Brief Introduction to UML 2.0 *III*

– State Machine Modeling in UML2.0

(for SEG seminar)

Tian Zhang

Nanjing University,China

November 2005

Appendix I — 关于并发和并行

- 并行性(parallelism)有三种含义：
  - 同时性(simultaneity)：指两个或多个事件在同一时刻发生在多个资源上；
  - 并发性(concurrency)：指两个或多个事件在同一时间间隔发生在多个资源上；
  - 流水线(pipeline)：指两个或多个事件发生在可能重叠的时间段内。

Outline

- 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.
- 总之，UML2.0中的状态机是传统状态机的一个发展，它支持**OO**、层次化、并发性、通信等，以实现对现代复杂分布式系统的建模。

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

- Some related work recommending *state machines* for :
  - the user interface of interactive software;
  - the specification of data-processing systems;
  - hardware system description;
  - the specification of communication protocols;
  - computer aided instruction

*注：以上均为上世纪**70,80**年代针对传统状态机的工作*

Harel Statecharts

- David Harel
  - The William Sussman Professorial Chair
  - Dept. of Computer Science and Applied Mathematics
  - The Weizmann Institute of Science
  - cofounder of **i-Logix** Inc.
- Research interests
  - In the past, but diminished recent years
    - computability and complexity theory, logics of programs, database theory, *automata* theory
  - recent years
    - systems engineering, OO analysis and design, visual languages, layout of diagrams

Warm up

- How to model the following in conventional state machine diagrams :
  1. In all airborne states, when yellow handle is pulled seat will be ejected.
  2. Gearbox change of state is independent of braking system.
  3. When selection button is pressed enter selected mode.
  4. Display-mode consists of time-display, date-display and stopwatch-display.

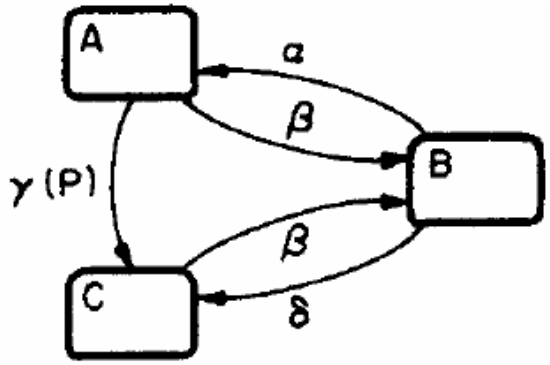
Warm up (2)

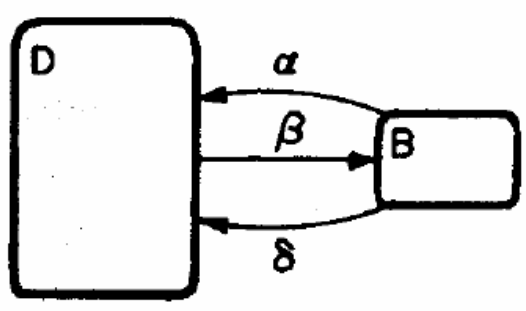
- In all airborne states, when yellow handle is pulled seat will be ejected
  - calls for the ability to *cluster* states into a superstate
- Gearbox change of state is independent of braking system
  - introduces independence, or *orthogonality*
- When selection button is pressed enter selected mode
  - hints at the need for more general transitions than the single event-labelled arrow
- Display-mode consists of time-display, date-display and stopwatch-display
  - captures the refinement of states

Three Elements

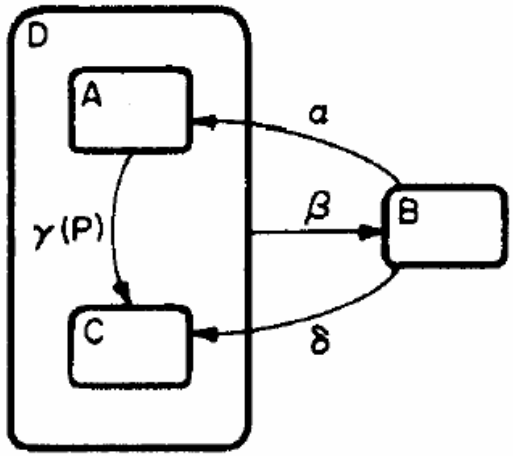
- Statecharts extend conventional state-transition diagrams with essentially three elements :
  - Hierarchy
  - Concurrency
  - Communication
- In a nutshell, one can say :
  - statecharts = state-diagrams + *depth*
    orthogonality + *broadcast-communication*

State-levels: Clustering

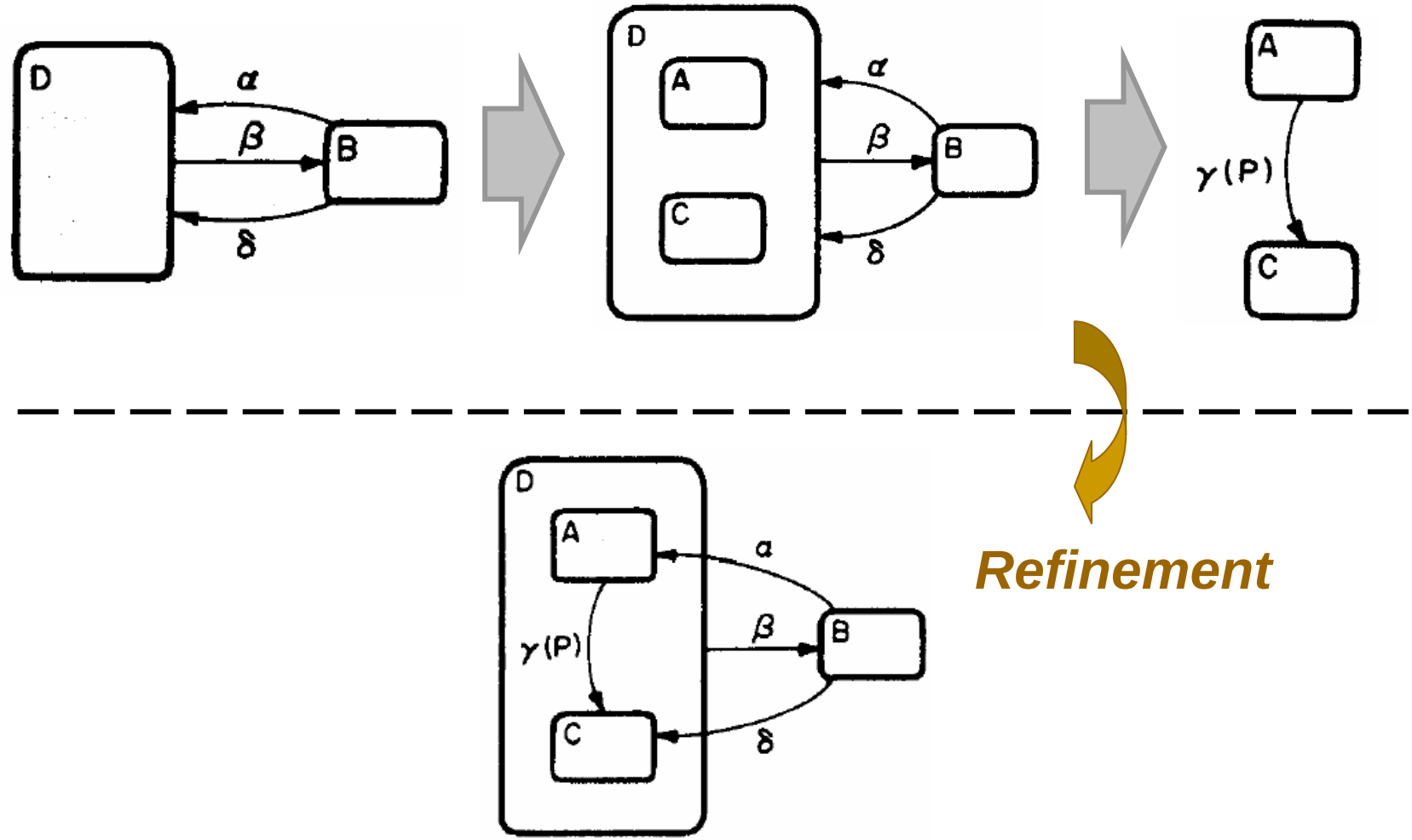




Clustering

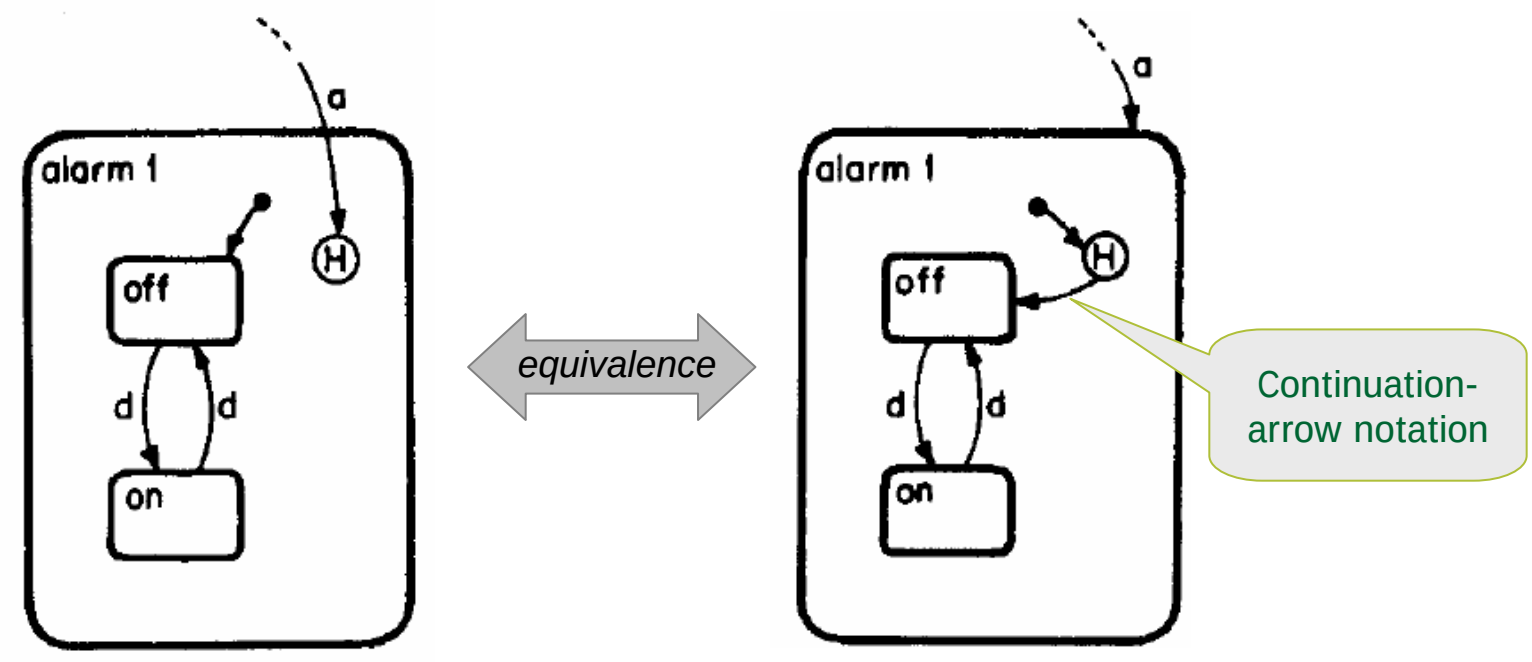


State-levels: Refinement



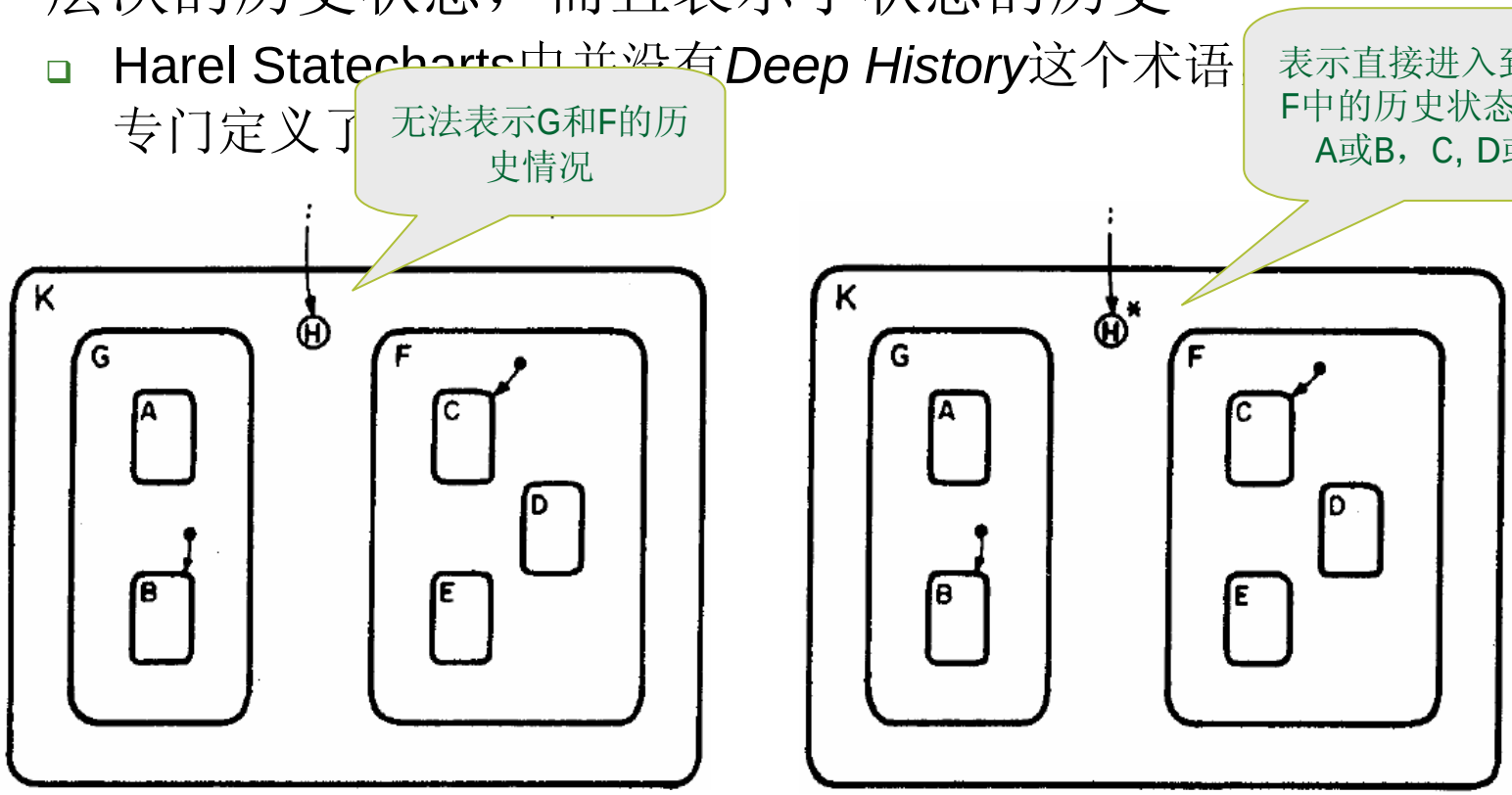
History states

- History states (H) is one of the most interesting and frequent ways of entering a group of states.
  - the simplest ‘enter-by-history’ is entering the state most recently visited



