Modeling System Behaviours

Mariano Ceccato

mariano.ceccato@univr.it

Notice

- Candidate exam dates (possibly subject to change)

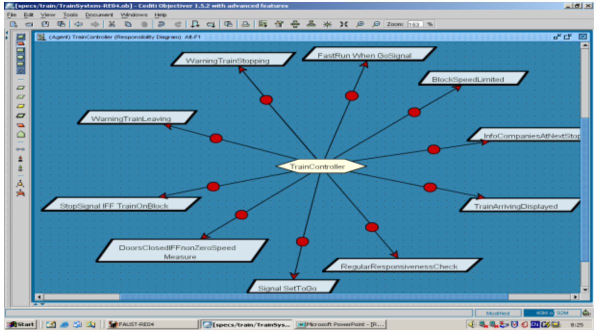
June 16th

July 14th

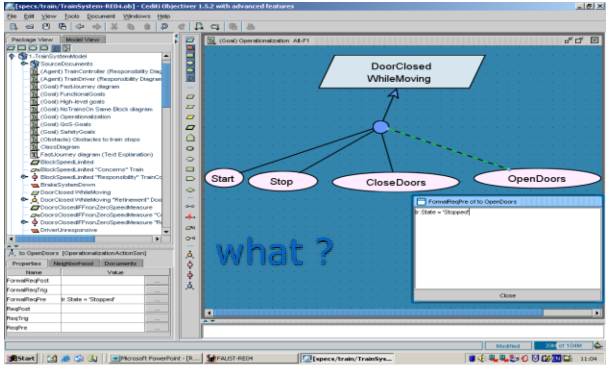
September 29th

Building models for RE

Agents & responsibilities


[specs/train/TrainSystem-RE04.ob] - Cediti Objectiver 1.5.2 with advanced features
File Edit View Tools Document Windows Help
Zoom: 153 %
(Agent) TrainController (Responsibility Diagram) Alt-F1
WarningTrainStopping
FastRun When GoSignal
BlockSpeedLimited
WarningTrainLeaving
InfoCompaniesAtNextStop
TrainController
StopSignal IFF TrainOnBlock
TrainArrivingDisplayed
DoorsClosedIFFnonZeroSpeed Measure
RegularResponsivenessCheck
Signal SetToGo
Modified
Start
FAUST-RE04
[specs/train/TrainSys...
Microsoft PowerPoint - [R...
8:25

Operations

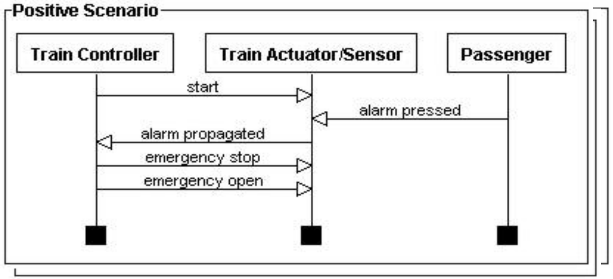

[specs/train/TrainSystem-RE04.ob] - Cediti Objectiver 1.5.2 with advanced features
File Edit View Tools Document Windows Help
Package View
Model View
1-TrainSystemModel
SourceDocuments
(Agent) TrainController (Responsibility Diag
(Agent) TrainDriver (Responsibility Diagram
(Goal) FastJourney diagram
(Goal) FunctionalGoals
(Goal) High-level goals
(Goal) NoTrainsOn Same Block diagram
(Goal) Operationalization
(Goal) QoS-Goals
(Goal) SafetyGoals
(Obstacle) Obstacles to train stops
ClassDiagram
FastJourney diagram (Text Explanation)
BlockSpeedLimited
BlockSpeedLimited "Concerns" Train
BlockSpeedLimited "Responsibility" TrainCo
BrakeSystemDown
DoorClosed WhileMoving
DoorClosed WhileMoving "Refinement" Doo
DoorsClosedIFFnonZeroSpeedMeasure
DoorsClosedIFFnonZeroSpeedMeasure "C
DoorsClosedIFFnonZeroSpeedMeasure "R
DriverUnresponsive
to OpenDoors [OperationalizationActionSon]
Properties
Neighborhood
Documents
(Goal) Operationalization Alt-F1
DoorClosed WhileMoving
Start
Stop
CloseDoors
OpenDoors
FormalReqPre of to OpenDoors
Close
Modified
39M of 104M
Start
Microsoft PowerPoint - [R...
FAUST-RE04
[specs/train/TrainSys...
11:04

Name	Value
FormalReqPost	
FormalReqTrig	
FormalReqPre	tr.State = 'Stopped'
ReqPost	
ReqTrig	
ReqPre	

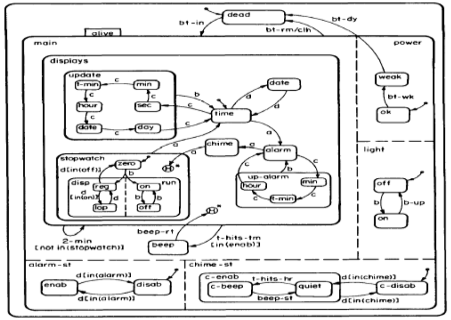
tr.State = 'Stopped'

what ?

Behaviors -*Scenarios*



Behaviors -*State machines*



The behavior model

- **System dynamics**: behavior of agents in terms of temporal sequences of state transitions for variables they control
  - **instance behaviors**: specific behaviors of specific agent instances

    -> scenarios: implicit states, explicit events
  - **class behaviors**: all possible behaviors of any agent instance

    -> state machines: explicit states, explicit causing events
- Actual behaviors (syst-as-is) or required behaviors (syst-to-be)
- Represented by UML sequence diagrams, UML state diagrams
- Multiple uses:
  - instance level: scenarios for understanding, elicitation, validation, explanation, acceptance test data
  - class level: state machines for animation, model checking, code generation

