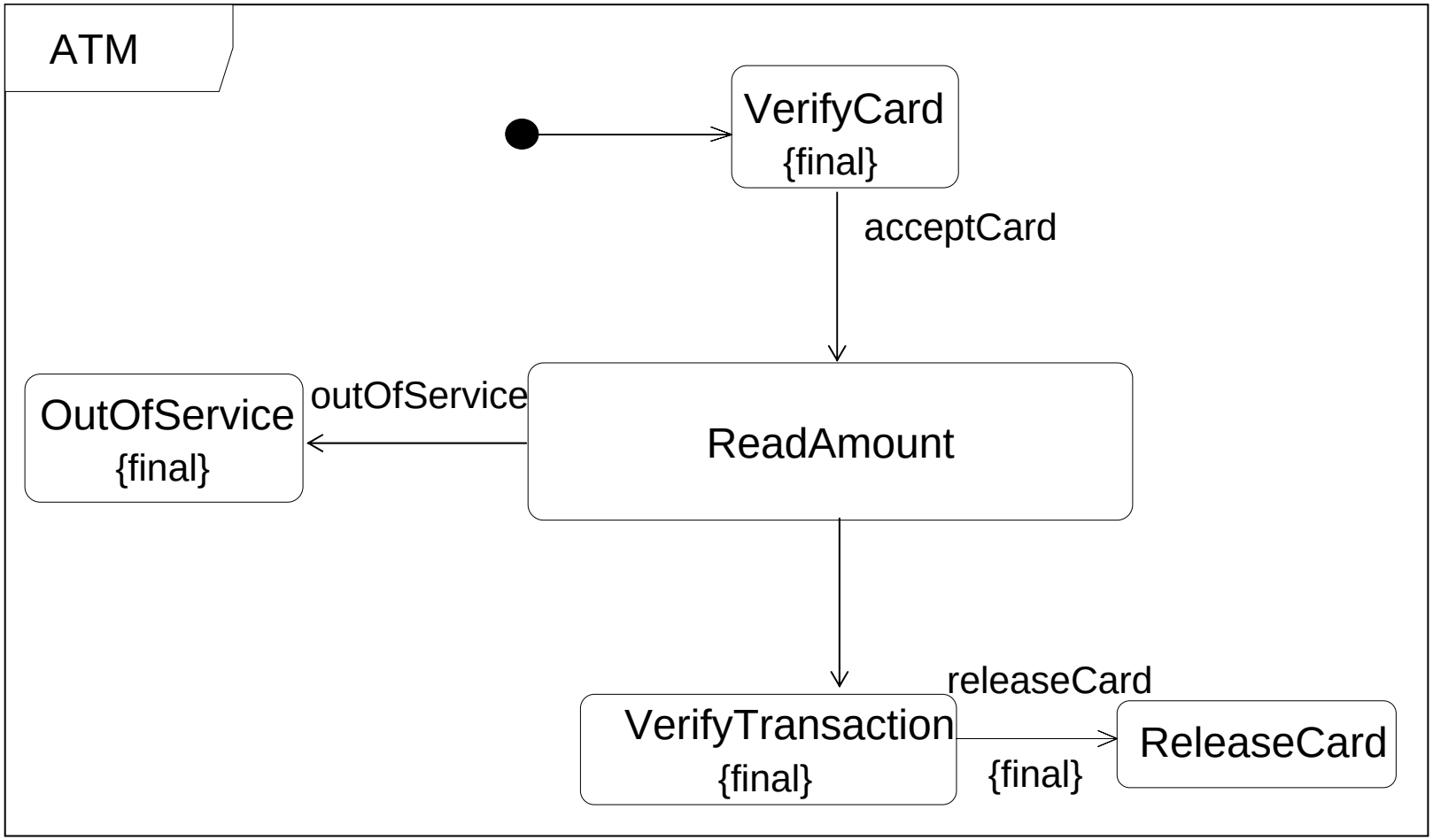
- A simple state can be redefined (extended) to become a composite state (by adding a region)
- a composite state can be redefined (extended) by adding regions and by adding vertices, states, entry/exit/do activities and transitions to *inherited regions*.
- The redefinition of a state applies to the whole state machine.
  - For example, if a state list as part of the extended state machine includes a state that is redefined, then the state list for the extension state machine includes the redefined state.