Deep history

- 在Harel Statecharts中有一种历史状态(H*)不但表示所在层次的历史状态，而且表示子状态的历史
  - Harel Statecharts中并没有*Deep History*这个术语，专门定义了



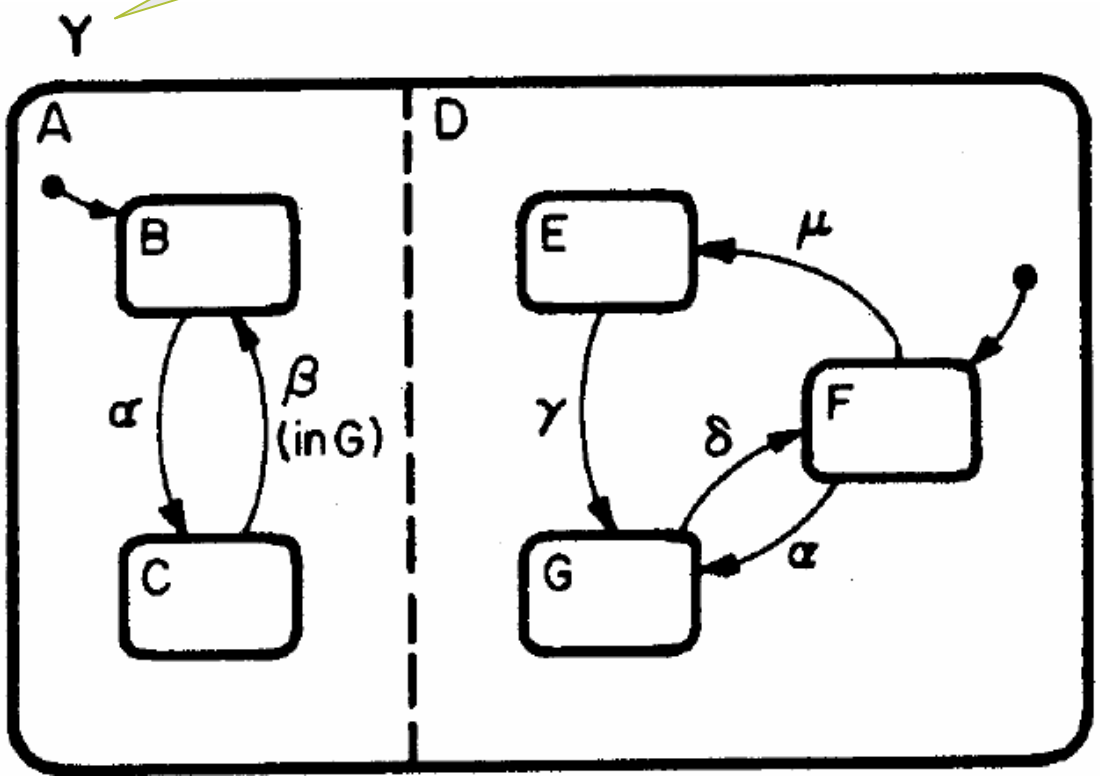
Orthogonality

- XOR (exclusive or) decomposition represents *Refinement* or *Clustering*
  - XOR means one and only one state in the superstate is the current state
- AND decomposition represents *Concurrency*
  - AND means all of AND components are current state
- Note that AND must be specialized but not XOR

Example

Y is the orthogonal product of A and D

Being in Y entails being in some combination of B or C with E, F, or G that is *A X D*

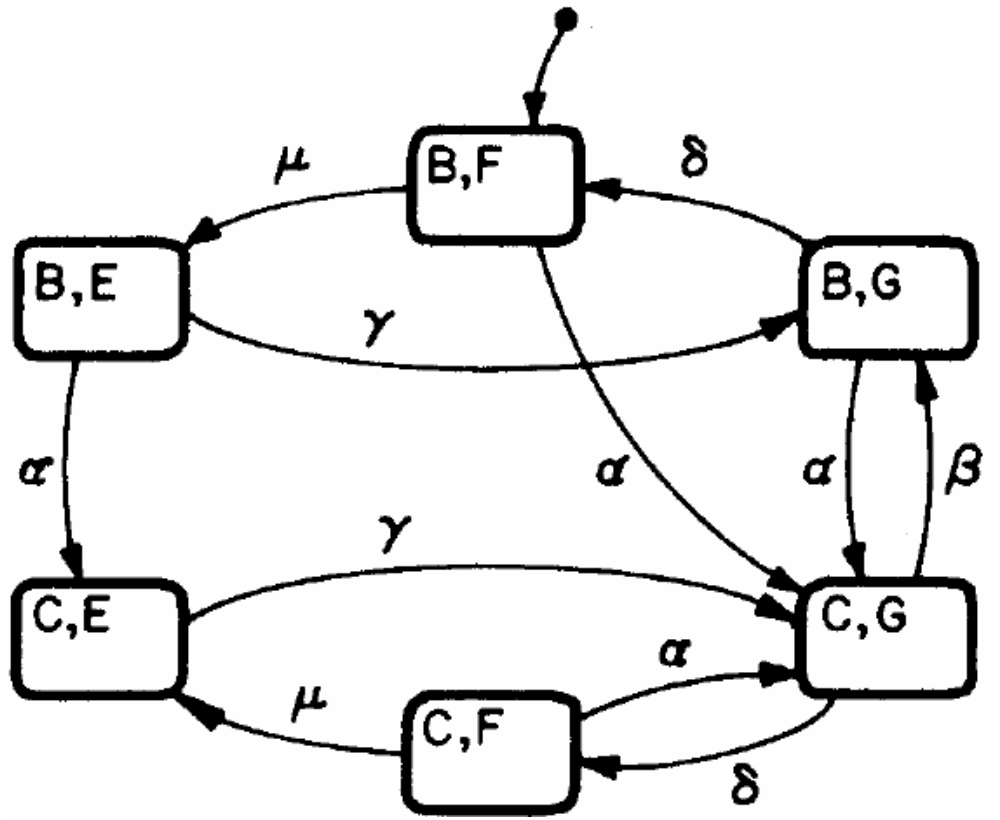


default

Entering Y by defaulting is actually entering the combination (B, F) by the default arrows

Example (2)

- Conventional AND-free equivalent



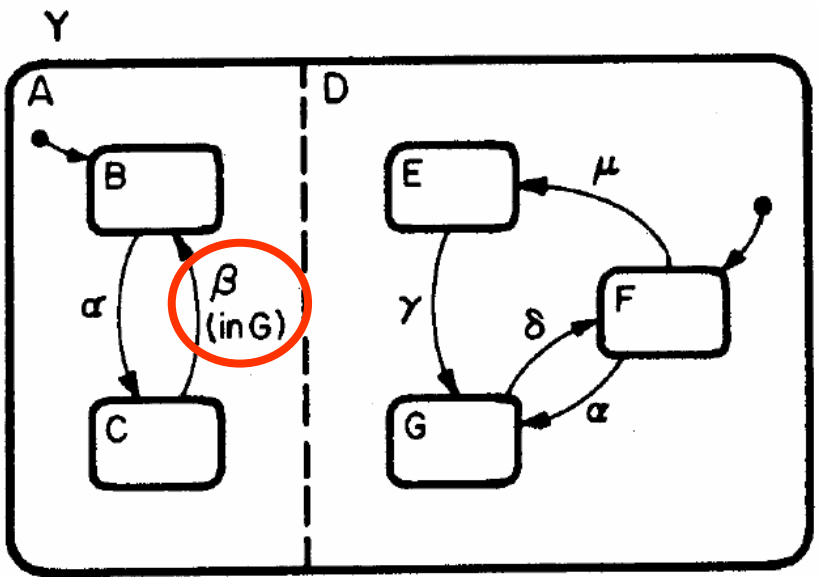
This is root of the exponential blow-up in the number of states

AND is Syntax sugar?

Dependency in Independency

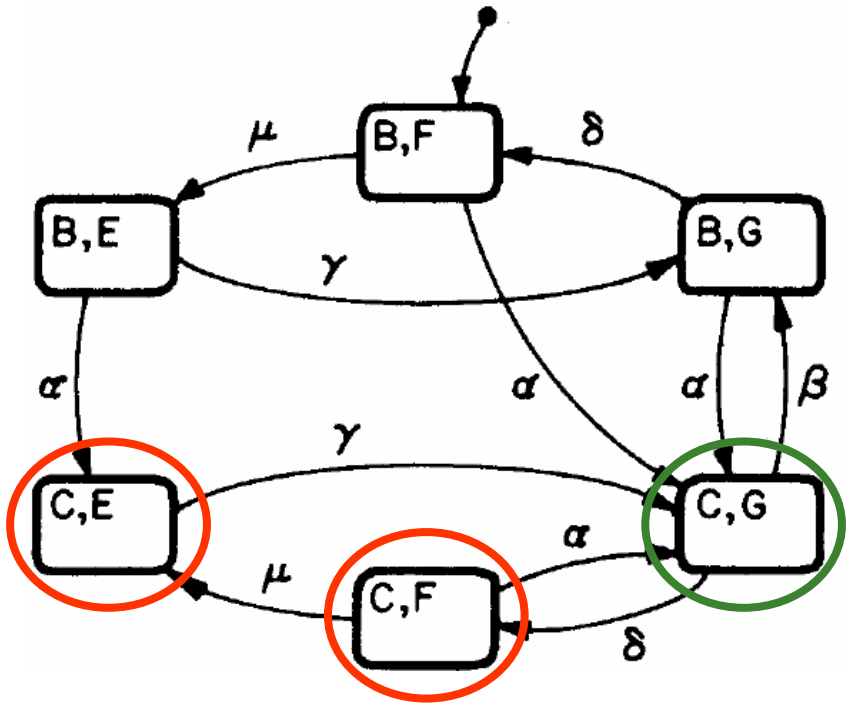
- Formally, orthogonal product is a generalization of the usual product of automata
  - the difference is that the latter is usually required to be a *disjoint* product, whereas here some *dependence* between components can be introduced
  - show dependence by *common events* or *conditions*

"in G"- like Condition



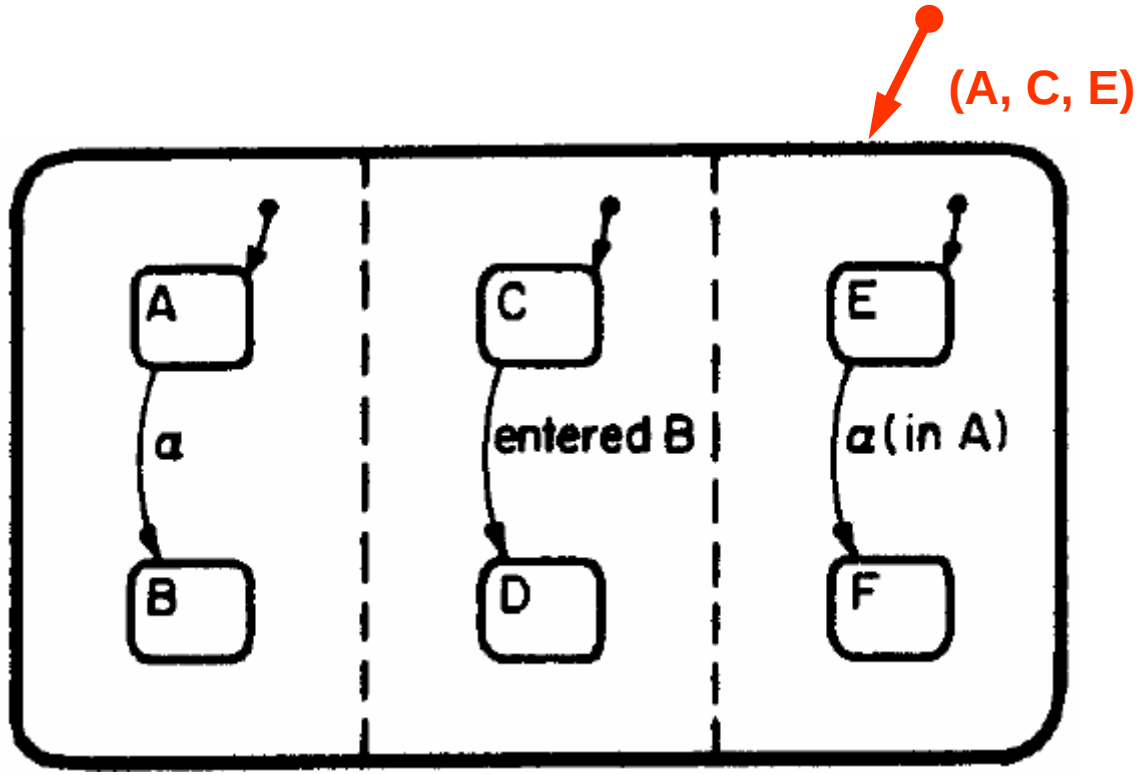
orthogonal product

usual product of automata



Broadcast-Communication

- Broadcast-communication实际上是orthogonality所需要的一种产物

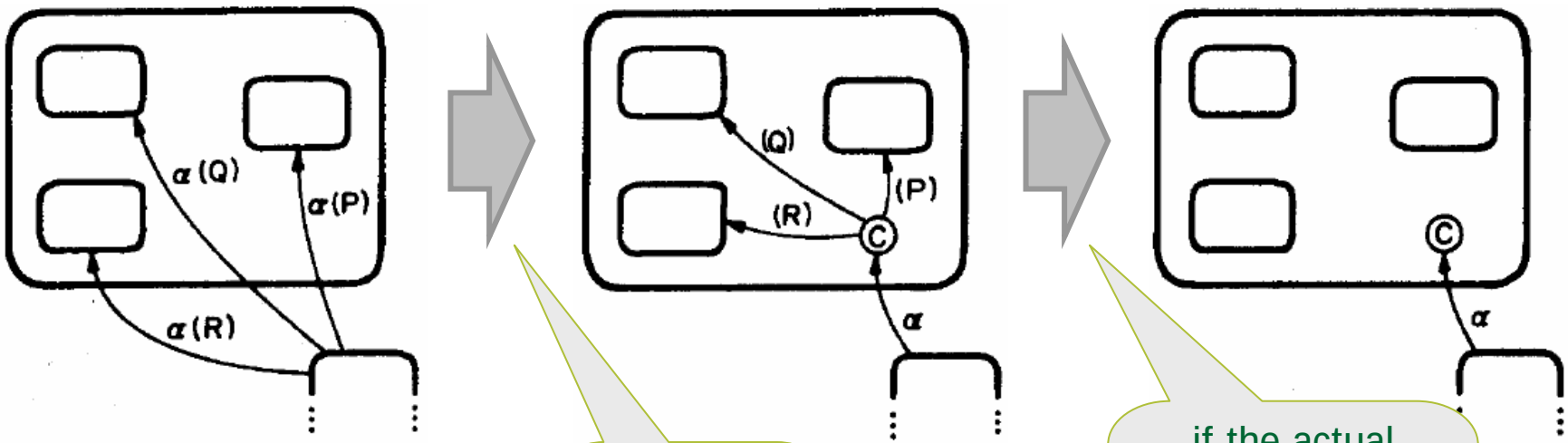


Additional statechart features

- Condition and selection entrances
- Delays and timeouts
- Unclustering
- Actions and activities