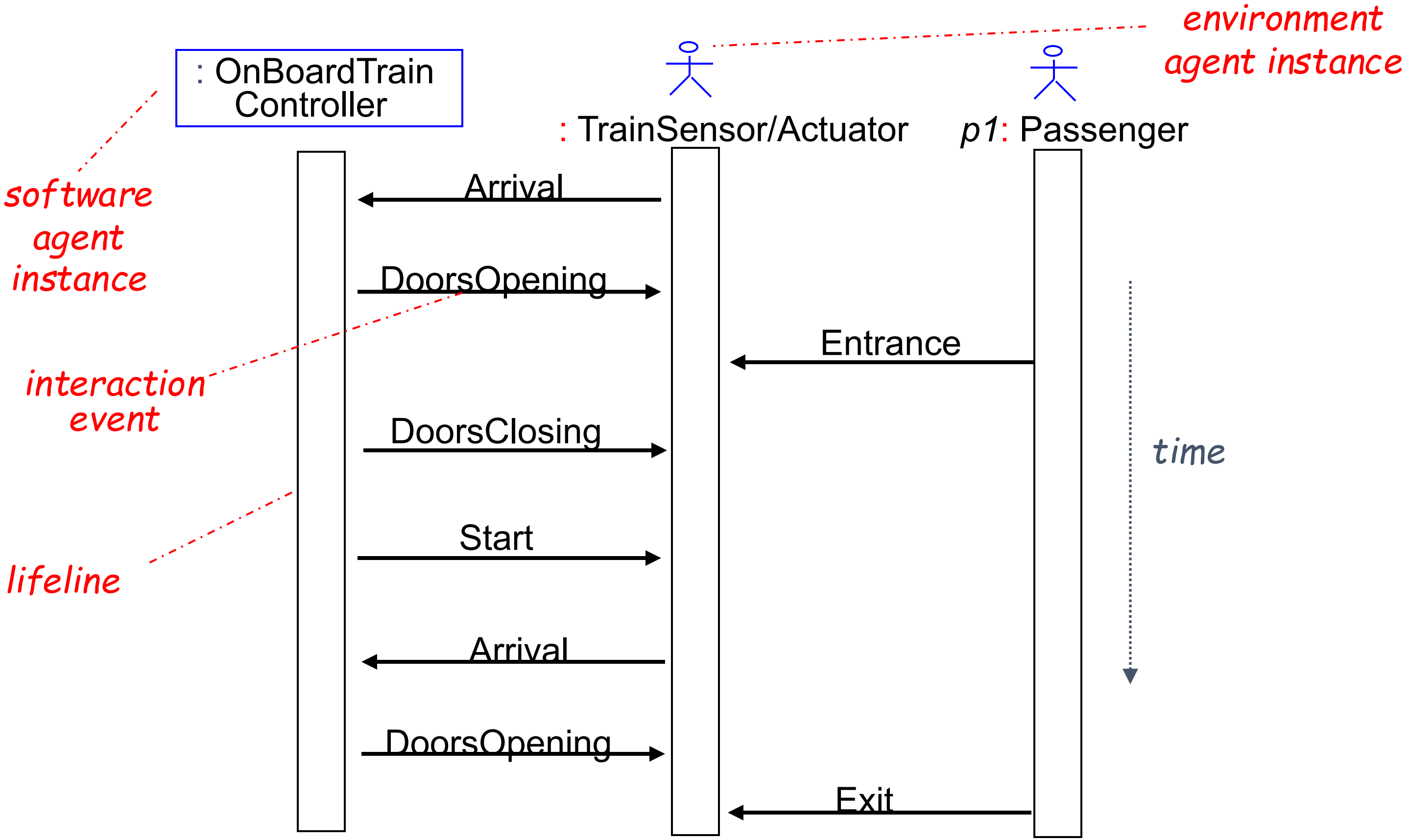
Modeling system behaviors: outline

- Modeling instance behaviors
  - Scenarios as UML sequence diagrams
  - Scenario refinement: episodes and agent decomposition
- Modeling class behaviors
  - State machines as UML state diagrams
  - State machine refinement: sequential & concurrent substates
- Building behavior models
  - Elaborating relevant scenarios for good coverage
  - Decorating scenarios with explicit state conditions
  - From scenarios to state machines
  - From scenarios to goals
  - From operationalized goals to state machines

Modeling instance behaviors through scenarios

- **Scenario** = temporal sequence of **interaction events** among agent instances
  - instances of different agents or of same agent
  - interactions are **directed**
    - **from** source agent instance, <u>controlling</u> the event,
    - **to** target agent instance, <u>monitoring</u> the event
  - interactions are **synchronous** among event controller/monitor
- **Positive** scenario: illustrates some way of achieving implicit goal(s)
  - normal scenario: in normal cases
  - abnormal scenario: in exception cases (don’t forget these!)
- **Negative** scenario: illustrates some inadmissible behavior (obstacle)

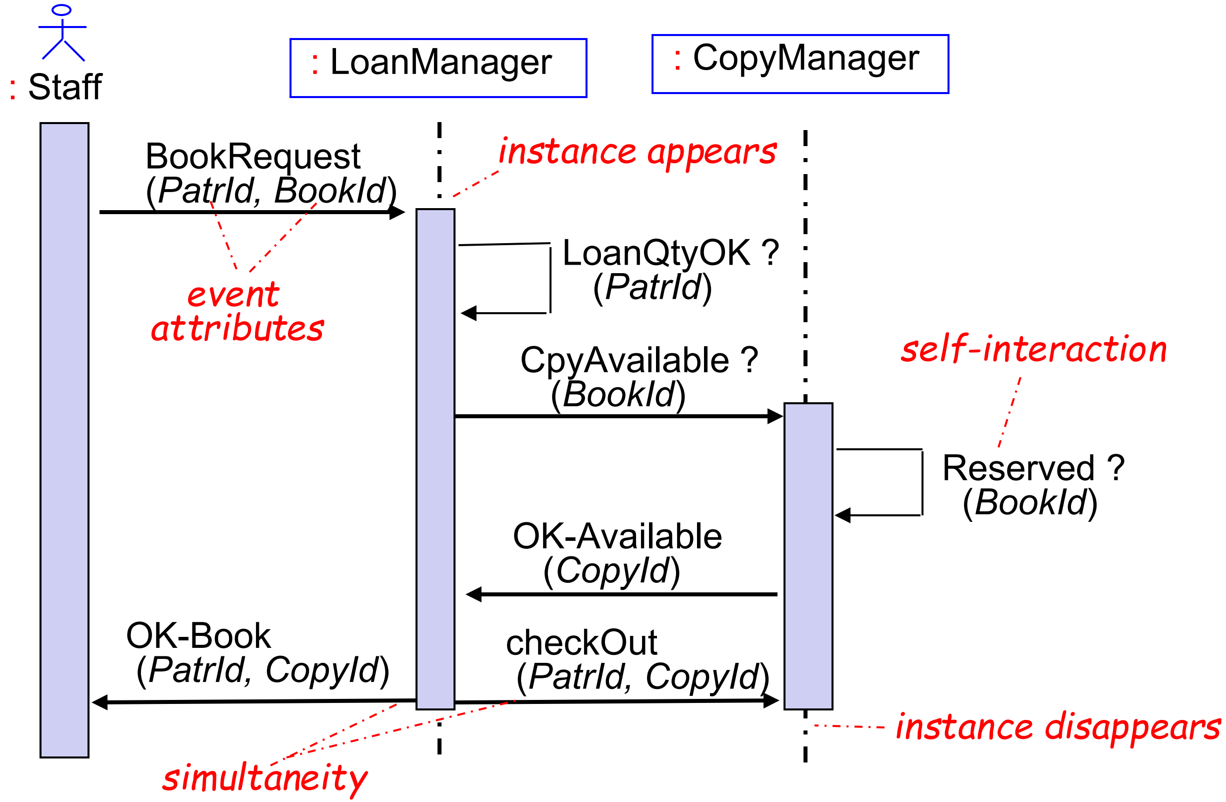
Scenarios as UML sequence diagrams



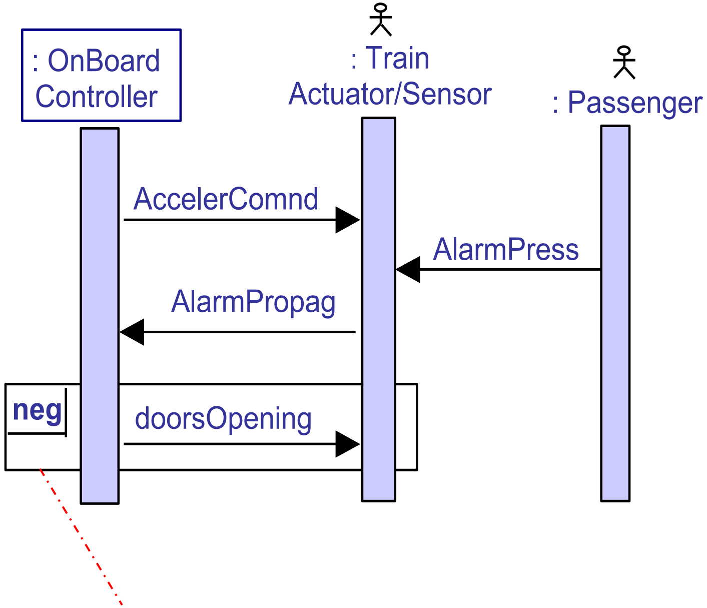
Scenarios as UML sequence diagrams

- **Event** = instantaneous conceptual object
  - instances exist in **single** system states only
  - can be structured in the object model (cf. "conceptual objects" lecture)
    - attributes, structural associations
    - used for **information transmission** along interaction
    - event specialization/generalization with inheritance
- Interaction events correspond to applications of **operations**
  - by source agent, notified to target agent (cf. operation model)
- A sequence diagram defines ...
  - a **total order** on events along an agent's lifeline (precedence)
  - a **partial order** on all scenario events
    - independent events on different lifelines are not comparable under precedence
- A scenario may be composed of episodes (sub-scenarios)