Redefinition by Specialization

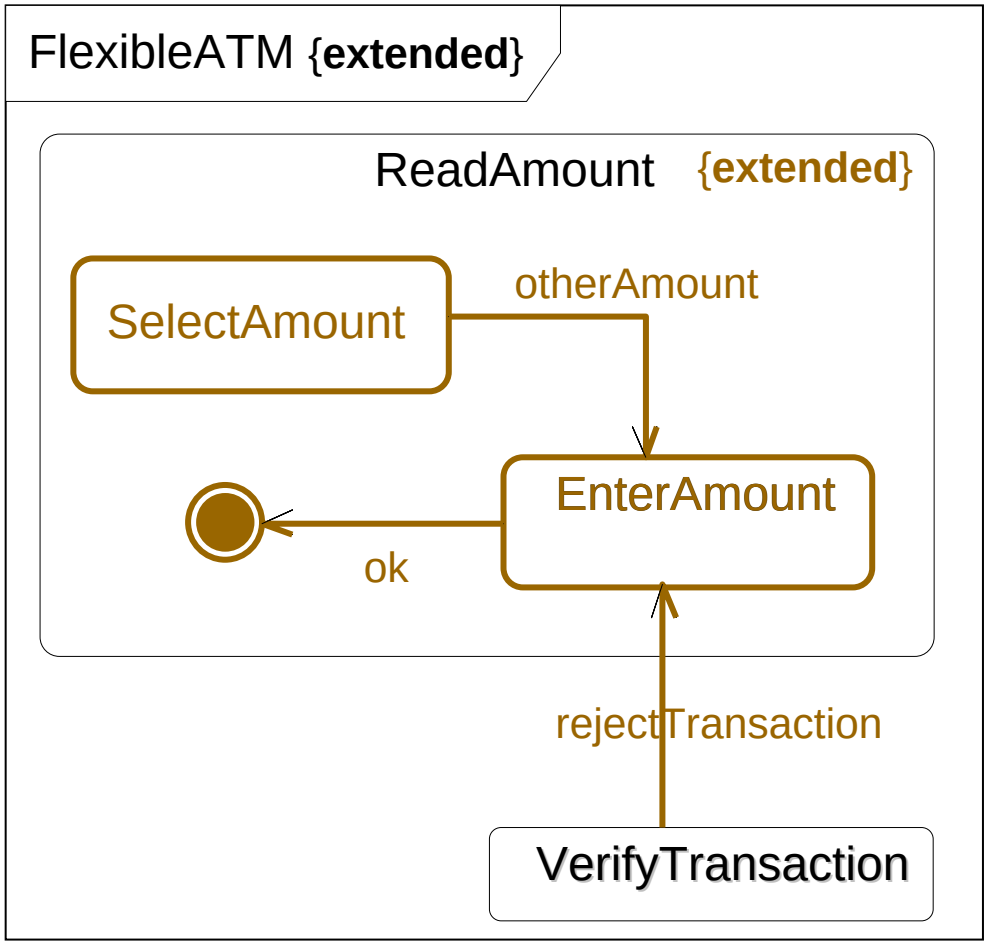
- as part of class specialization



State Machine of General Class



State Machine Specializations



- States and transitions can be added
- States can be extended
- Regions can be added, and regions can be extended
- Submachine states can be replaced
- Transitions can be replaced or extended
  - Actions can be replaced
  - Guards can be replaced
  - Targets can be replaced