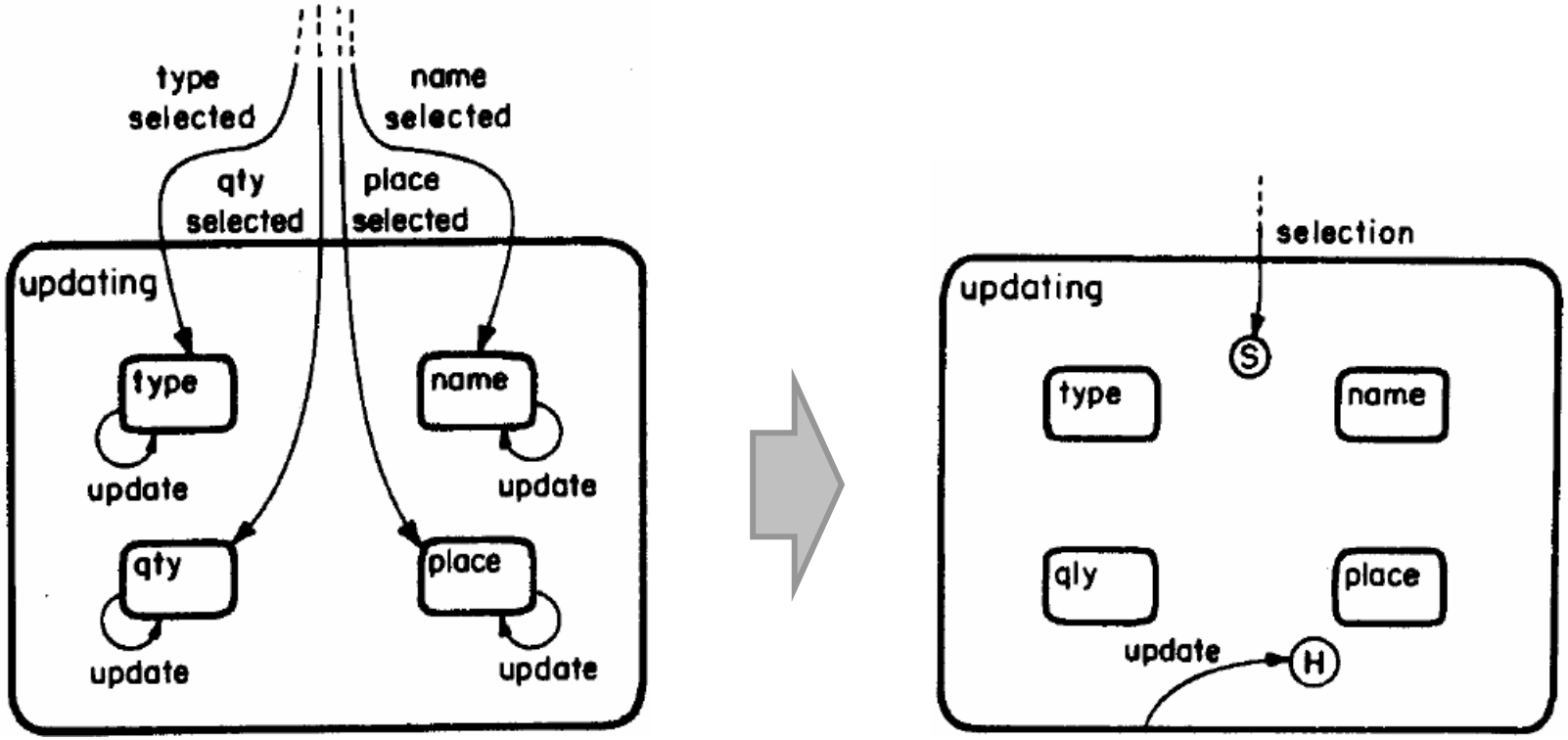
Condition entrances

- For abbreviation more complicated entrances to substates of superstate than a simple direct arrow.



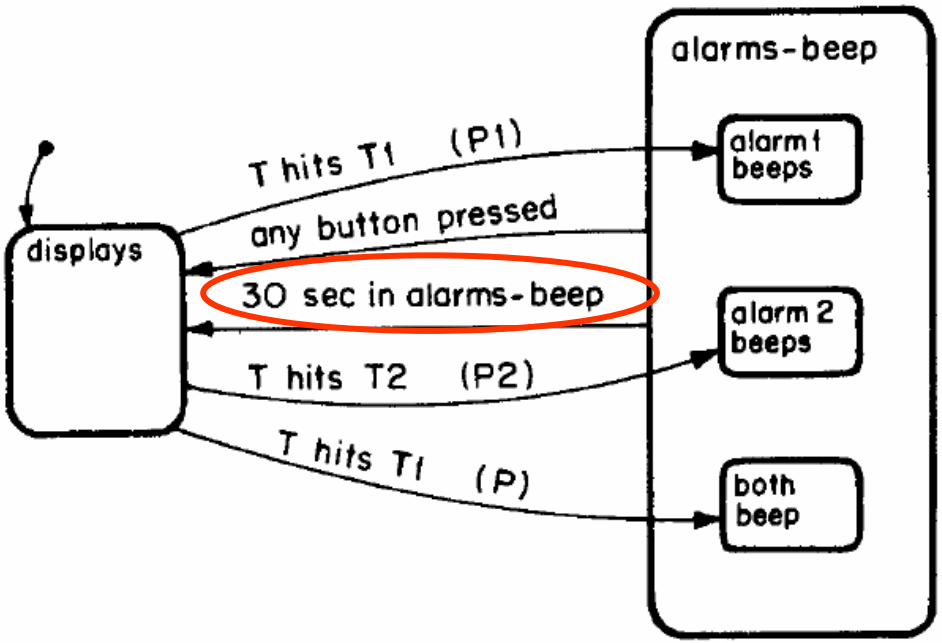
Selection entrances

- Selection occurs when the state to be entered is determined in a simple one-one fashion by the ‘value’ of a generic event.

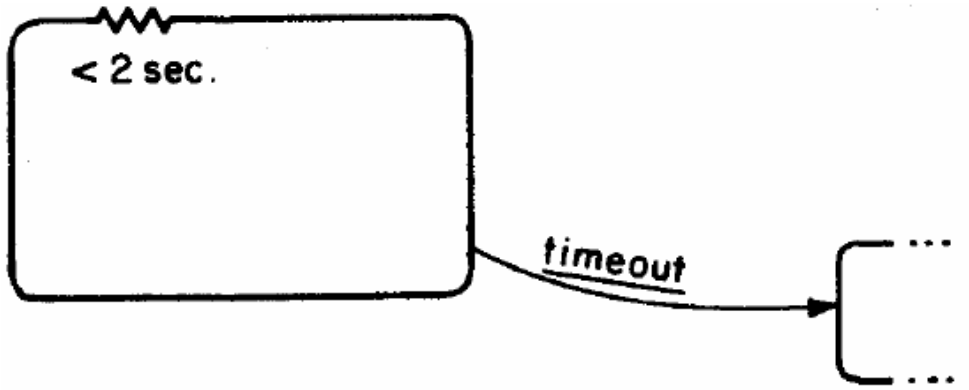


Delays

- The present formalism treats time restrictions using implicit timers.



Timeouts



timeout (*entered state*, *bound*)

- In general, the *syntax* of the specification attached to a squiggle is $\Delta t_1 < \Delta t_2$
  - either one of the Δt_i can be omitted
  - lower bound means that events do not apply in the state until the lower bound is reached

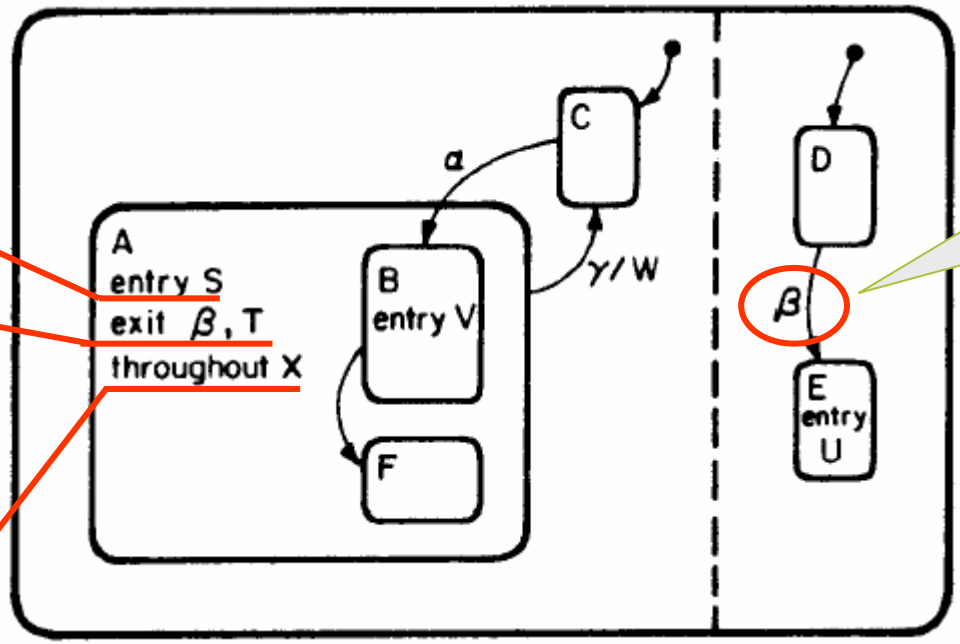
Actions and activities

- Review, so far, the ‘pure’ statecharts
  - what ‘pure’ statecharts represent is the control part of the system
  - the reactivity is expressed only by the system changing its internal states in response to events and conditions
- New challenge is
  - How to *generate events* and to *change the value of conditions* in statecharts ?

Actions and activities (2)

- 动作
  - 动作是瞬时的，可以被看作是一个输出
  - 在某一个跃迁下发生的动作可以被作为进入另一个状态的跃迁事件
  - 动作可以标志在跃迁上，也可以标志在状态上
- 活动
  - 活动是有持续时间的
  - 活动的开始、结束等均由动作来控制
  - 活动主要是对应于特定的物理部件，如显示屏、响铃、计时器等

Actions and activities (3)



进入状态时执行动作S

离开状态时将作为β作为输出事件，并执行动作T

在整个状态A期间都要执行活动X

左半部的输出事件β是右半部的输入事件，并触发跃迁

Object Variant of Harel Statecharts

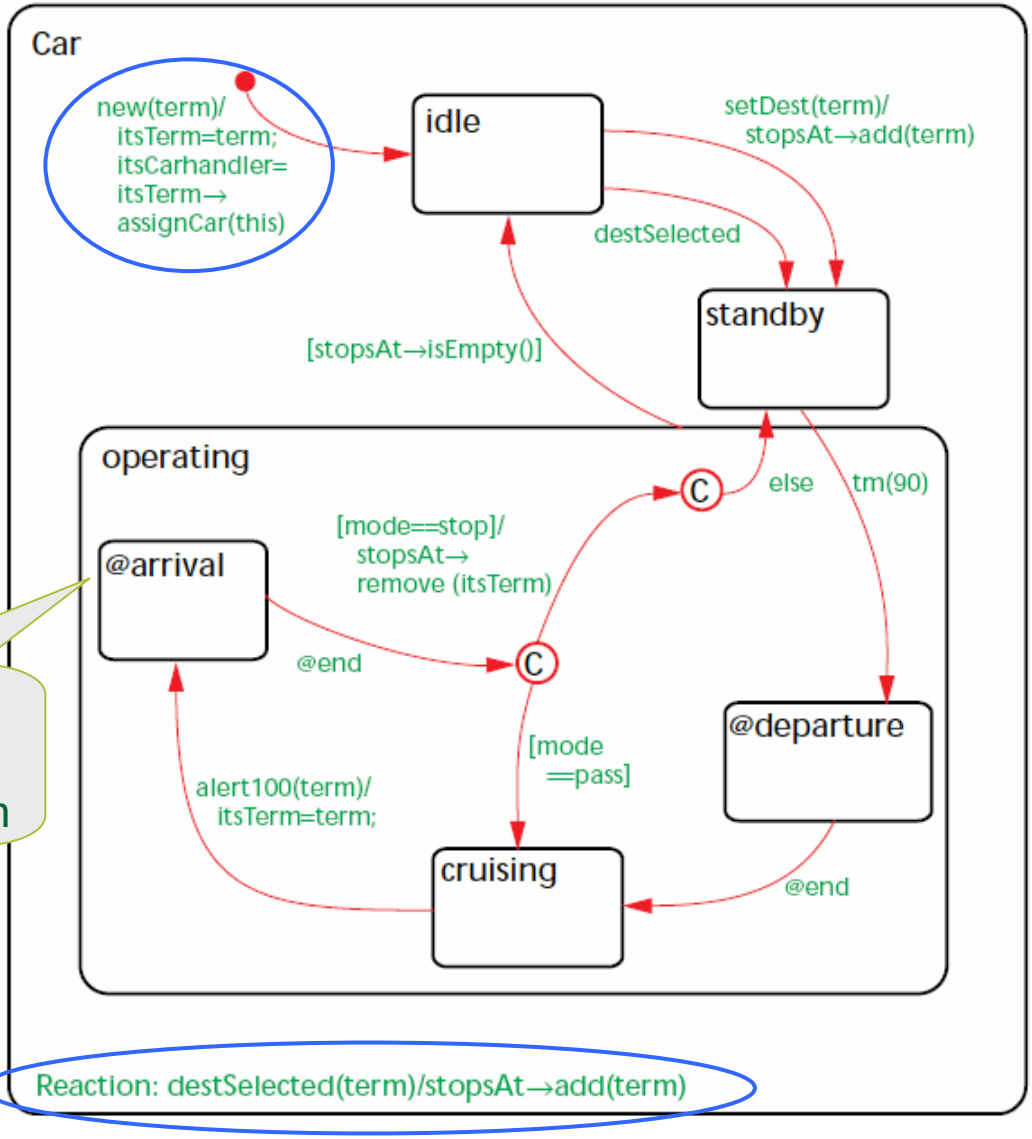
- One of the main technical issues is what mechanism to use for inter-object interaction
- Harel Statecharts adopt two mechanisms:
  - An object can *generate an event* to the target server object;
  - An object can also directly *invoke an operation* of another object.
- Next question is how to present them in statecharts?