Scenarios as UML sequence diagrams



UML sequence diagrams: negative scenarios, optional interactions



: OnBoard Controller

: Train Actuator/Sensor

: Passenger

AccelerComnd

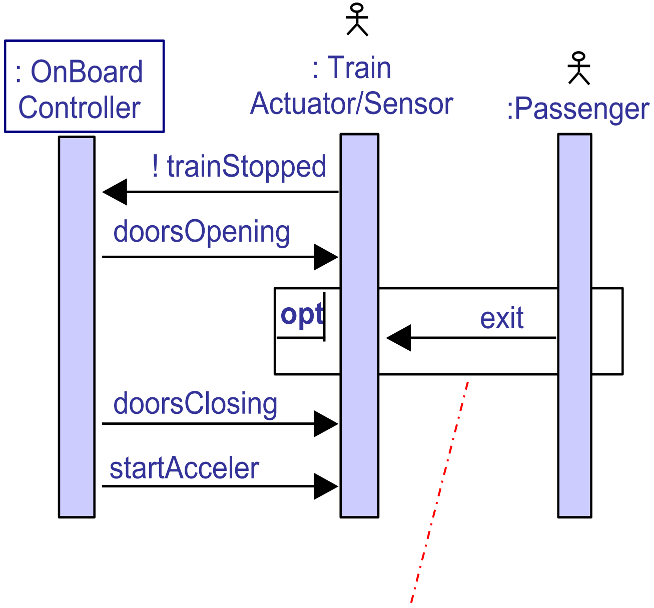
AlarmPress

AlarmPropag

neg

doorsOpening

prohibited interaction



: OnBoard Controller

: Train Actuator/Sensor

:Passenger

! trainStopped

doorsOpening

opt

exit

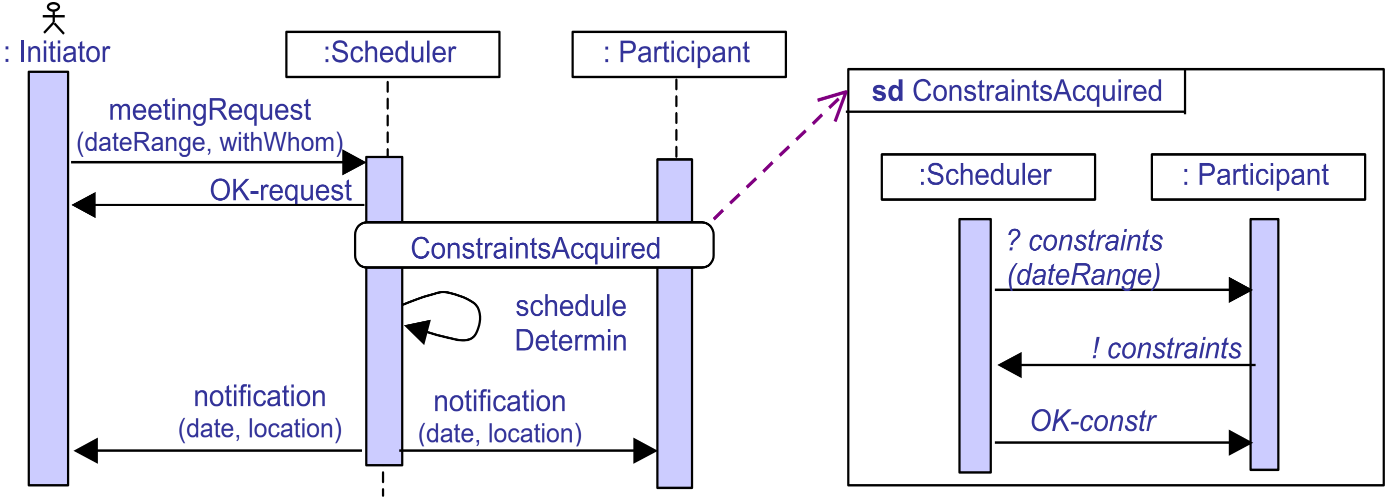
doorsClosing

startAcceler

optional interaction

Scenario refinement: episodes

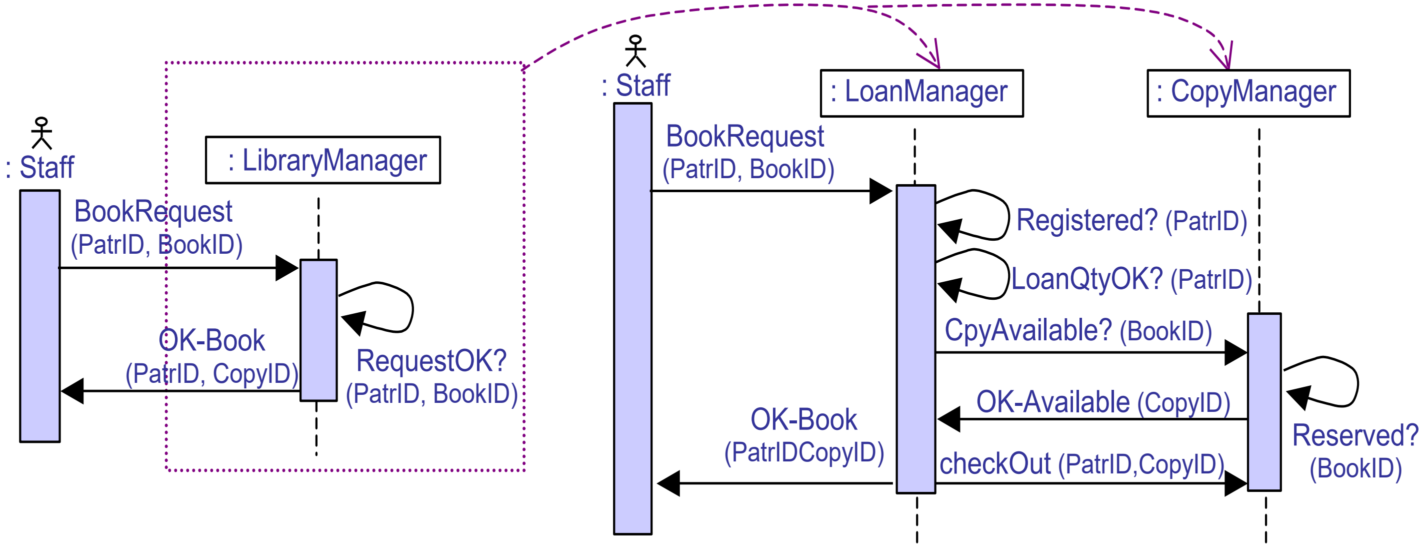
- Episode = subsequence of interactions for specific subgoal
- Appears as coarse-grained interaction
- To be detailed in another diagram with specific interactions
- Helpful for incremental elaboration of complex scenarios