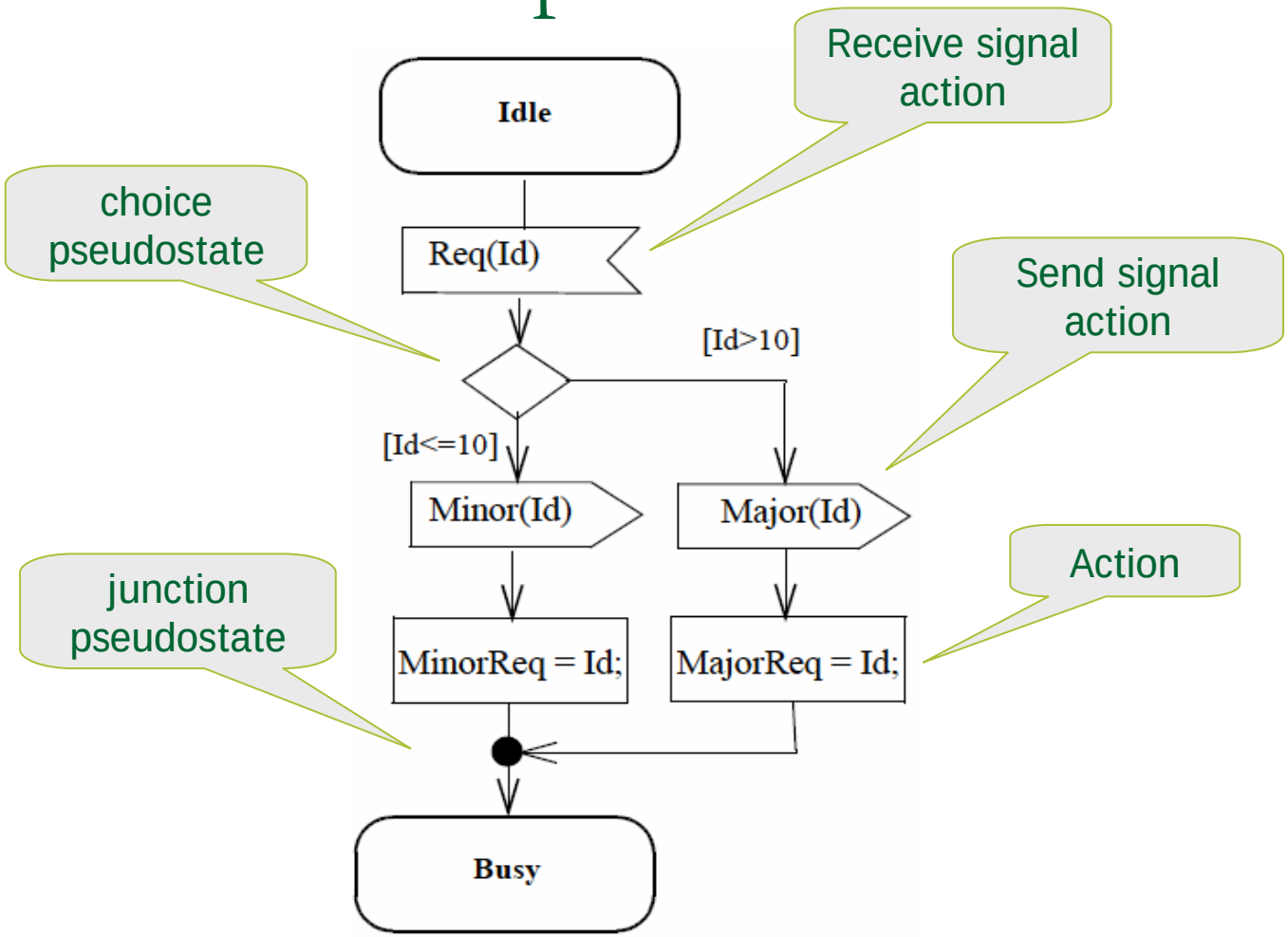
Transition

- The transition indicates a movement from one state to another.
- Each transition has a label that comes in three parts: *trigger-signature [guard] / activity*.
  - The trigger-signature is usually a single event that triggers a potential change of state.
  - The guard, if present, is a Boolean condition that must be true for the transition to be taken
  - The activity is some behavior that's executed during the transition. It may be any behavioral expression

All three parts to a transition are optional

- A missing activity indicates that you don't do anything during the transition.
- A missing guard indicates that you always take the transition if the event occurs.
- A missing trigger-signature indicates that you take the transition immediately.
  - It’s rare but does occur, which you see mostly with *activity states*

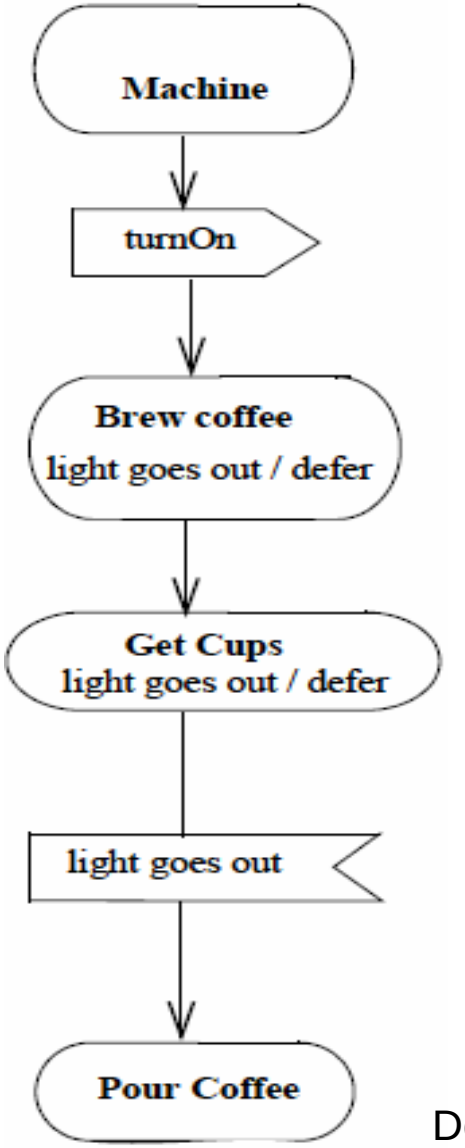
Presentation options



Symbols for Signal Receipt, Sending and Actions on transition

Deferred Events

- A state may specify a set of event types that may be *deferred* in that state.
- An event that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state.
  - Instead, it remains in the event pool while another non-deferred event is dispatched instead.
  - This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.



- If the event occurs, it is saved and it recurs when the object transitions to another state, where it may be deferred again.
- When the object reaches a state in which the event is not deferred, it must be accepted or lost.