Triggers and actions

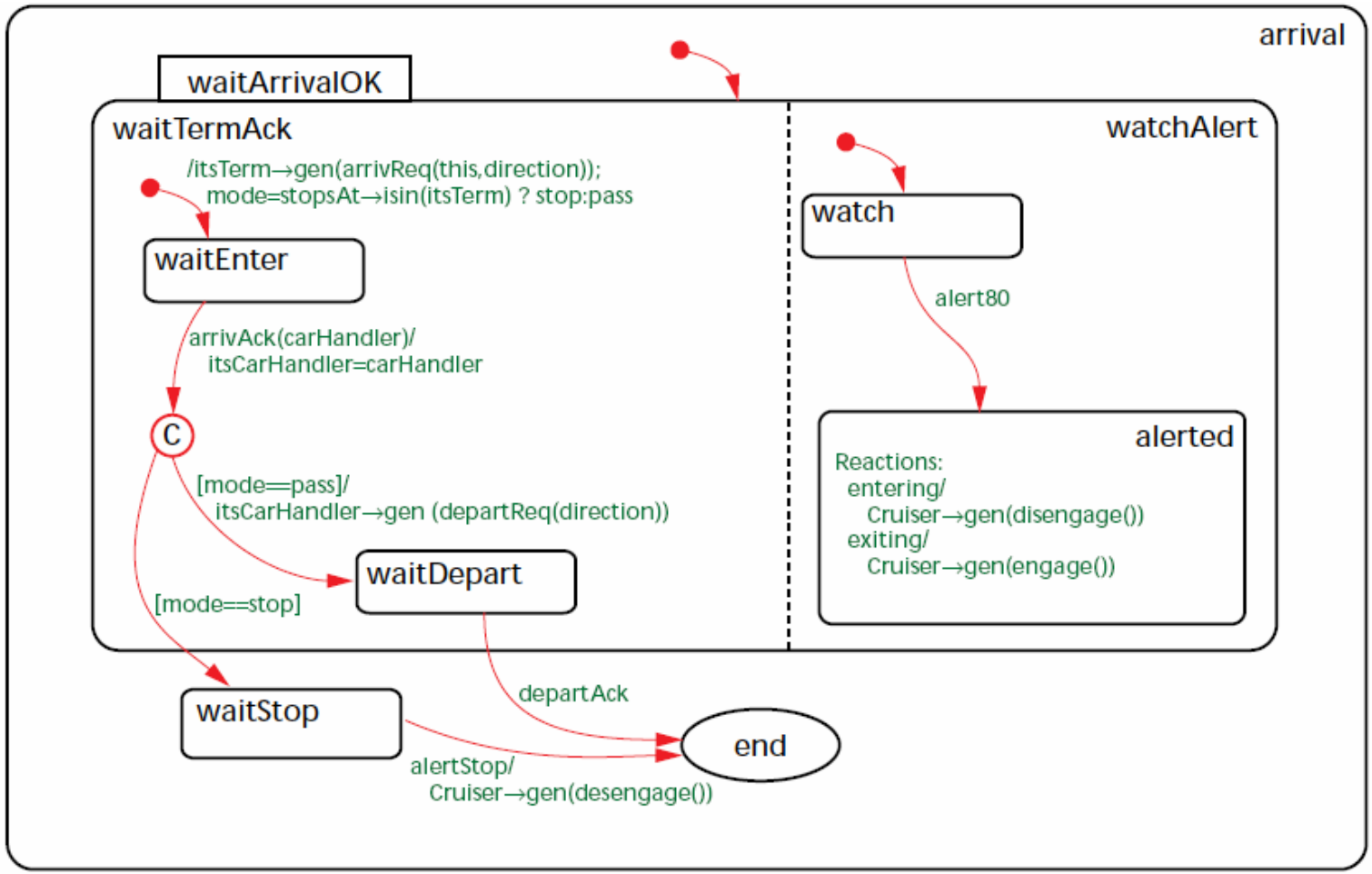
- Statecharts involve reactions of the form

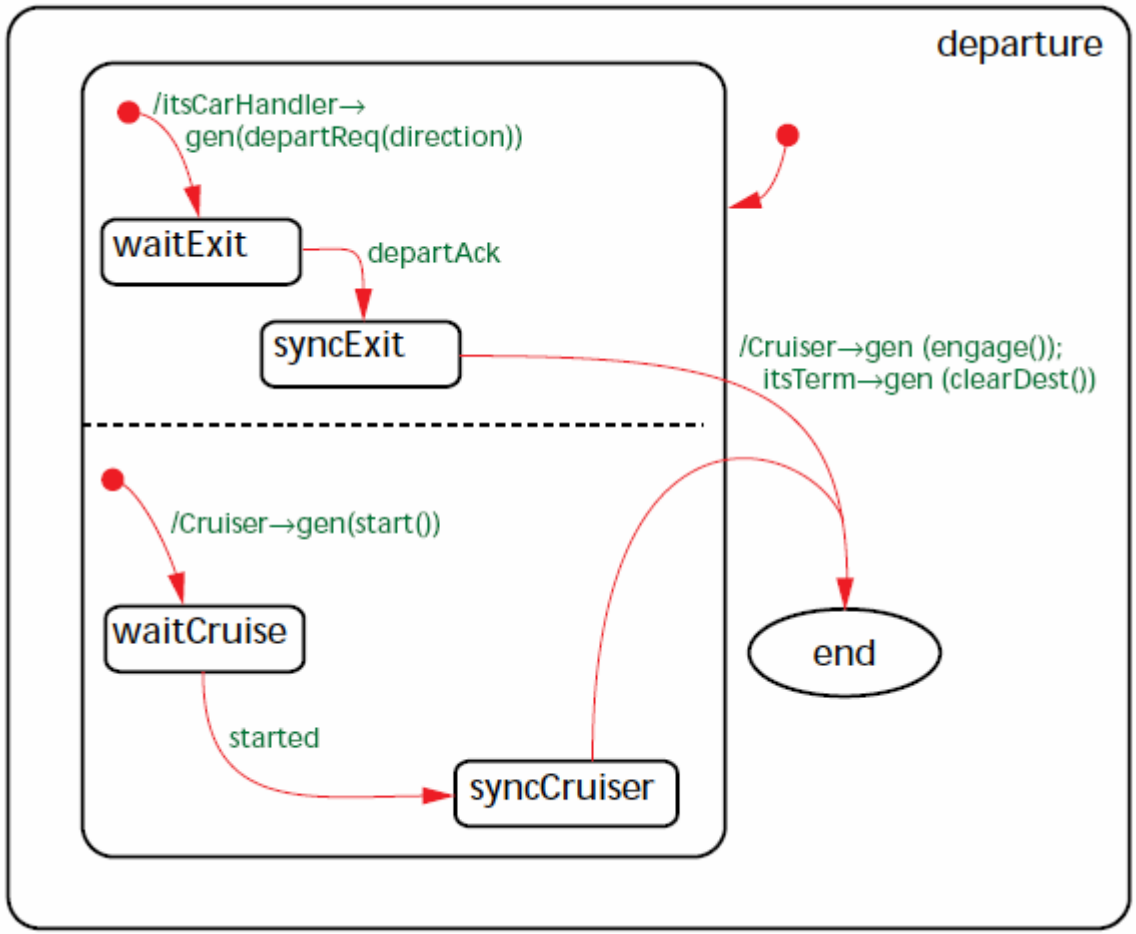
  trigger [condition] / action-list

- Note that
  - all parts of which are optional
  - such a reaction can adorn a transition arrow or appear within a state's reaction spec
  - a trigger is either *an event expression* or *an operation request*
  - actions are sequences of *event-generation expressions*, *operation invocations*, and *C++ statements*



"@" means superstate, and for decomposition


arrival
waitArrivalOK
waitTermAck
/itsTerm→gen(arrivReq(this,direction));
mode=stopsAt→isin(itsTerm) ? stop:pass
waitEnter
arrivAck(carHandler)/
itsCarHandler=carHandler
C
[mode==pass]/
itsCarHandler→gen (departReq(direction))
waitDepart
[mode==stop]
waitStop
departAck
end
alertStop/
Cruiser→gen(desengage())
watchAlert
watch
alert80
alerted
Reactions:
entering/
Cruiser→gen(disengage())
exiting/
Cruiser→gen(engage())

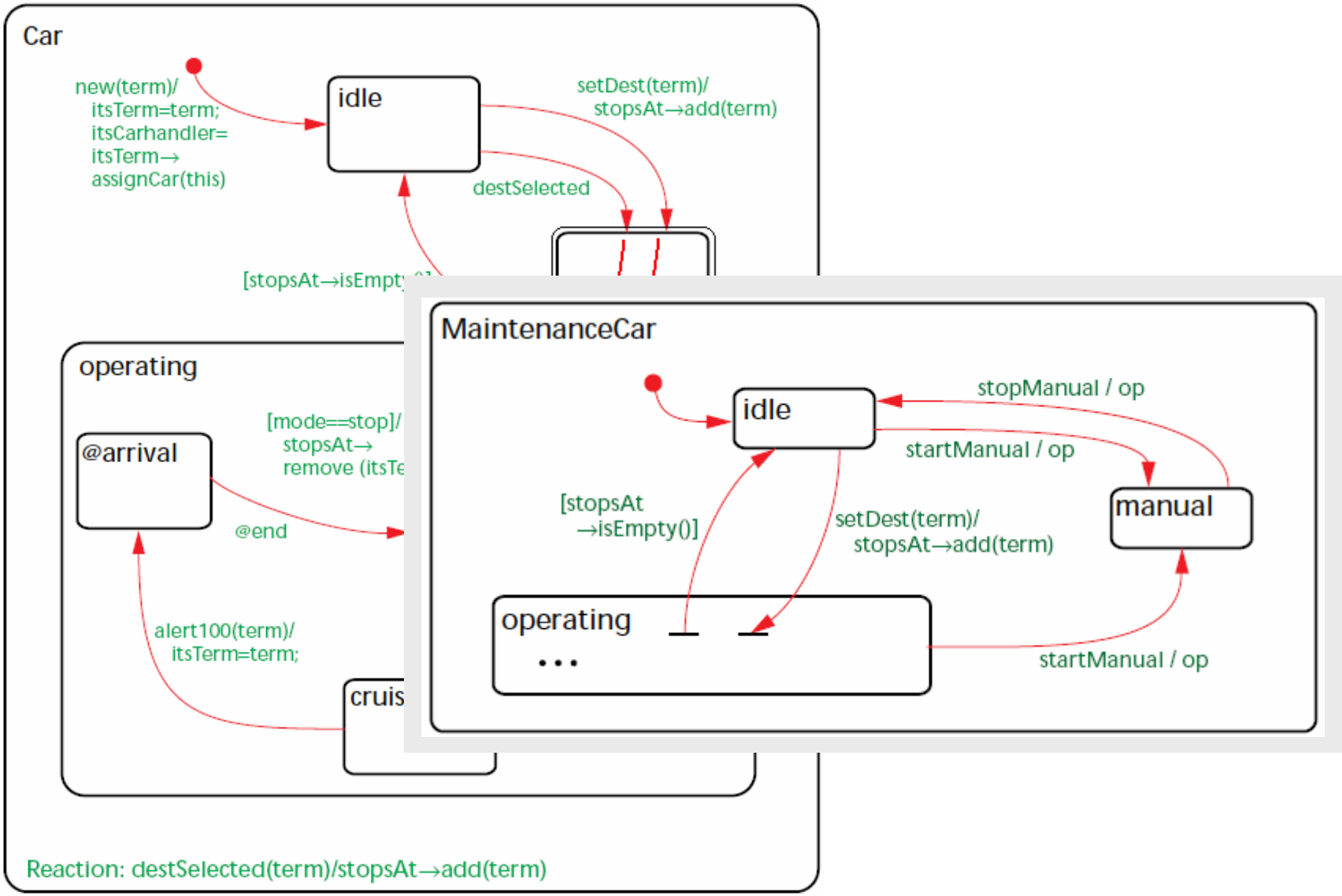

departure
/itsCarHandler→
gen(departReq(direction))
waitExit
departAck
syncExit
/Cruiser→gen (engage());
itsTerm→gen (clearDest())
/Cruiser→gen(start())
waitCruise
started
syncCruiser
end

Inheritance: Structural or Behavioral Conformity?

- 引入Inheritance的一个重要的意图是更大程度的实现重用，以降低开发成本，简化开发过程。
- 从外在表现来看
  - 父类、子类之间是*is-a*的关系
  - 父类出现的地方可由子类替代
  - 子类从内部结构和外部行为接口上都具备了父类的特点
- 从结构上来看，子类确实具备了父类的特点。但从完整的行为上来看，子类和父类之间有相当大的差别
  - 这种行为上的差异性导致了状态上的巨大差异
  - 如何在*statecharts*中实现子类对象重用父类对象的*statecharts*?

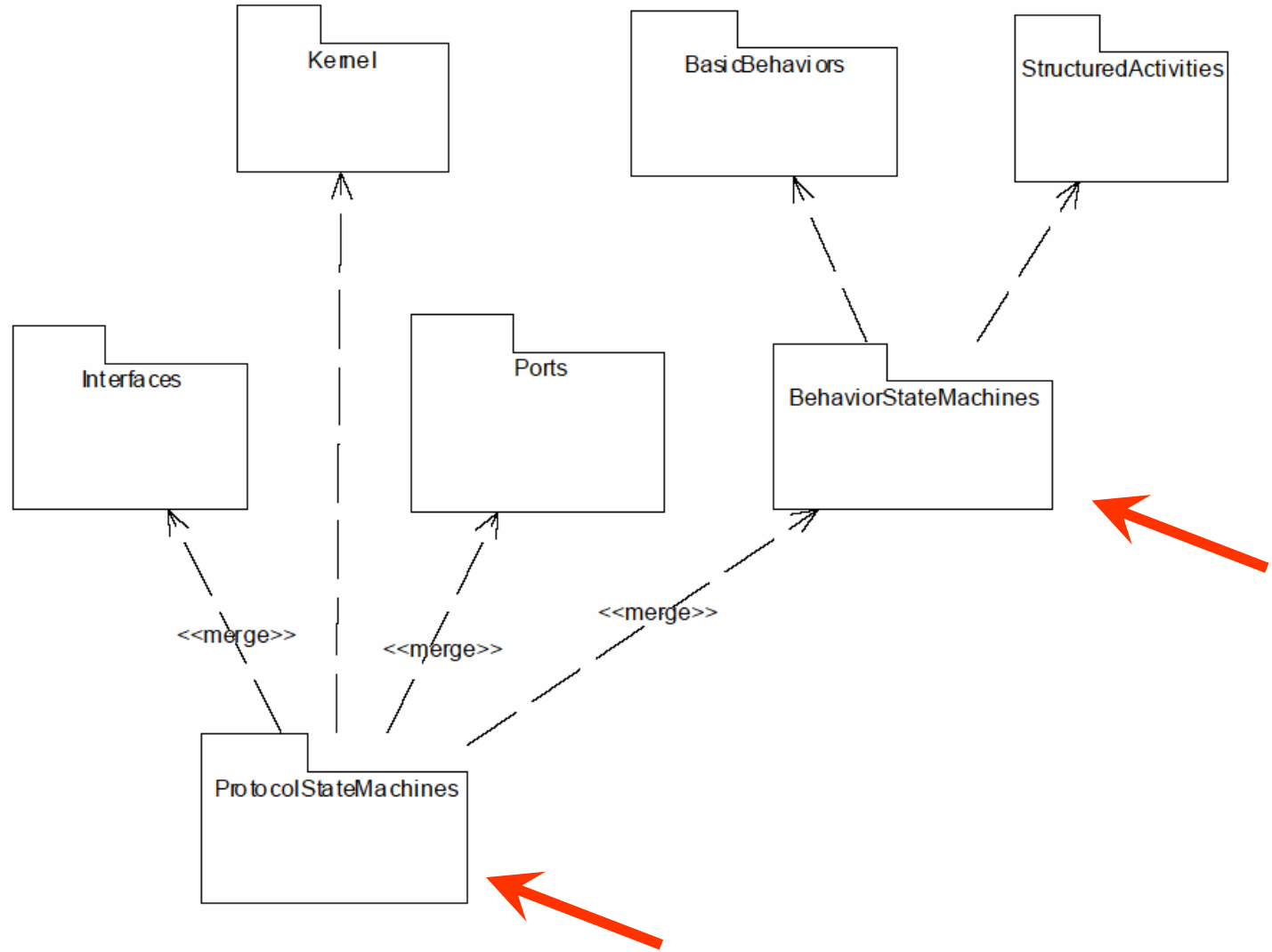
Inheritance in Harel Object-Statecharts

- The main guideline in addressing inheritance
  - *careful modifiability* but not full behavioral conformity
  - to base the two statecharts on the same underlying *state/ transition topology*
- Inherited states and transitions cannot be removed, but certain refinements are allowed
  - Decompose a basic (atomic) state by OR or by AND
  - Add substates to an OR state
  - Add orthogonal components to any state
  - Transitions can be added
    - the target state can be changed, but not the source state
    - the source can be changed to a lower level state