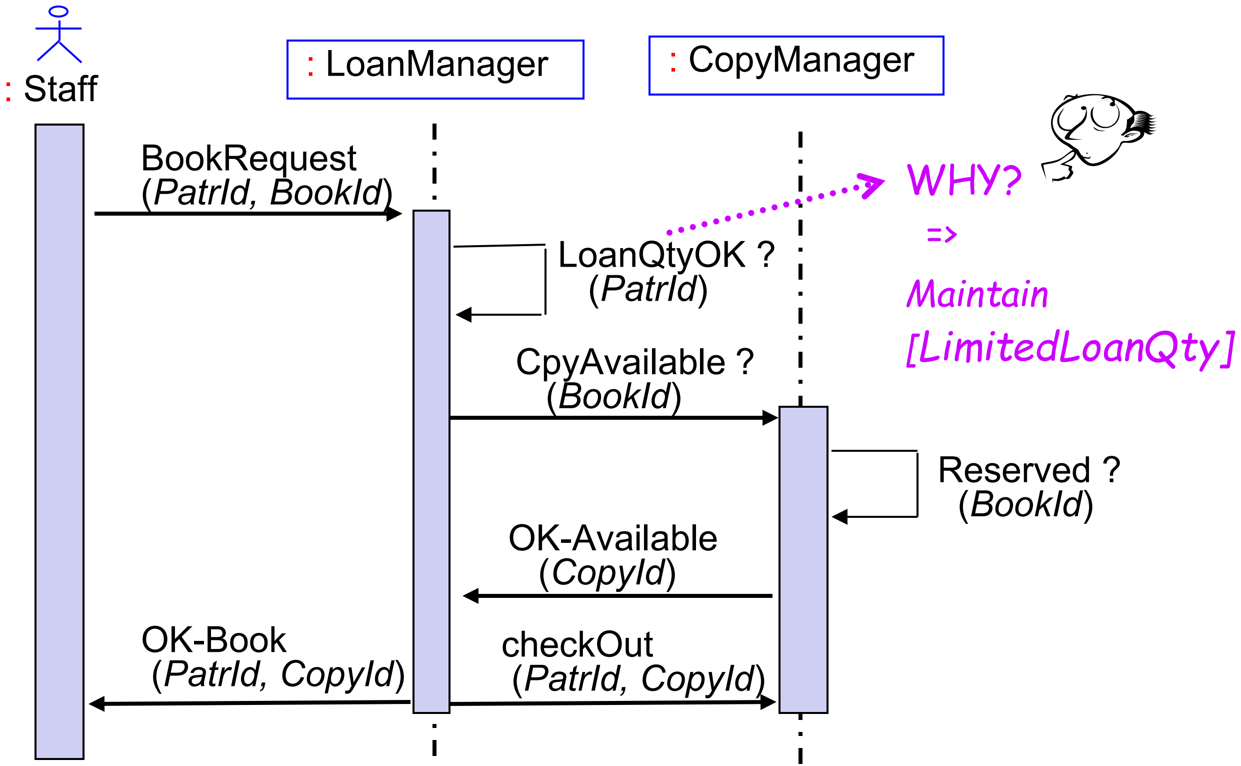
reference to another diagram

Scenario refinement: agent decomposition

- Coarse-grained agent instances may subsequently be decomposed into finer-grained ones
- With finer-grained interactions

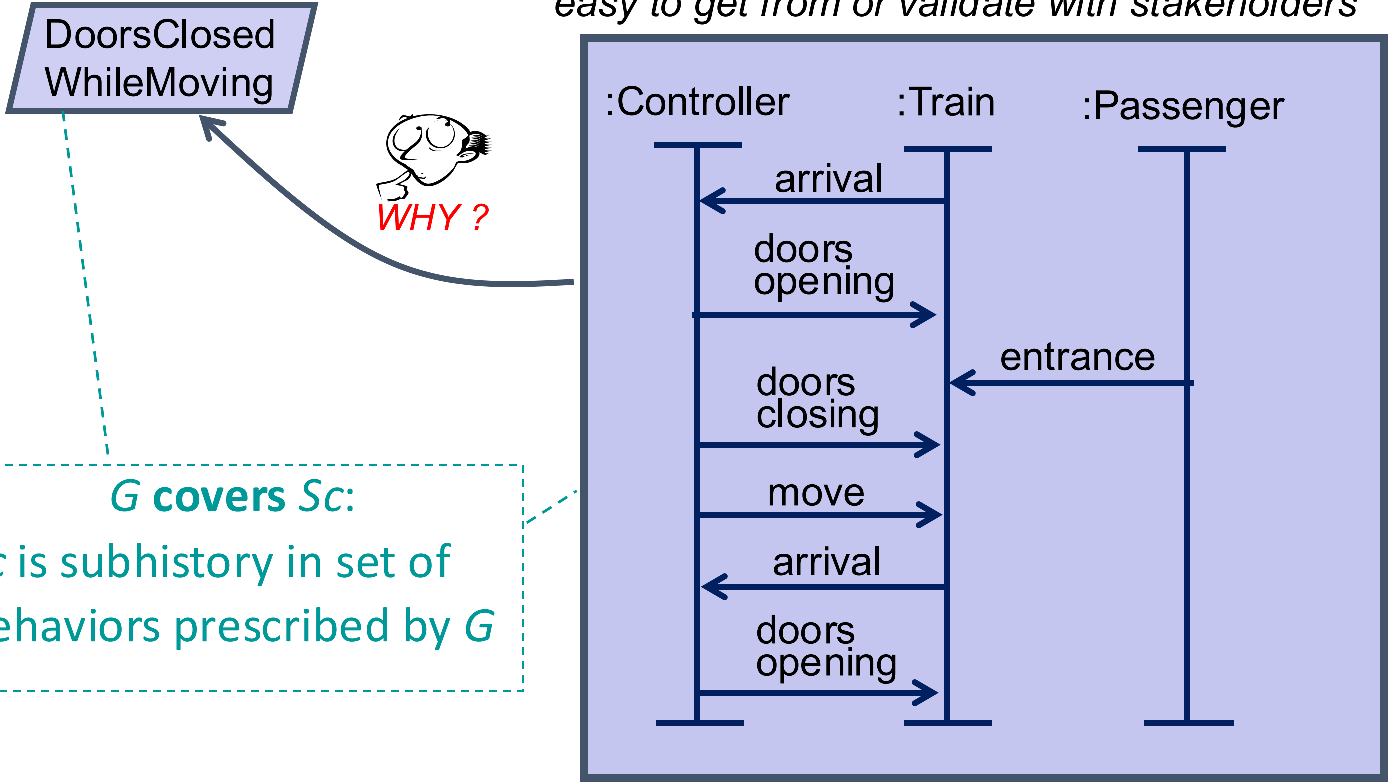


Scenarios are concrete vehicles for goal elicitation



Scenarios are concrete vehicles for goal elicitation

easy to get from or validate with stakeholders



G **covers** *Sc*:

Sc is subhistory in set of behaviors prescribed by *G*

Modeling system behaviors: outline

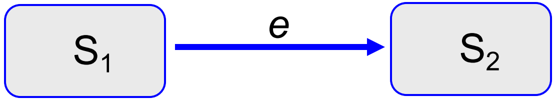
- Modeling instance behaviors
  - Scenarios as UML sequence diagrams
  - Scenario refinement: episodes and agent decomposition
- Modeling class behaviors
  - State machines as UML state diagrams
  - State machine refinement: sequential & concurrent substates
- Building behavior models
  - Elaborating relevant scenarios for good coverage
  - Decorating scenarios with state conditions
  - From scenarios to state machines
  - From scenarios to goals
  - From operationalized goals to state machines

State machines

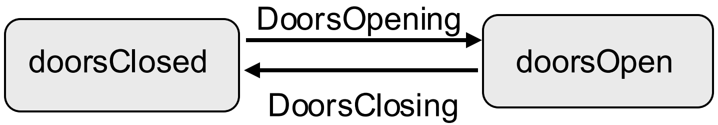
- A state machine (SM) is specified by a transition relation

$$\text{tr: } S \times E \rightarrow S$$

  - *S*: set of explicit states (usually finite), for any class instance
  - *E*: finite set of events causing state transitions
  - Preferably, *tr* is a function: deterministic SM
- Graphically, tr (S1, e) = S2 is represented by ...



- Semantics: the transition to S_2 gets fired iff we are in state S_1 and event *e* occurs



SM states vs. snapshot states

- **Snapshot state** of an object instance (cf. "conceptual objects" lecture):

  tuple of functional pairs $\mathbf{x_i} \mapsto \mathbf{v_i}$

  $\mathbf{x_i}$: state variable (object attribute, association)

  $\mathbf{v_i}$: corresponding value

  e.g. (tr.Speed |→ 0, tr.Location |→ 9.25, tr.DoorsState |→ Open, On |→ (tr, block13), At |→ (tr, platform1))

- **SM state** of object instance: **class** of snapshot states sharing same value for some behavioral state variable (equivalence class)

  e.g.