Deferred Trigger Notation

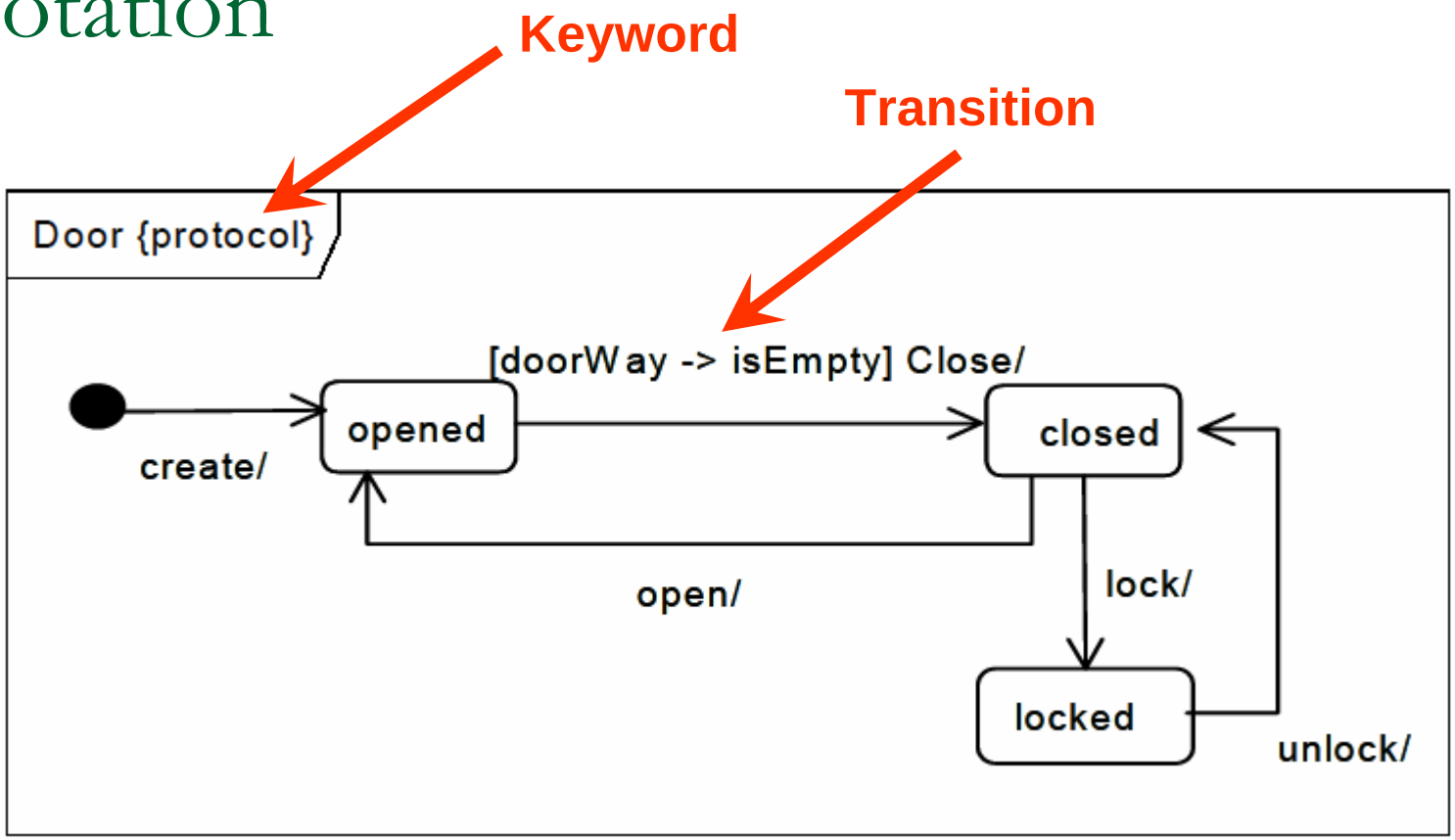
Behavioral State Machines

- There are no separate sections in Specification for the Behavioral State Machine.
- It can be seen as general state machine.
- Based on this Behavioral State Machines, the Protocol State Machine is given.

Protocol State Machine

- The states of a protocol state machine (protocol states) present an external view of the class that is exposed to its clients.
- The protocol state machine defines all allowed transitions for each operation.
  - The protocol state machine must represent all operations that can generate a given change of state for a class.
  - Those operations that do not generate a transition are not represented in the protocol state machine.

Notation


Keyword
Transition
Door {protocol}
[doorWay -> isEmpty] Close/
opened
closed
create/
open/
lock/
locked
unlock/

ProtocolTransition

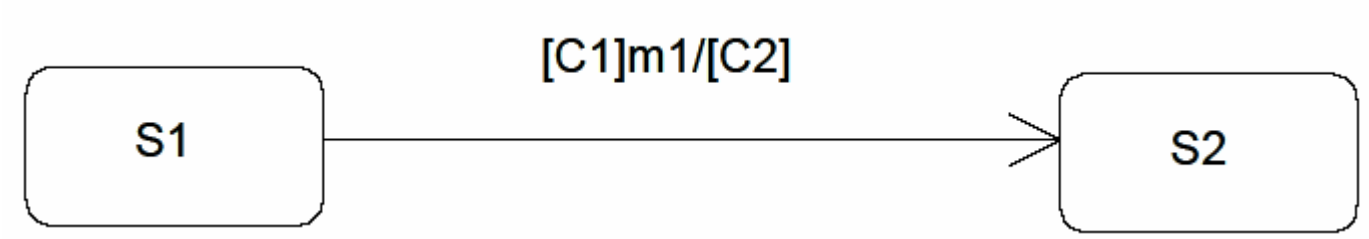
- A protocol transition specifies a legal transition for an operation.
- Transitions have the following information:
  - a pre condition (guard),
  - on trigger,
  - a post condition.
- The associated (referred) operation can be called for an instance in the origin state
- Under the initial condition (guard), and that at the end of the transition, the destination state will be reached under the final condition (post).

Notation

- The difference is that no actions are specified for protocol transitions, and that post conditions can exist.