Car
new(term)/
itsTerm=term;
itsCarhandler=
itsTerm→
assignCar(this)
idle
setDest(term)/
stopsAt→add(term)
destSelected
[stopsAt→isEmpty
operating
@arrival
[mode==stop]/
stopsAt→
remove (itsTe
@end
alert100(term)/
itsTerm=term;
cruis
Reaction: destSelected(term)/stopsAt→add(term)
MaintenanceCar
idle
stopManual / op
startManual / op
[stopsAt
→isEmpty()]
setDest(term)/
stopsAt→add(term)
manual
operating
• • •
startManual / op

State Machine Diagram

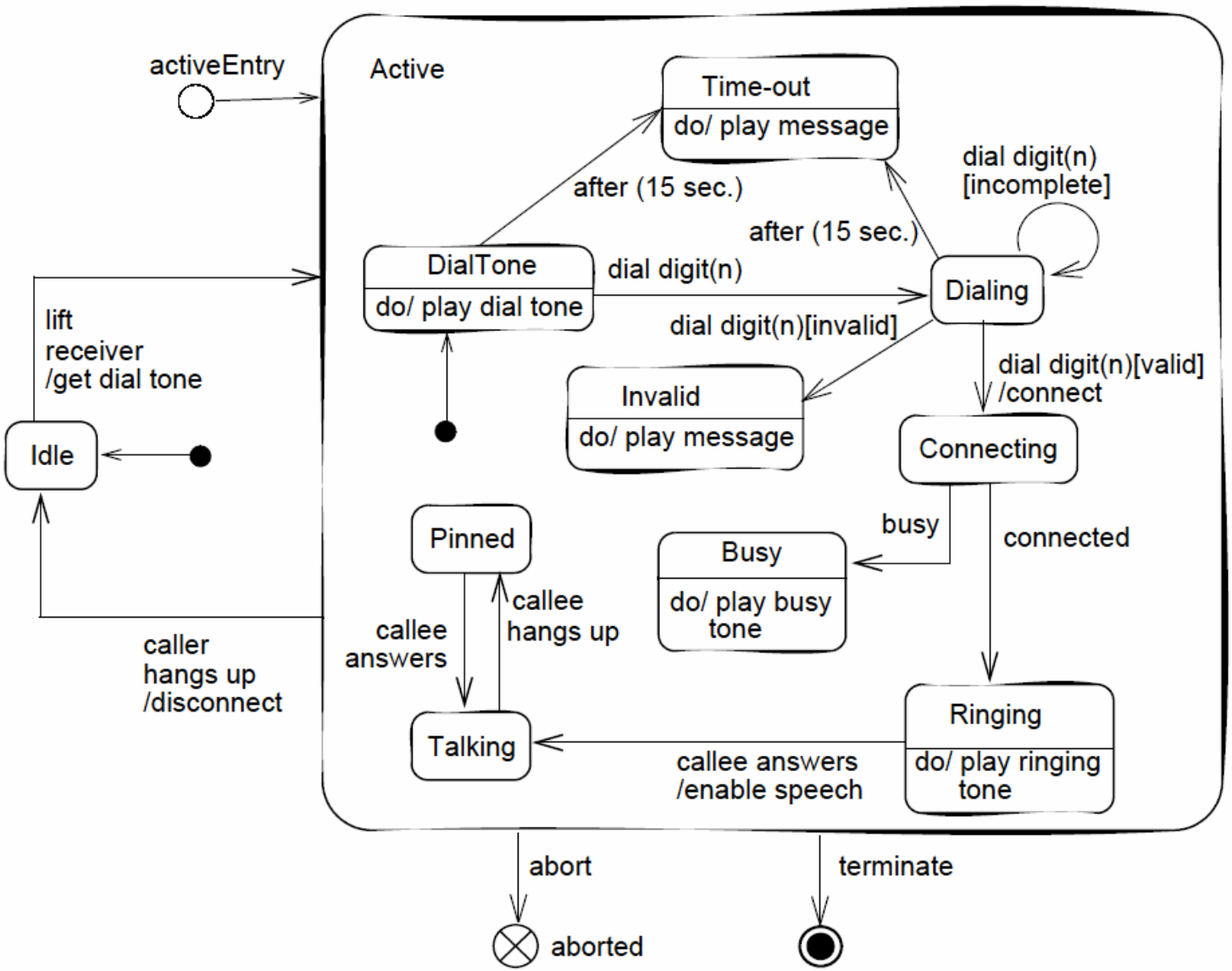
- Used for modeling discrete behavior through finite state transition systems
  - In addition to expressing the *behavior* of a part of the system, state machines can also be used to express the *usage protocol* of part of a system.
- Behavioral State Machines
  - State machines can be used to specify 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

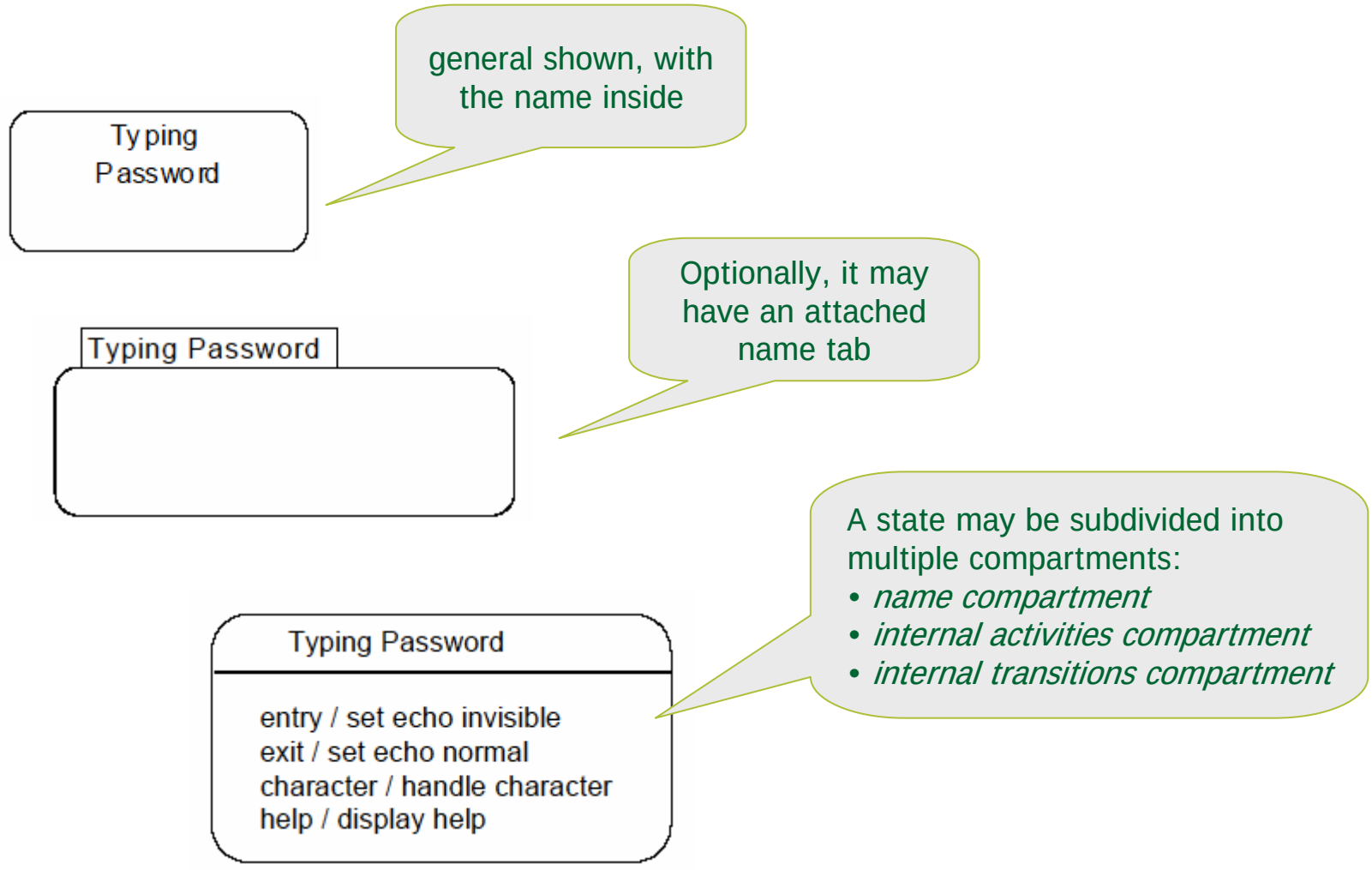
State

- 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*

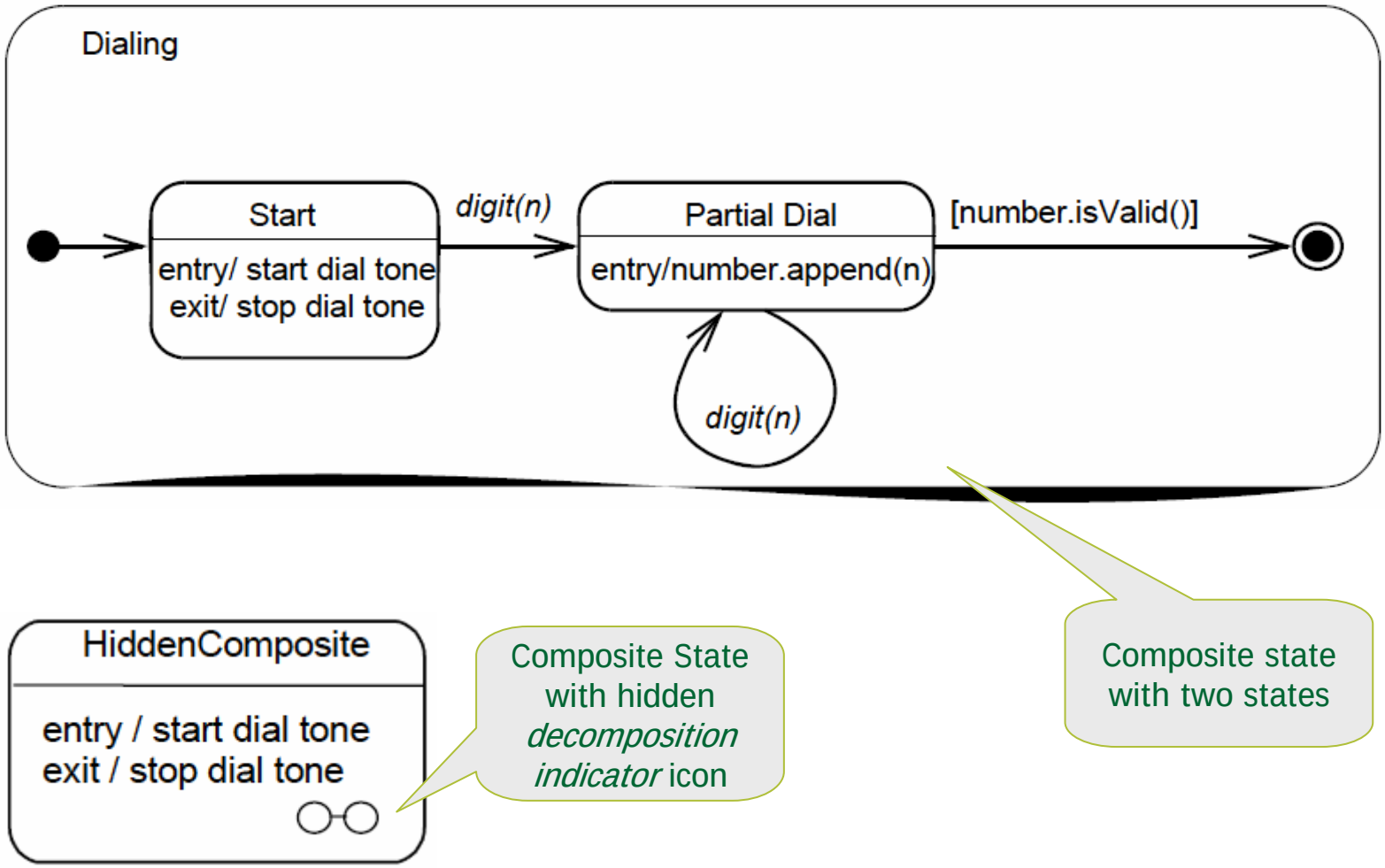
Notation

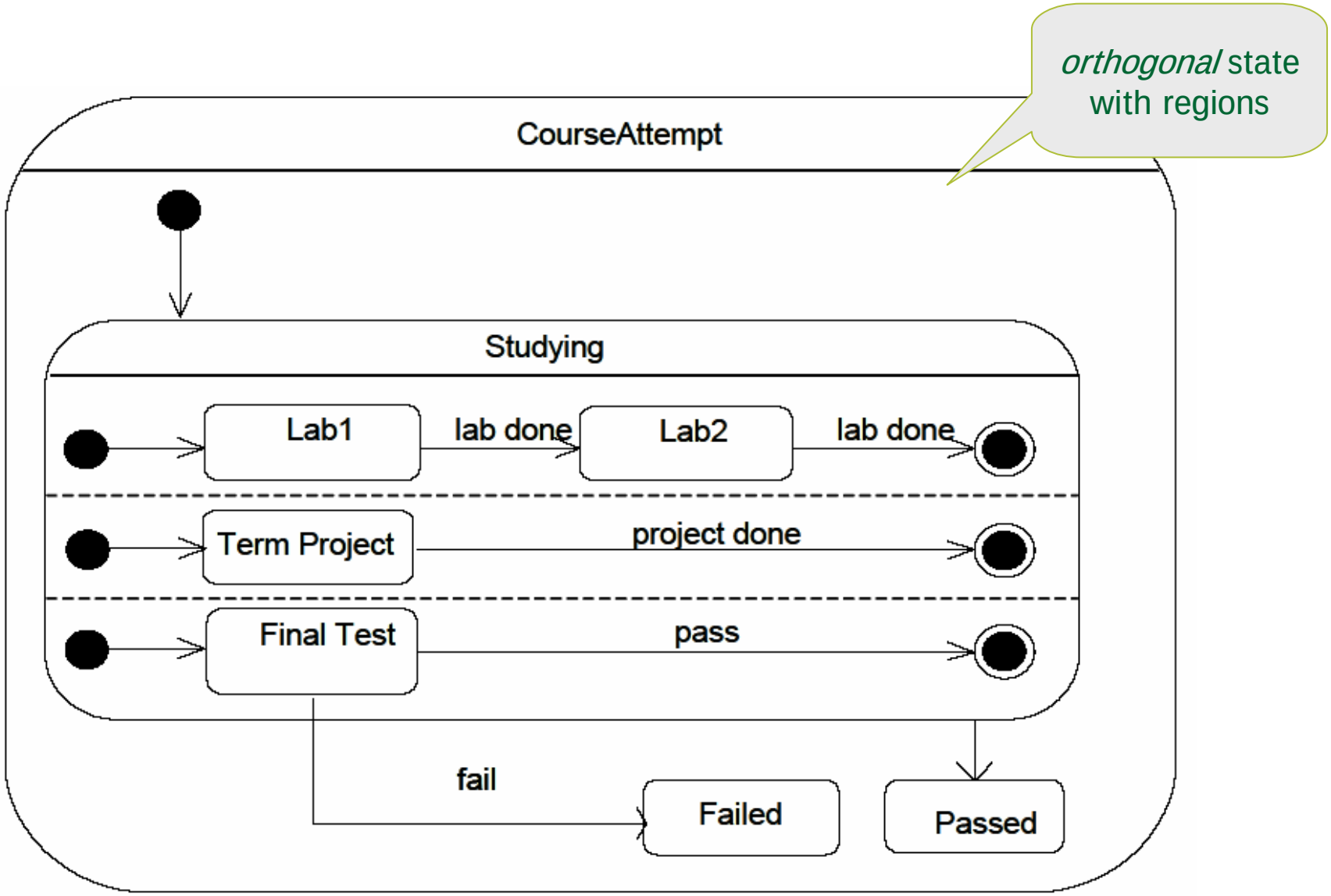

general shown, with the name inside
Typing Password
Optionally, it may have an attached name tab
Typing Password
A state may be subdivided into multiple compartments:
• name compartment
• internal activities compartment
• internal transitions compartment
Typing Password
entry / set echo invisible
exit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