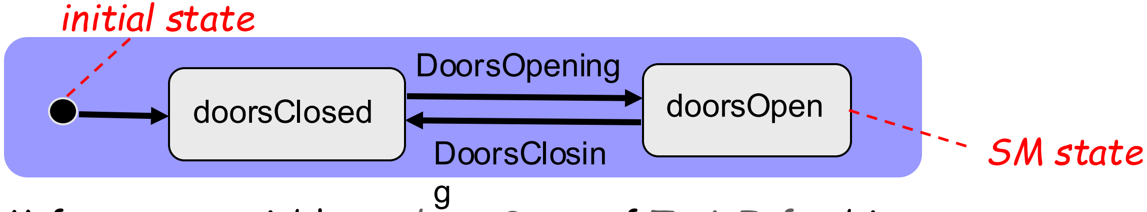
  SM state doorsClosed of Train instance tr includes snapshot states

  { tr.Speed |→ 0, tr.Loc | → 3, tr.DoorsState | → 'closed', At | → (tr, pl1) } ,

  { tr.Speed | → 5, tr.Loc | → 9, tr.DoorsState | → 'closed', At | → (tr, nil) }

SM states

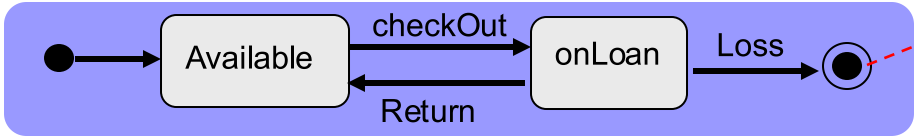
- To model agent behaviors: **one** state machine **per state variable controlled** by an arbitrary agent instance
  - captures admissible sequences of transitions among SM states of corresponding object instance --for **this** variable
- A SM state has some duration
  - corresponding object instance remains some time in it
- **Initial state** = state when object instance appears in system
  - InstanceOf (o, Ob) gets *true*



SM for state variable *tr.doorsState* of TrainInfo object controlled by TrainController

SM states

- **Final state** = state when object instance disappears from system
  - InstanceOf (o, Ob) gets *false*

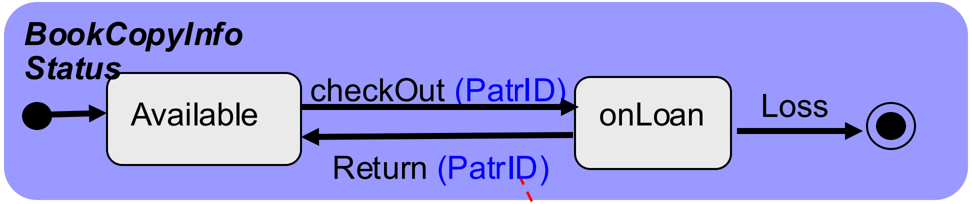


final state

SM for state variable *bc.Status* of BookCopyInfo object controlled by LibraryManager

SM events

- Event instances are instantaneous phenomena
  - (As seen before) event = object whose instances exist in single states
    - InstanceOf (ev, E) denoted by Occurs (E)
  - Can be structured
    - attributes, associations, specializations, to be declared in object model
    - corresponding attribute values can be attached to events in SM
  - No duration, unlike SM states



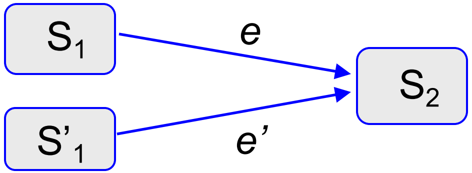
attribute value

Typology of SM events

- **External event**: not controlled by agent associated with SM
  - **temporal event**
    - elapsed time period e.g. **after** (3secs), **after** (Timeout)
    - clock state change e.g. **when** (12:00pm)
  - **external stimulus**: event occurring in SM controlled by another agent
    - state change, condition becoming true
    - to be notified from that other SM (see below)
    - e.g. TrainStart, PassengerAlarm
- **Internal event**: controlled by agent associated with this SM
  - application of operation performed by the agent
    - e.g. DoorsClosing is an application of operation CloseDoors
  - **Tip**: use suggestive verb for operation, corresponding noun for operation application

Events & state transitions

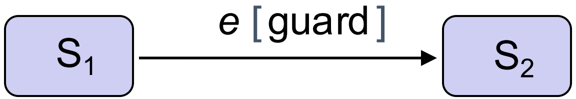
- **State transition** = state change caused by event occurrence



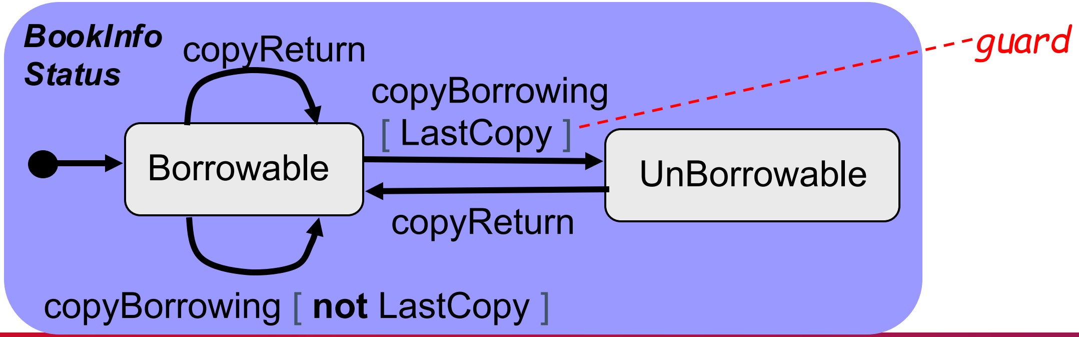
  - the associated object instance gets to target state S_2 iff...
    - it is in source state S_1 and instance of event *e* occurs; OR
    - it is in source state S'_1 and instance of event *e'* occurs
- **Automatic transition** = no event label
  - fires without waiting for event occurrence

Guarded transitions

- Transitions may also have a guard label
  - guard = Boolean expression on state variables

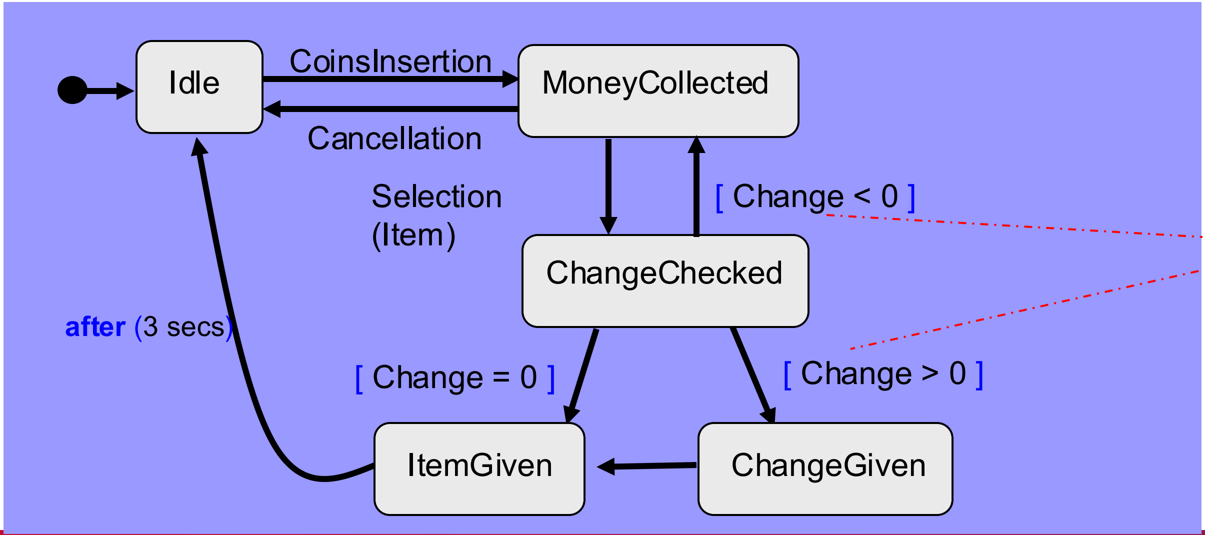


- Necessary condition for transition firing:
  - the associated object instance gets into state S_2
    - **if** it is in state S_1 and instance of event *e* occurs
    - and **only if** the guard is true



Guarded transitions

- Do not confuse
  - necessary condition for transition firing: guard true
  - sufficient condition for transition firing: event occurrence
- **Trigger condition**: guard on transition with no event label
  - automatic transition with guard => necessary/sufficient cond