Example

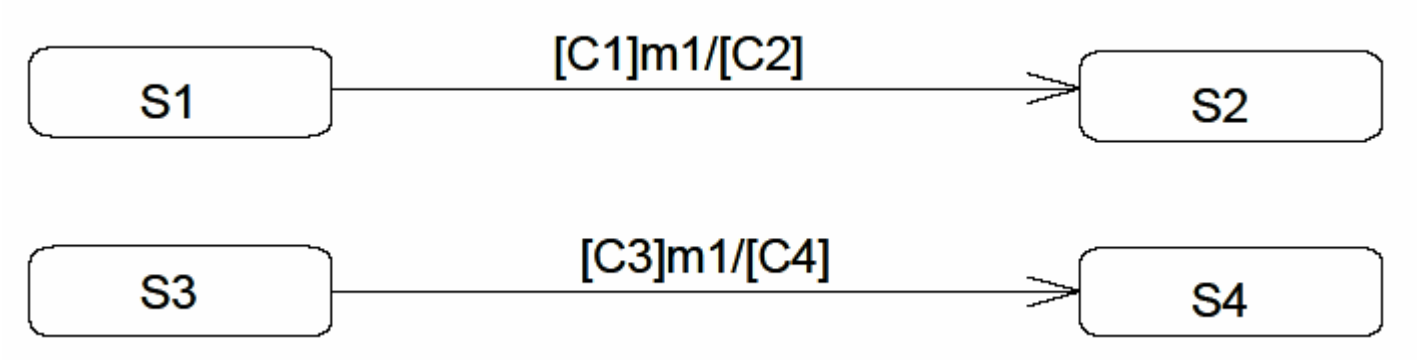
- A protocol transition can be semantically interpreted in terms of pre- and post conditions on the associated operation.
  1. The operation “m1” can be called on an instance when it is in the protocol state “S1” under the condition “C1.”
  2. When “m1” is called in the protocol state “S1” under the condition “C1,” then the protocol state “S2” must be reached under the condition “C2.”



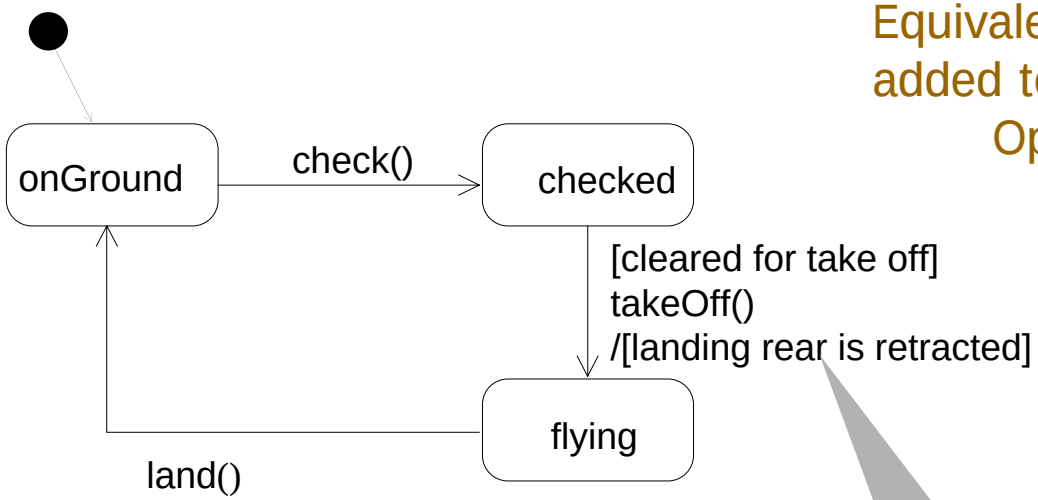
Example of a protocol transition associated to the "m1" operation

Note that

- A protocol state machine specifies all the legal transitions for each operation referred by its transitions.
- This means that for any operation referred by a protocol state machine, the part of its precondition relative to legal initial or final state is *completely* specified.



Protocol State Machines



Equivalent to pre and post conditions added to the related operations:

Operation: takeOff()