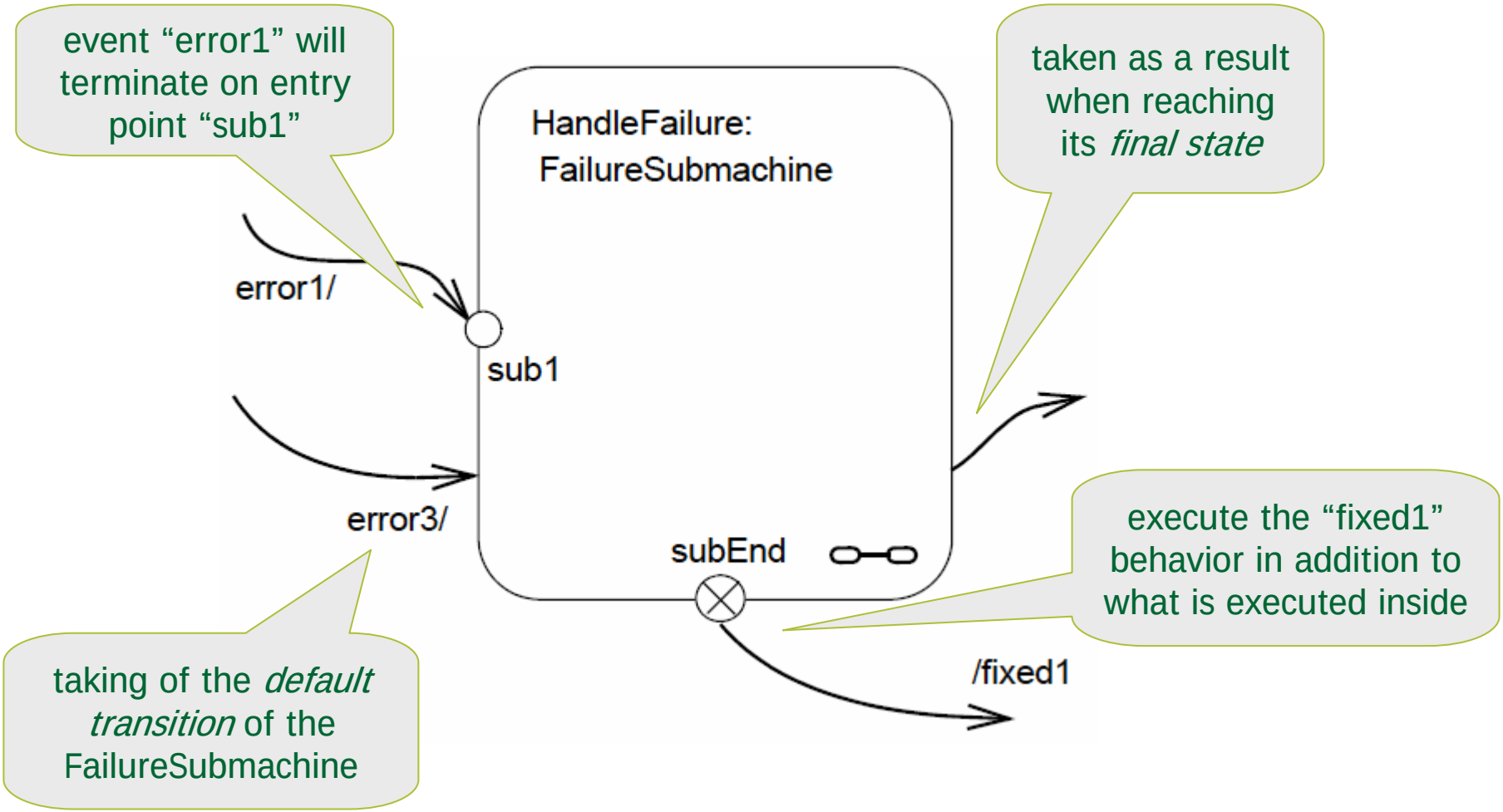

Dialing
Start
entry/ start dial tone
exit/ stop dial tone
digit(n)
Partial Dial
entry/number.append(n)
[number.isValid()]
digit(n)
HiddenComposite
entry / start dial tone
exit / stop dial tone
Composite State with hidden *decomposition indicator* icon
Composite state with two states


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

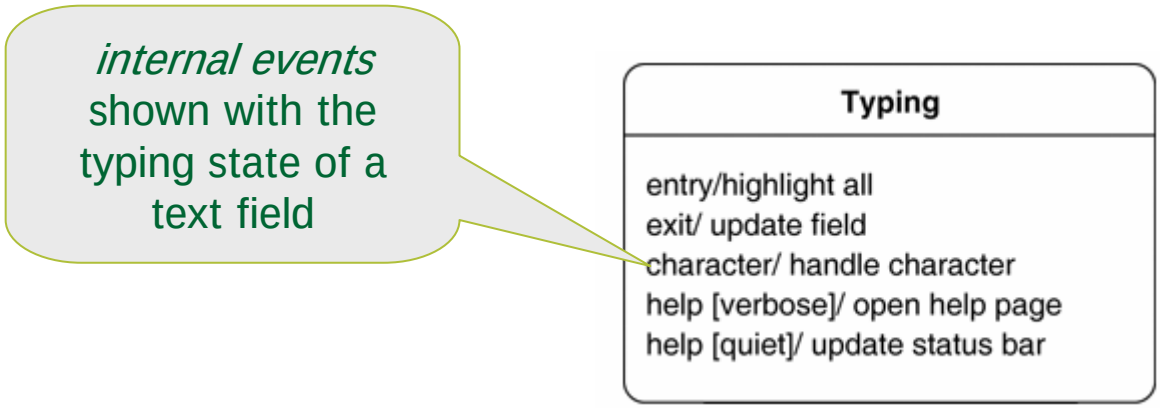
Submachine state

- A submachine state specifies the insertion of the specification of a submachine state machine.
- A submachine state is *semantically equivalent* to the composite state defined by the referenced state machine.
- *What difference?*
  - Entering and leaving submachine 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

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

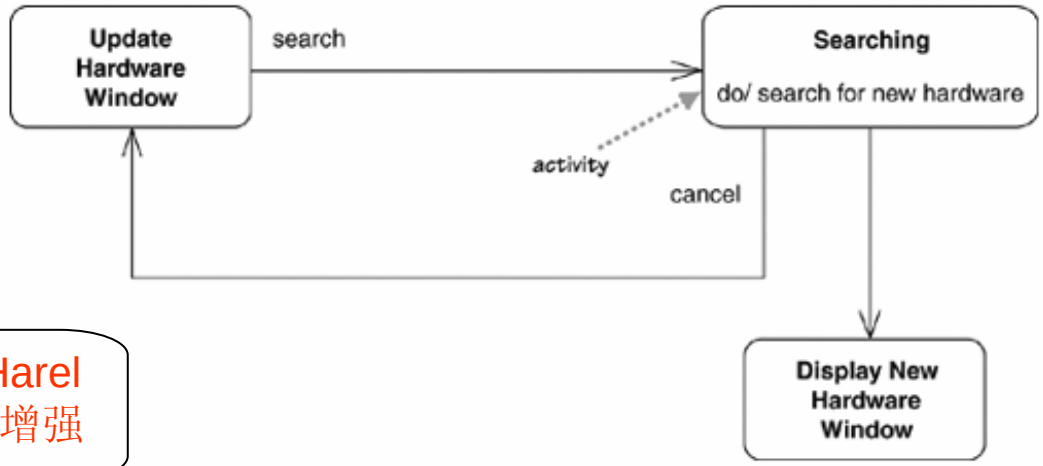


- States can react to events without transition, using internal activities: putting the event, guard, and activity inside the state box itself
- An internal activity is similar to a *self-transition*
  - They are both transitions that loop back to the same state
  - However, internal activities do not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



这一条是对Harel Statcharts的增强

- 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.