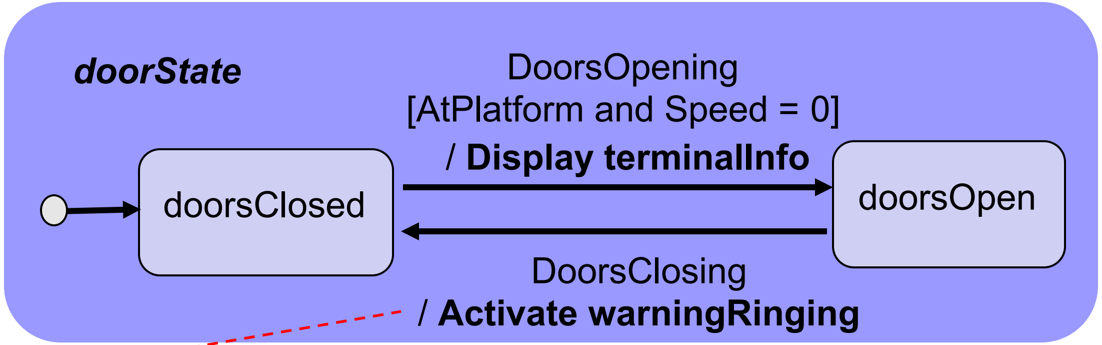


as soon as
guard
gets true

SM for
state variable
controlled by
VendingMachine

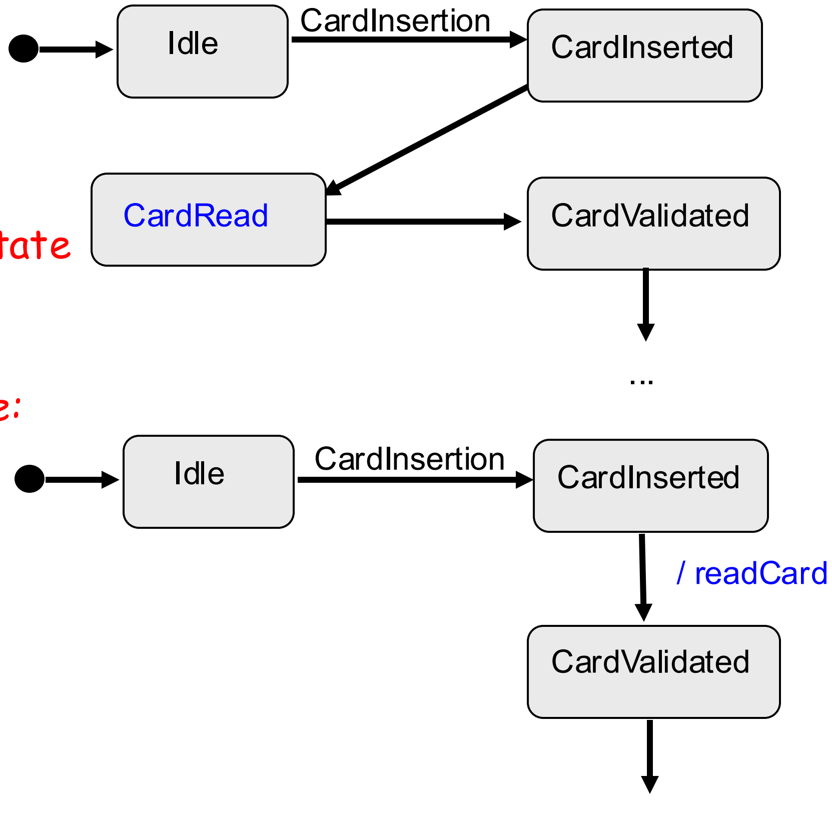
Auxiliary actions in a state diagram

- **Action** = operation associated with a transition
  - to be applied when transition fires
  - atomic
  - no meaningful effect on dynamics captured by SM states:
    - state resulting from operation application would clutter diagram
  - typically, info display/acquisition to/from agent's environment



action

Avoiding irrelevant states through actions: example



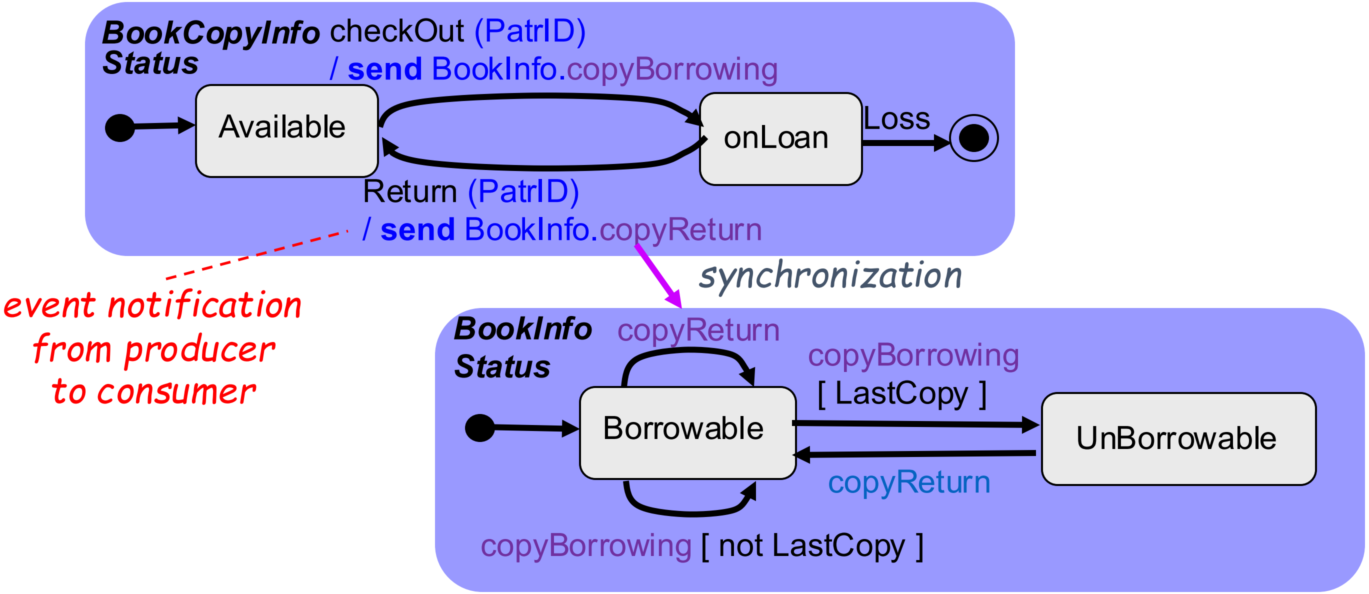
not a meaningful state

alternative:

Event notification

Important subclass of actions

- event is notified from **producing** diagram to **consuming** diagram
  - causes transitions in consuming diagram => diagram synchronization