Pre

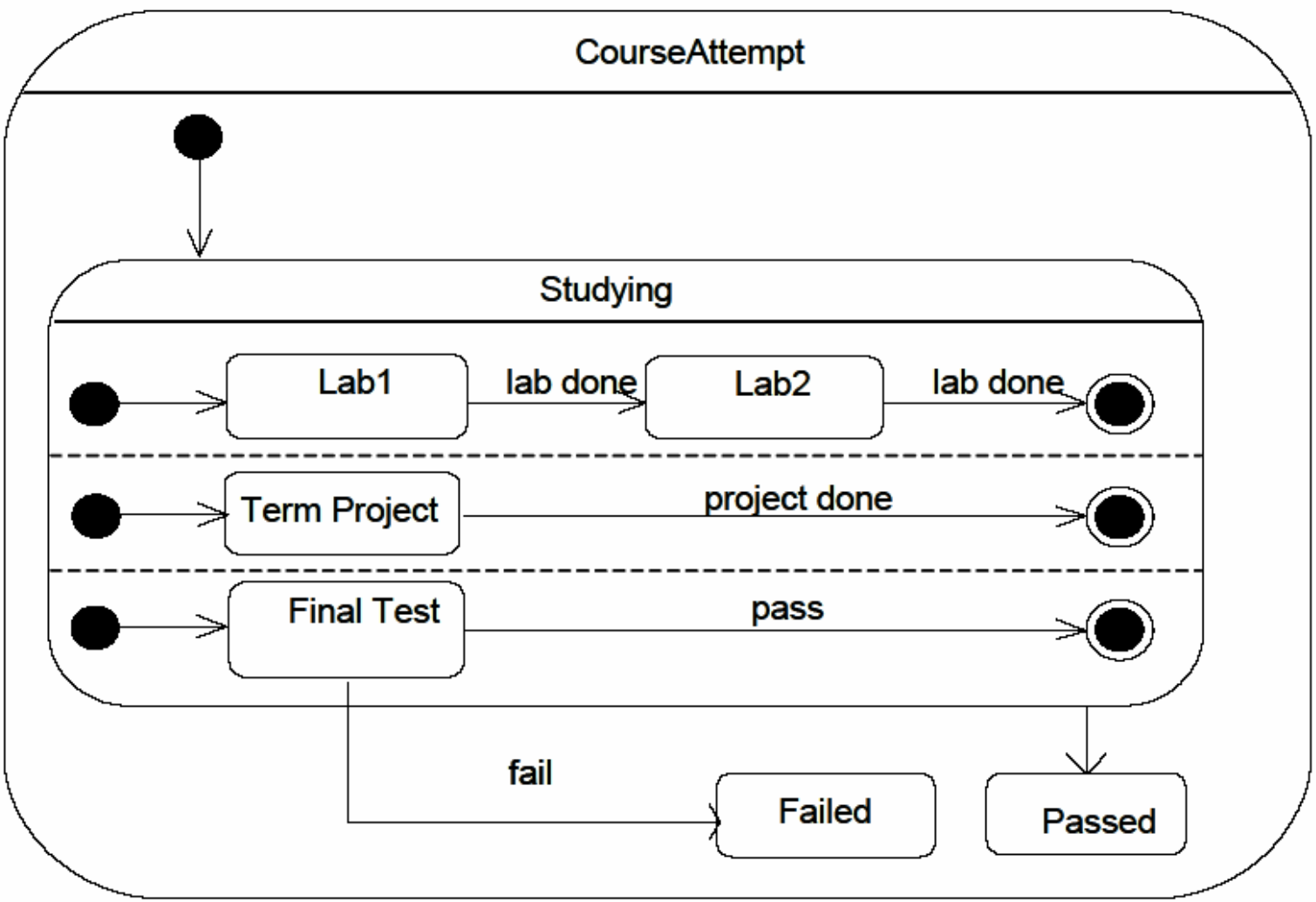
-in state "checked"
-cleared for take off

Post

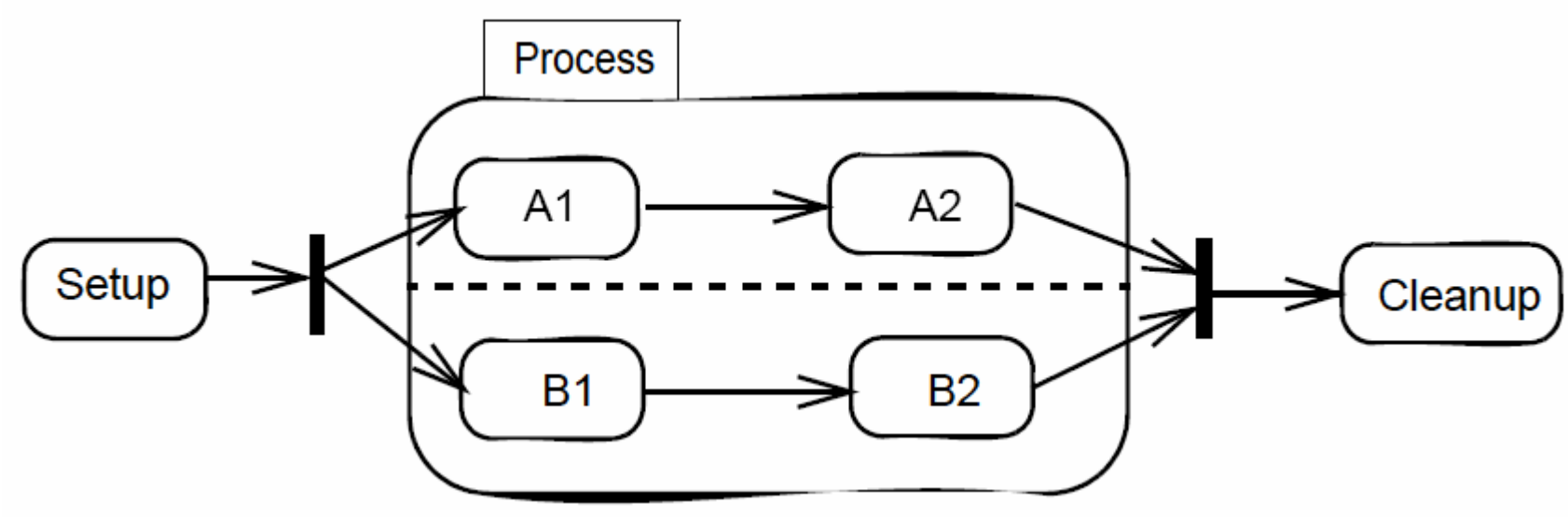
-landing rear is retracted
-in state "flying"

postcondition instead of action

Concurrency in State Machine



Concurrency in State Machine (2)