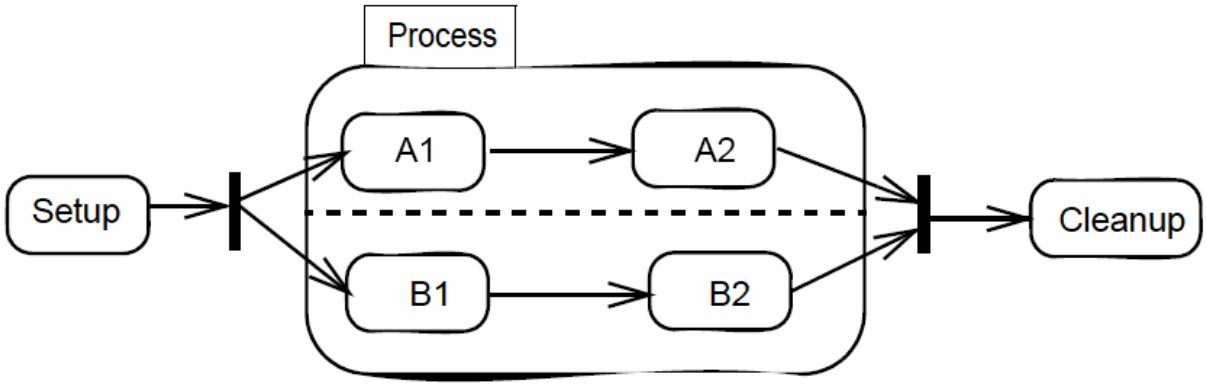
Notation

Initial Pseudo State

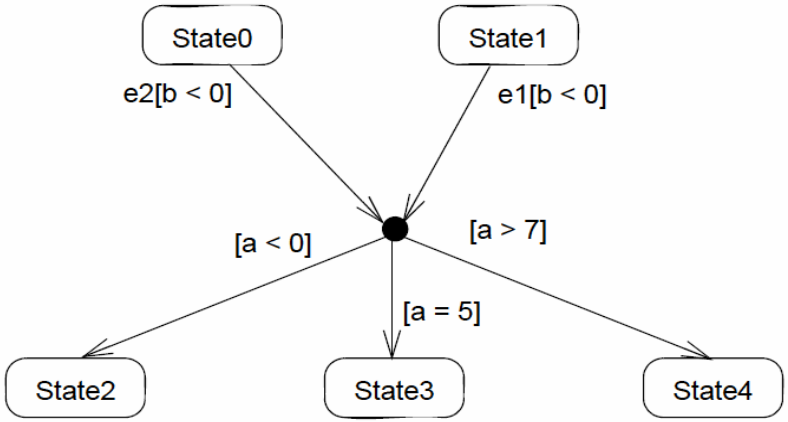
Shallow History

Deep History

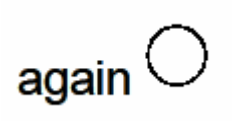
Terminate node



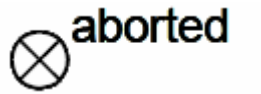
Fork and Join



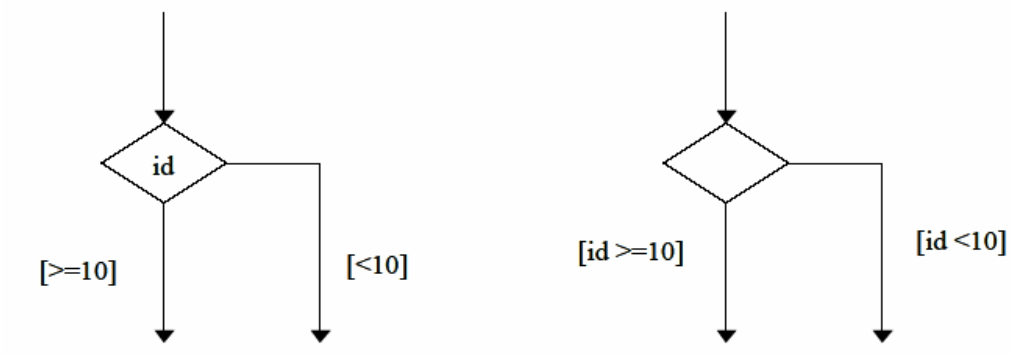
Junction Pseudo State



Entry point



Exit point



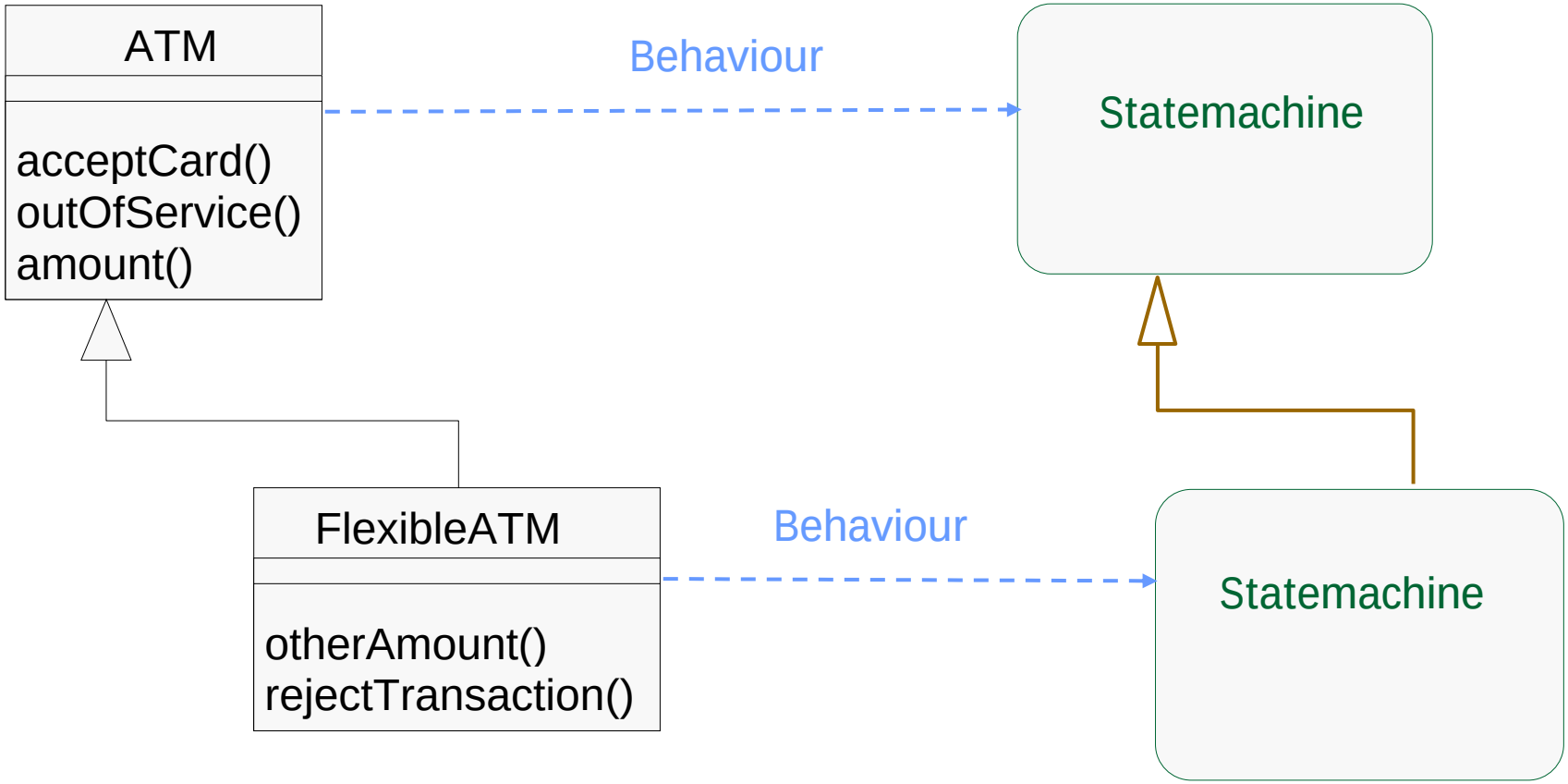
Choice Pseudo State

State redefinition

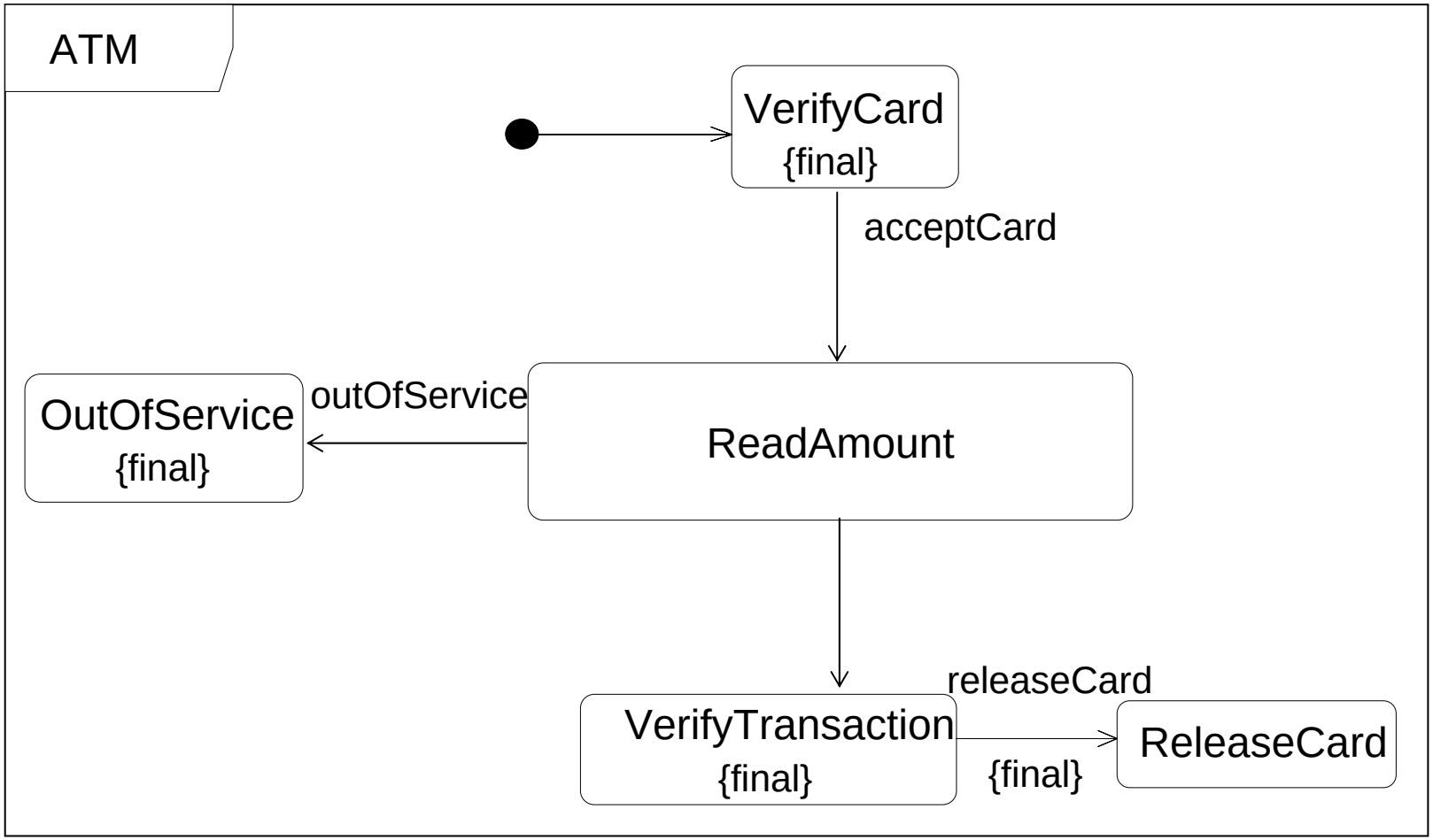
- A simple state can be redefined (extended) to become a composite state (by adding a region)
- a composite state can be redefined (extended) by adding regions and by adding vertices, states, entry/exit/do activities and transitions to *inherited regions*.
- The redefinition of a state applies to the 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