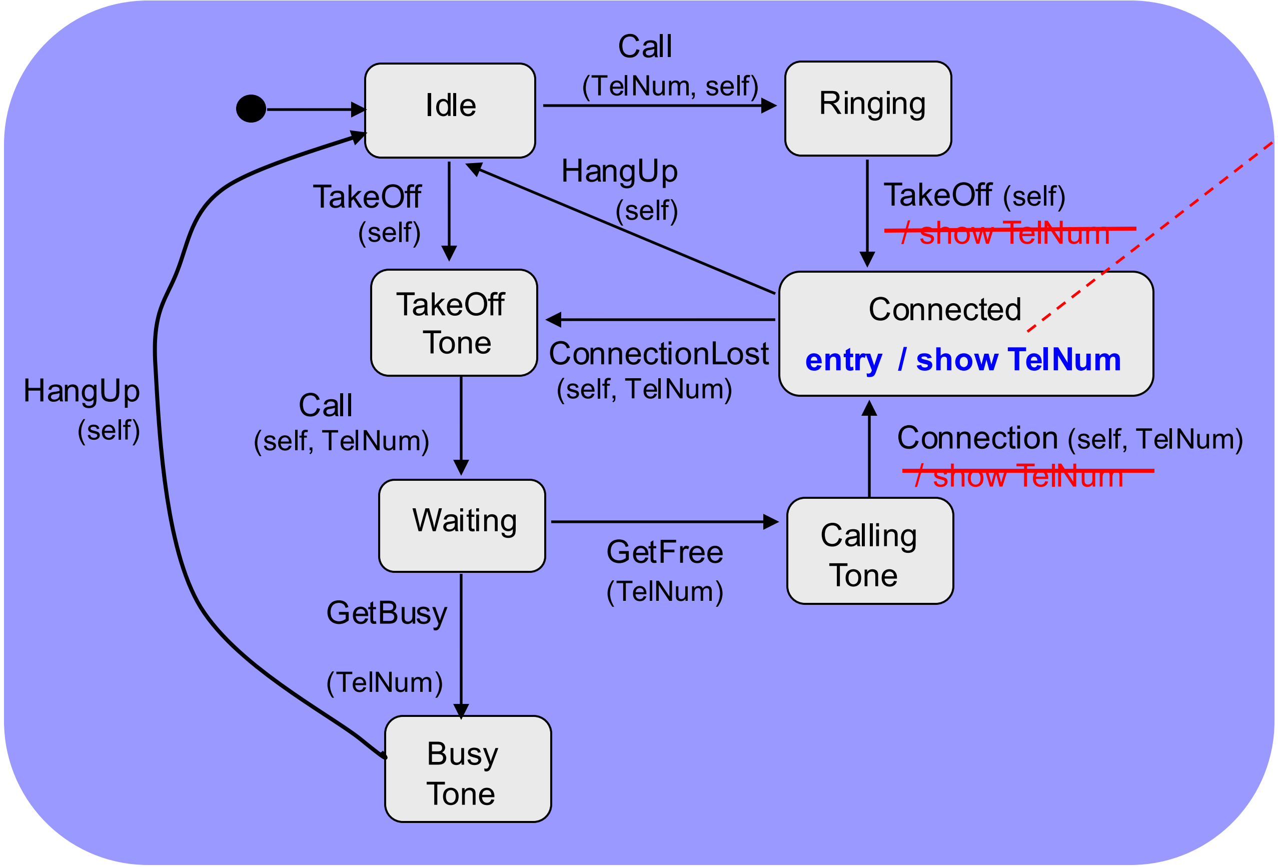

BookCopyInfo Status
checkOut (PatrID)
/ send BookInfo.copyBorrowing
Available
onLoan
Loss
Return (PatrID)
/ send BookInfo.copyReturn
synchronization
event notification from producer to consumer
BookInfo Status
copyReturn
copyBorrowing [LastCopy]
Borrowable
UnBorrowable
copyReturn
copyBorrowing [not LastCopy]

Entry/exit actions

- Entry action
    - within state, prefixed by “entry”
    - amounts to all incoming transitions (for a state) labelled with this action
- Exit action
    - within state, prefixed by “exit”
    - amounts to all outgoing transitions (for a state) labelled with this action

=> avoids action duplication in diagrams

Entry/exit actions: example



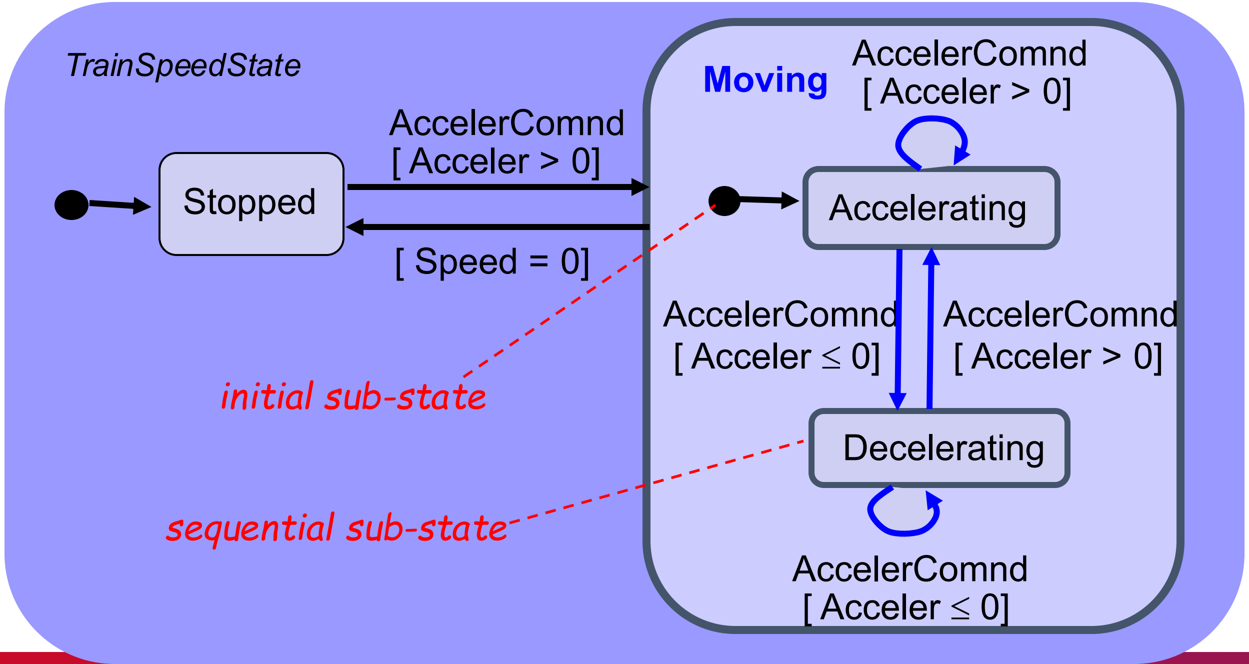
entry
action

Nested states for SM structuring

- SM states can be decomposed into sub-states
  - convenient for incremental elaboration of complex SM
    - coarse-grained states refined into finer-grained states
- Sequential decomposition:
  - coarse-grained state becomes SM on sequential sub-states: visited sequentially
- Parallel decomposition:
  - coarse-grained state becomes SM on concurrent sub-states: visited concurrently
- Structuring mechanisms in UML borrowed from Statecharts

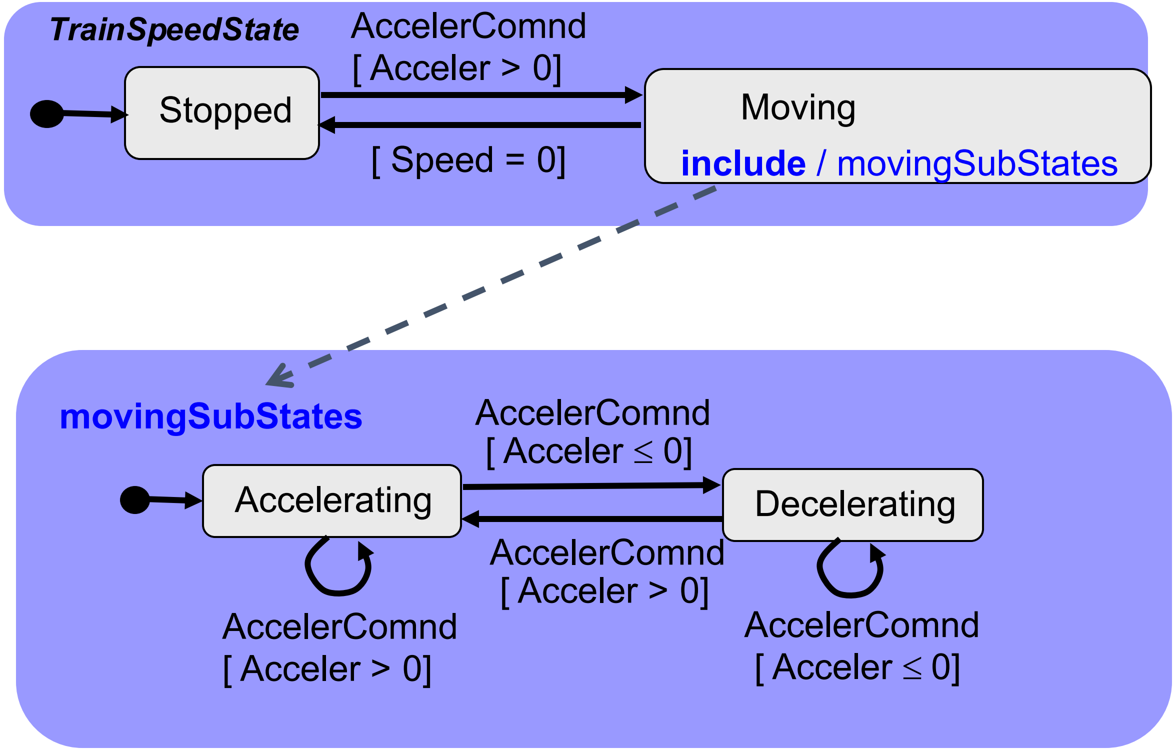
SM refinement: sequential decomposition

- Super-state is a diagram composed of sequential sub-states connected by new transitions
  - contains the nested sub-states (graphical nesting)
  - **or** contains “include” reference to other diagram