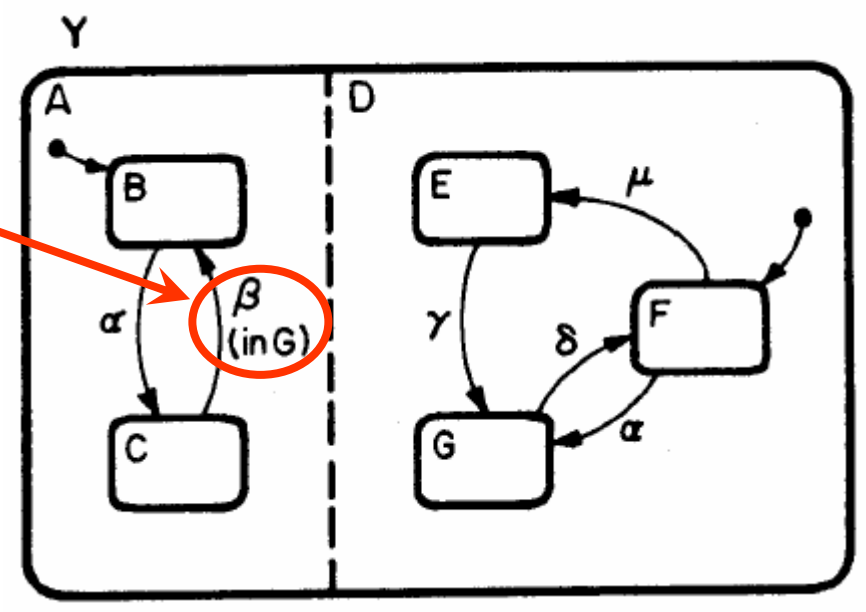


Harel Statecharts

- A broad extension of the conventional formalism of state machines and state diagrams
- Harel Statecharts extend conventional state-transition diagrams with essentially three elements, dealing with the notions of *hierarchy*, *concurrency* and *communication*.
- In a nutshell, one can say:

statecharts = state-diagrams + depth
+ orthogonality + broadcast-communication

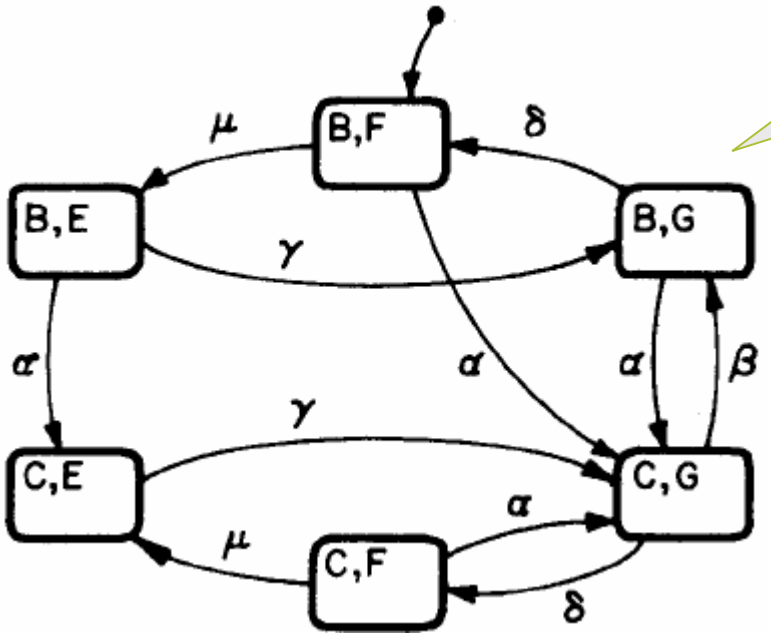
Concurrency by Orthogonality



AND decomposition

- Being in Y entails being in some combination of *B or C with E, F or G*
  - Entering Y by default is actually entering the combination (B, F) by the default arrows.

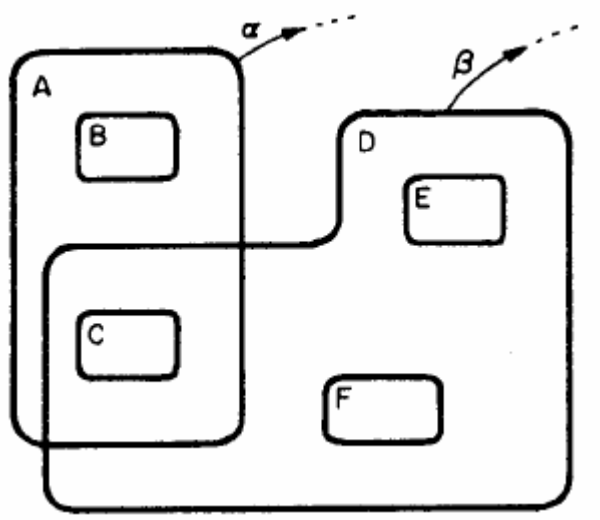
AND-free equivalent



B or C with E, F or G

- This is the equivalent to the last orthogonal diagram
  - It's hard to read with more states and transitions