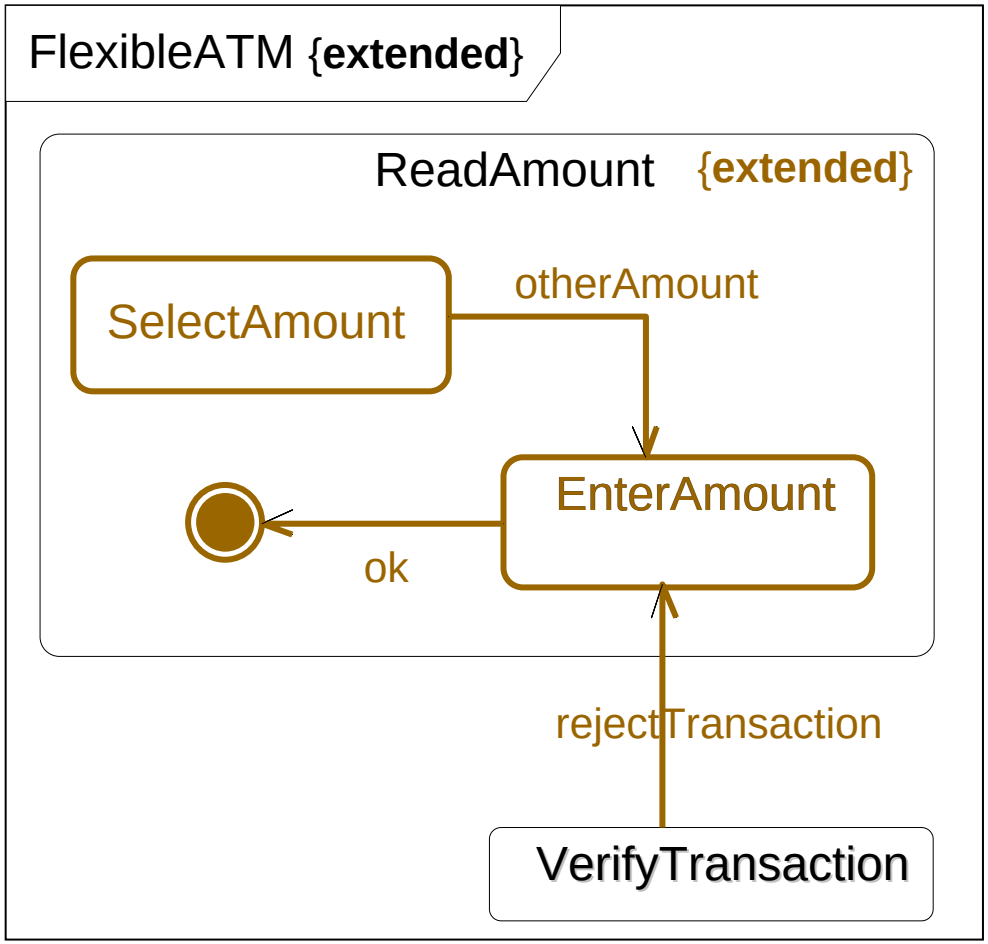
- as part of class specialization



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

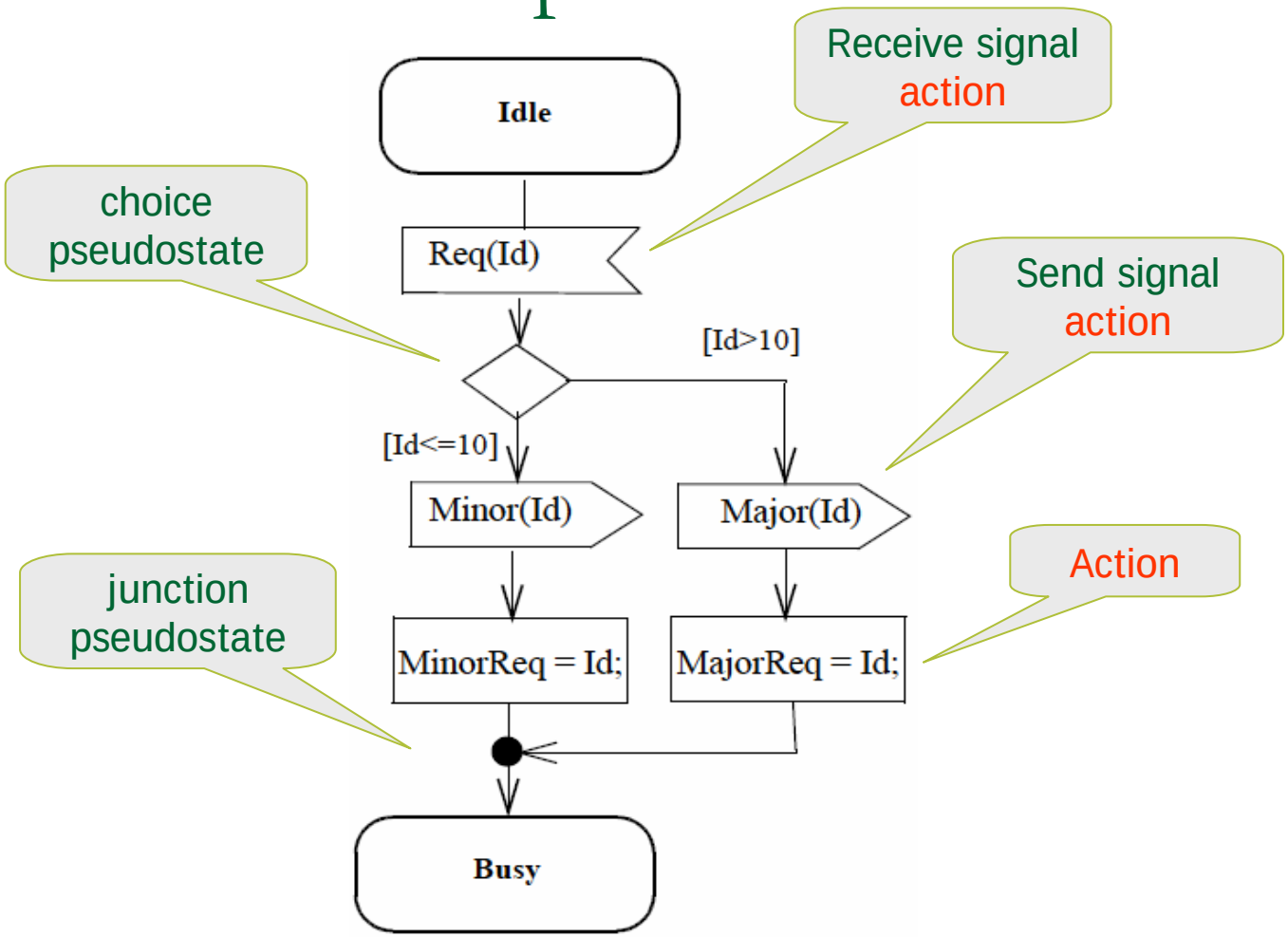
Transition

- 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*

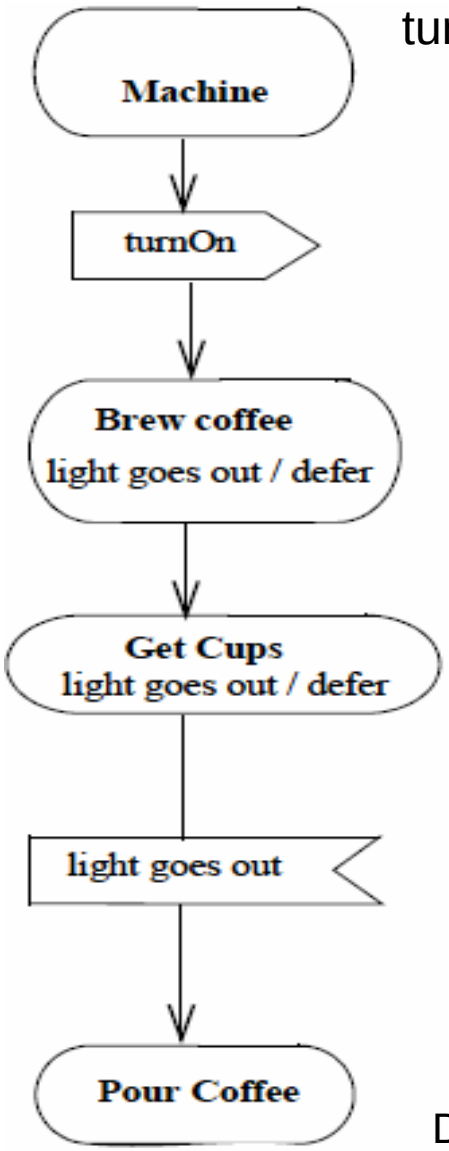
Presentation options



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

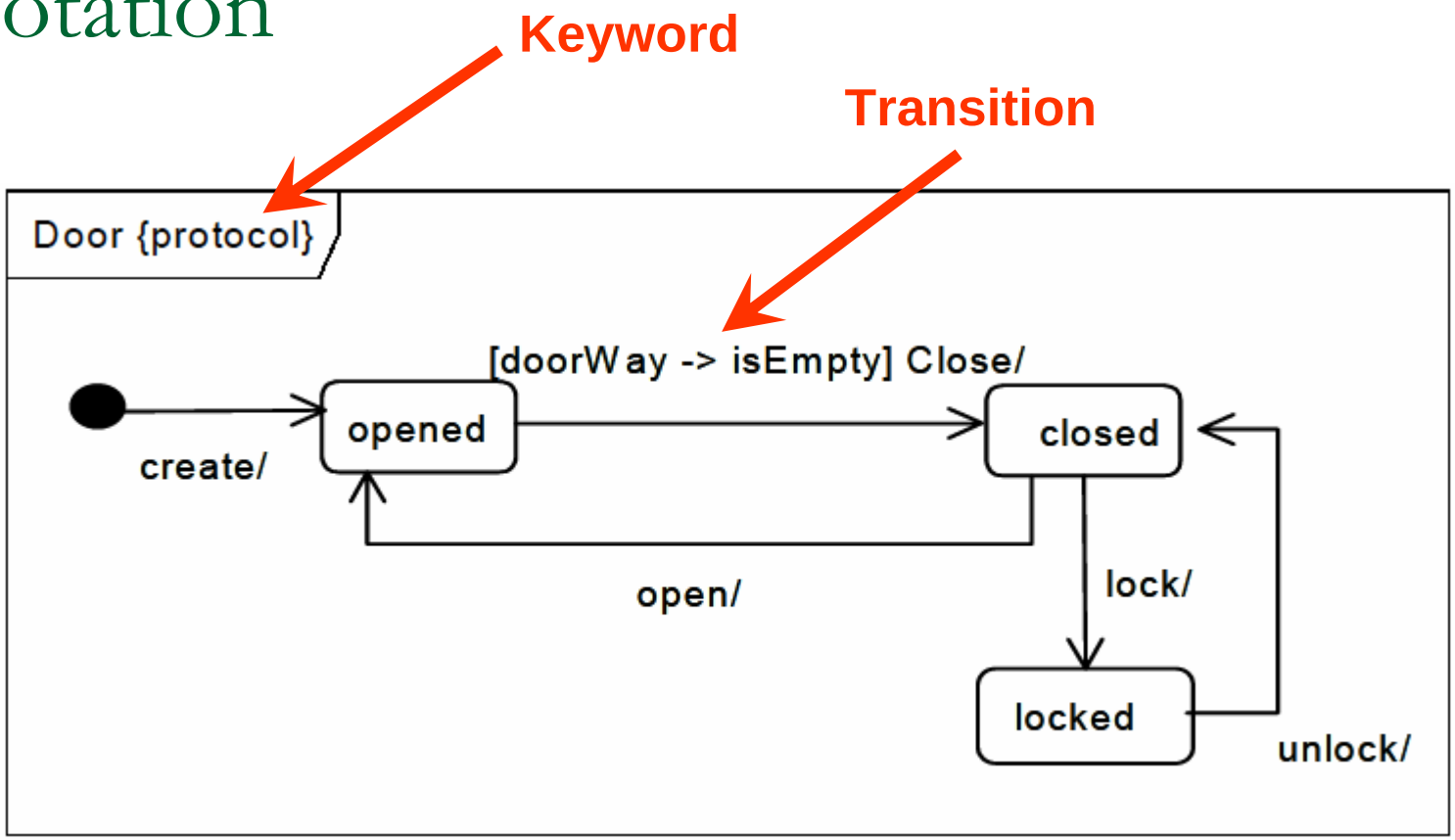
Behavioral State Machines

- 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.
- A protocol state machine is *always* defined in the *context of a classifier*.
  - 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

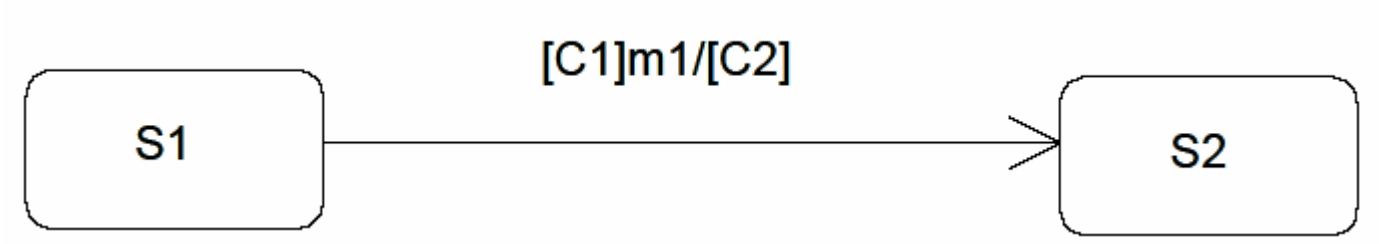
- A protocol transition specifies a legal transition for an operation.
- Transitions have the following information:
  - a pre condition (guard),
  - on trigger,
  - a post condition.
- The associated (referred) operation can be called for an instance in the origin state
- Under the initial condition (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.”

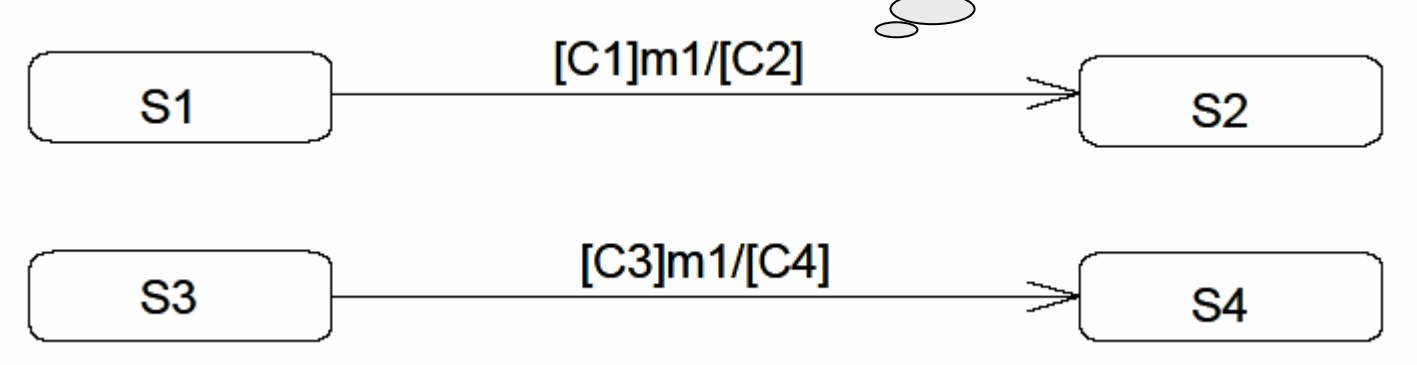


Example of a protocol transition associated to the "m1" operation

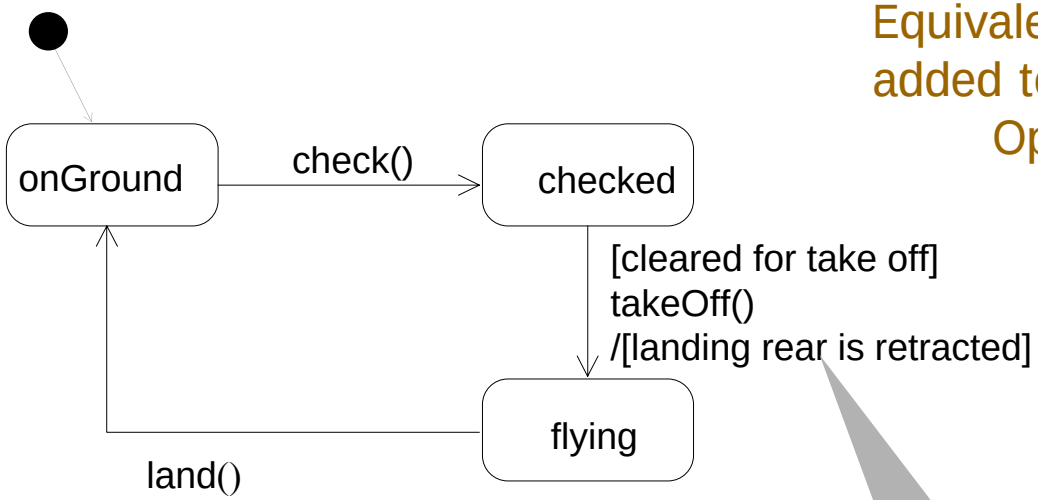
Note that

- 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 prec
  legal initial or final state is *comp*

可以看出protocol是以描述行为为中心的



Protocol State Machines



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

Operation: takeOff()

Pre

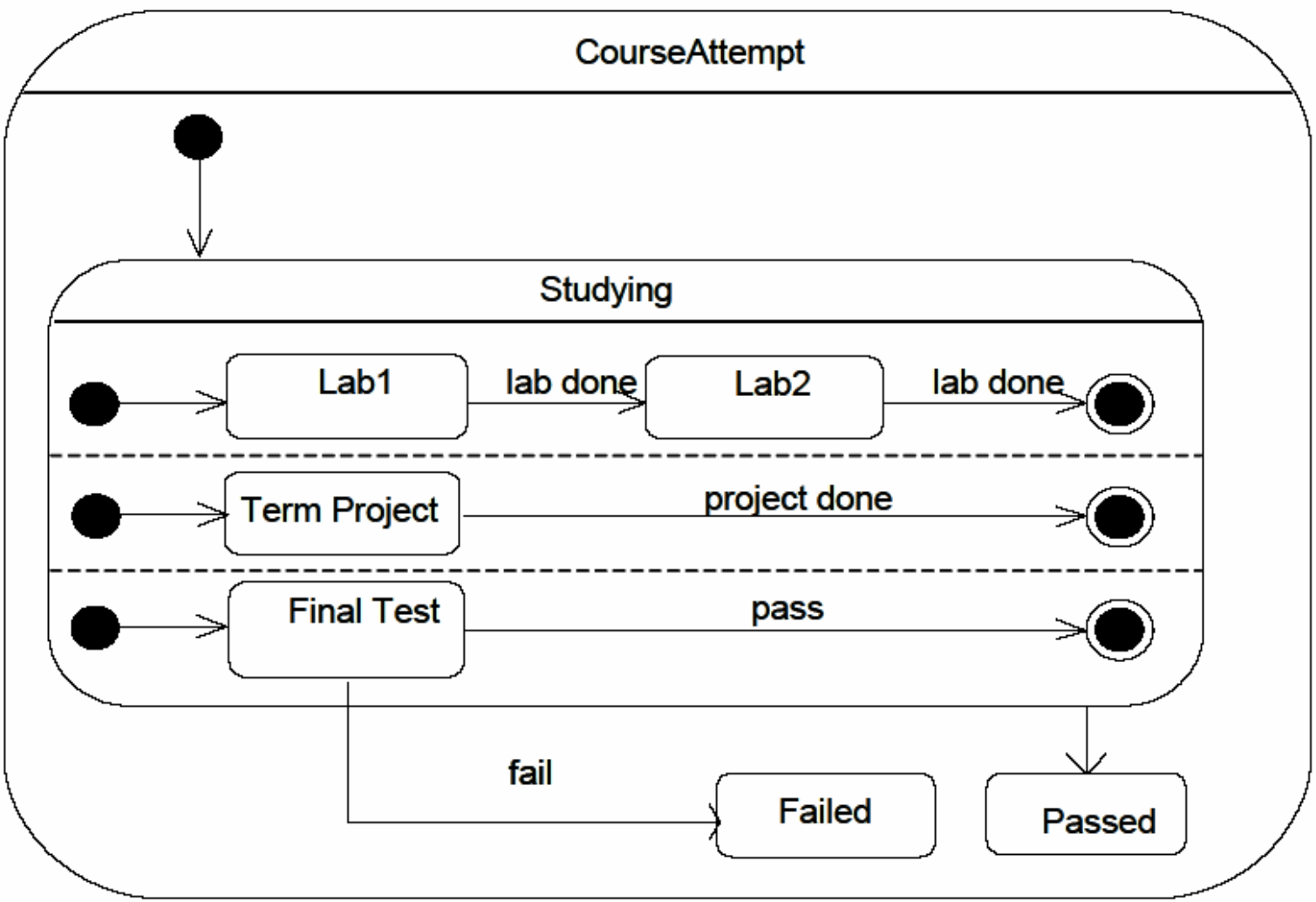
-in state "checked"
-cleared for take off

Post

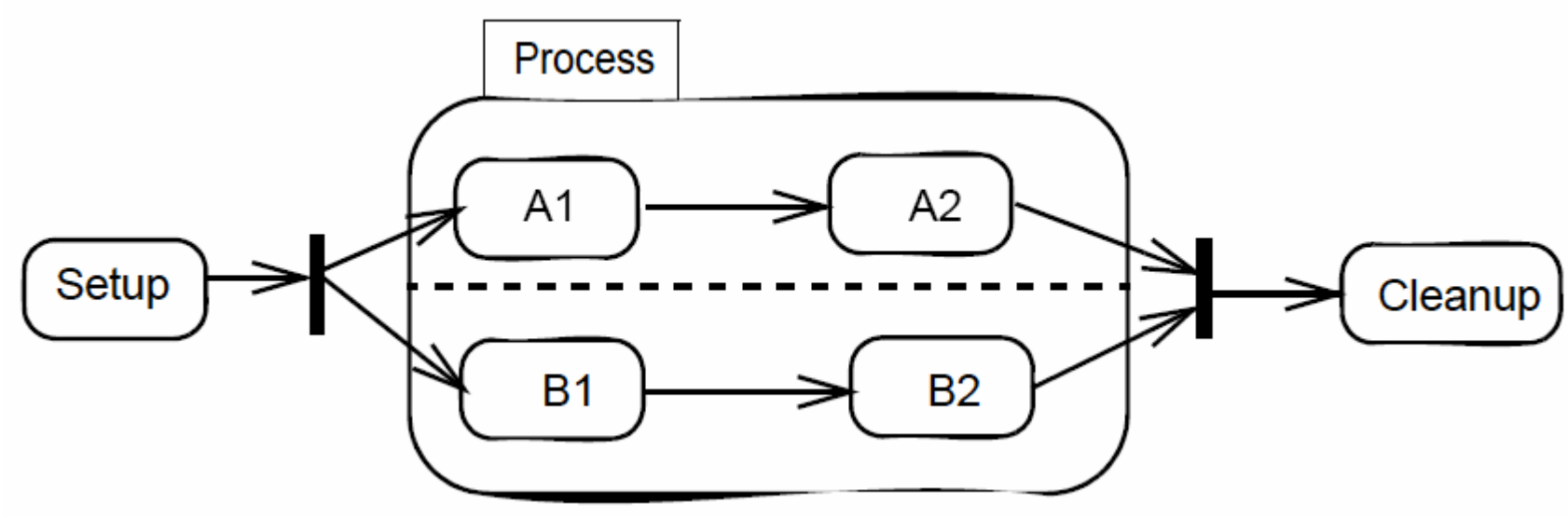
-landing rear is retracted
-in state "flying"

postcondition instead of action

Concurrency in State Machine



Concurrency in State Machine (2)



Thanks !

Appendix II - Difference of concepts ?

- reactive system
  - The behavior of a reactive system is really the set of allowed *sequences* of *input and output events*, *conditions*, and *actions*, perhaps with some additional information such as *timing constraints*.
- transformational system
  - e.g. many kinds of data-processing systems
- interactive software
- complex discrete-event system

- γ β α