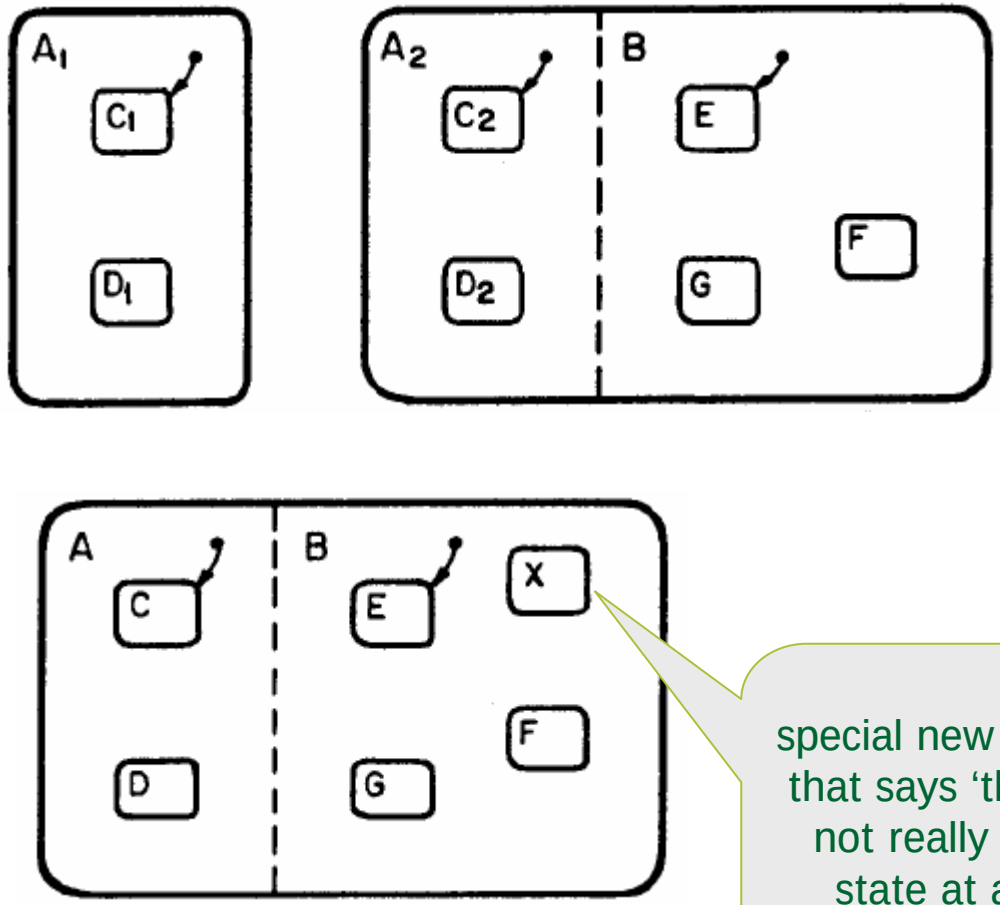
Overlapping states



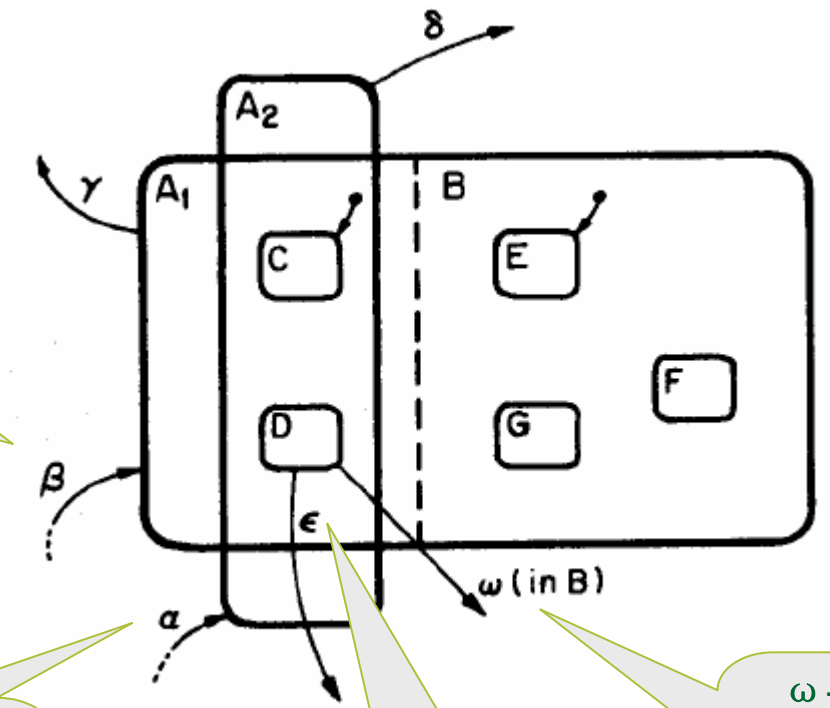
- A and D are now related by OR, not XOR
  - too much of this kind of overlapping will burden the specification

Example

state A, with substates, internal transitions, etc., that *'lives alone'* under some circumstances, but is joined in *'orthogonal marriage'* with a state B under others.



Example (2)



the β -entrance is an entrance to the product of A_1 and B

α -entrance is an entrance to A_2 alone

ε -exit leaves D regardless of B

ω -exit has the condition about being in B attached

Recommended Reading

- 陆汝钤，*计算机语言的形式语义*，1992
- Glynn Winskel, *The Formal Semantics of Programming Languages*, 1993
- David Harel, *Statecharts: A Visual Formalism for Complex Systems*, 1987
- W. Reisig, *Petri Nets: An Introduction*, 1985
- A. Pnueli, *The temporal logic of programs*, 1977
- R. Milner, A Calculus of Communication Systems, 1980

Thanks !

Appendix

- Timing aspect in Sequence Diagrams and Timing Diagrams
  - There are also timing description in Sequence Diagrams
  - timing description in Sequence Diagrams can be translated into Timing Diagrams