TrainSpeedState
Moving
AccelerCmnd [Acceler > 0]
Stopped
AccelerComnd [Acceler > 0]
[Speed = 0]
Accelerating
AccelerComnd [Acceler ≤ 0]
AccelerComnd [Acceler > 0]
Decelerating
AccelerComnd [Acceler ≤ 0]
initial sub-state
sequential sub-state

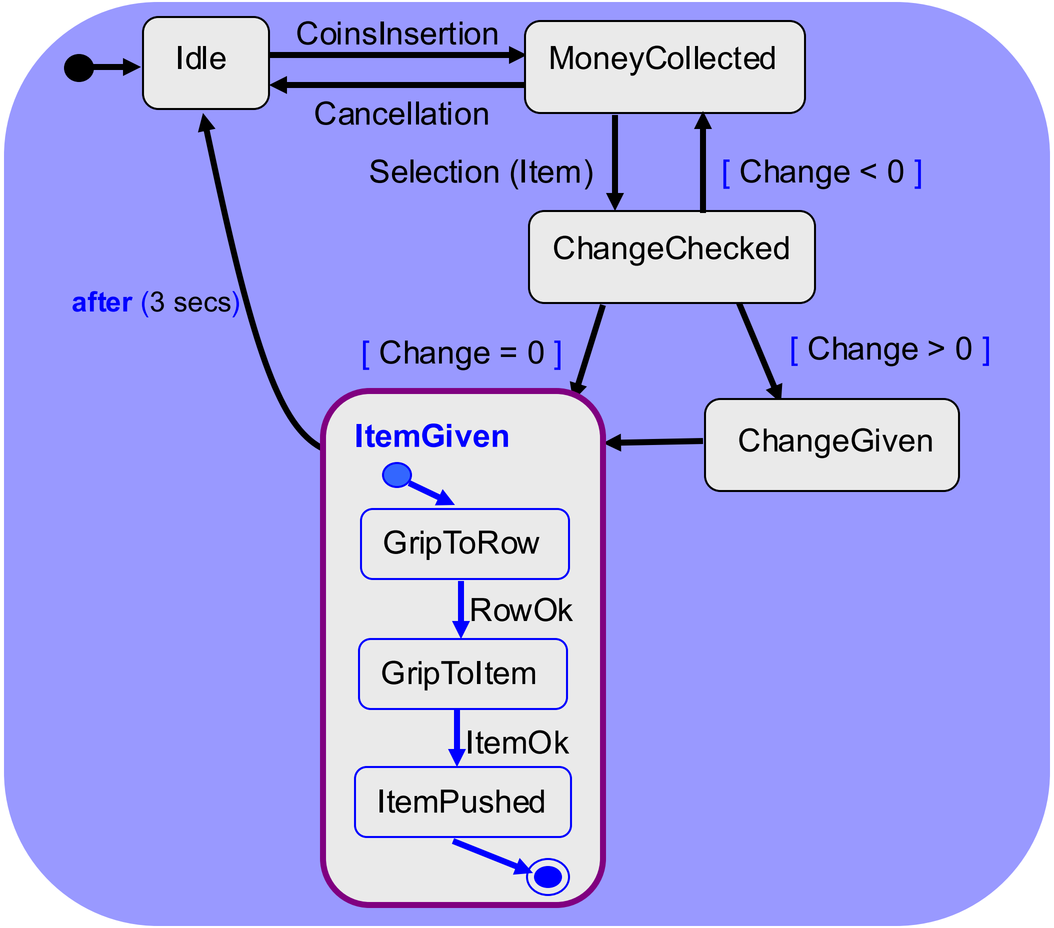
Sequential decomposition: same example with include reference



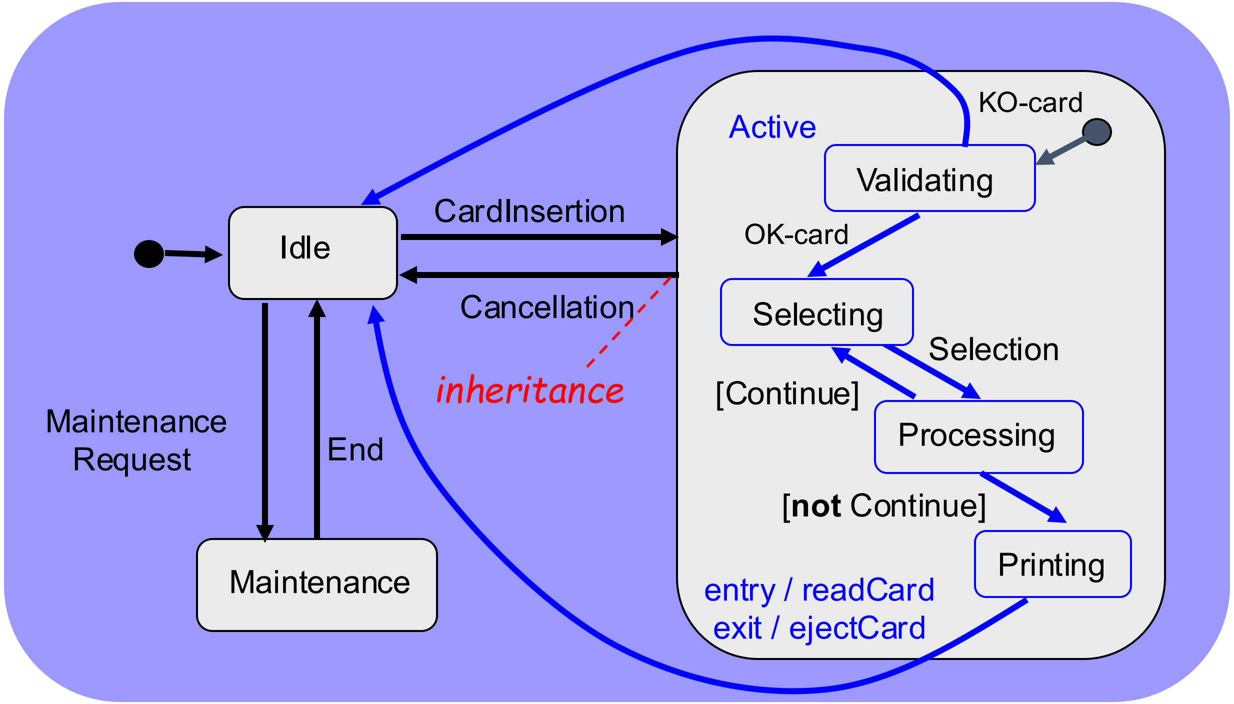
Sequential decomposition: semantic rules

- The object instance is in super-state iff it is in **one** (and only one) **of** the nested sub-states
- Every incoming or outgoing transition of super-state is by default **inherited** by each sub-state
  - substates may have their own incoming/outgoing transitions
    - within super-state or to external states
- To inhibit transition inheritance by each substate ...
  - for incoming transition to super-state: insert **initial sub-state** as predecessor of sub-state where to start
    - forces to start from there, not from anywhere
  - for outgoing transition from super-state: insert **final sub-state** as successor of sub-state where to leave
    - forces to leave from there, not from anywhere

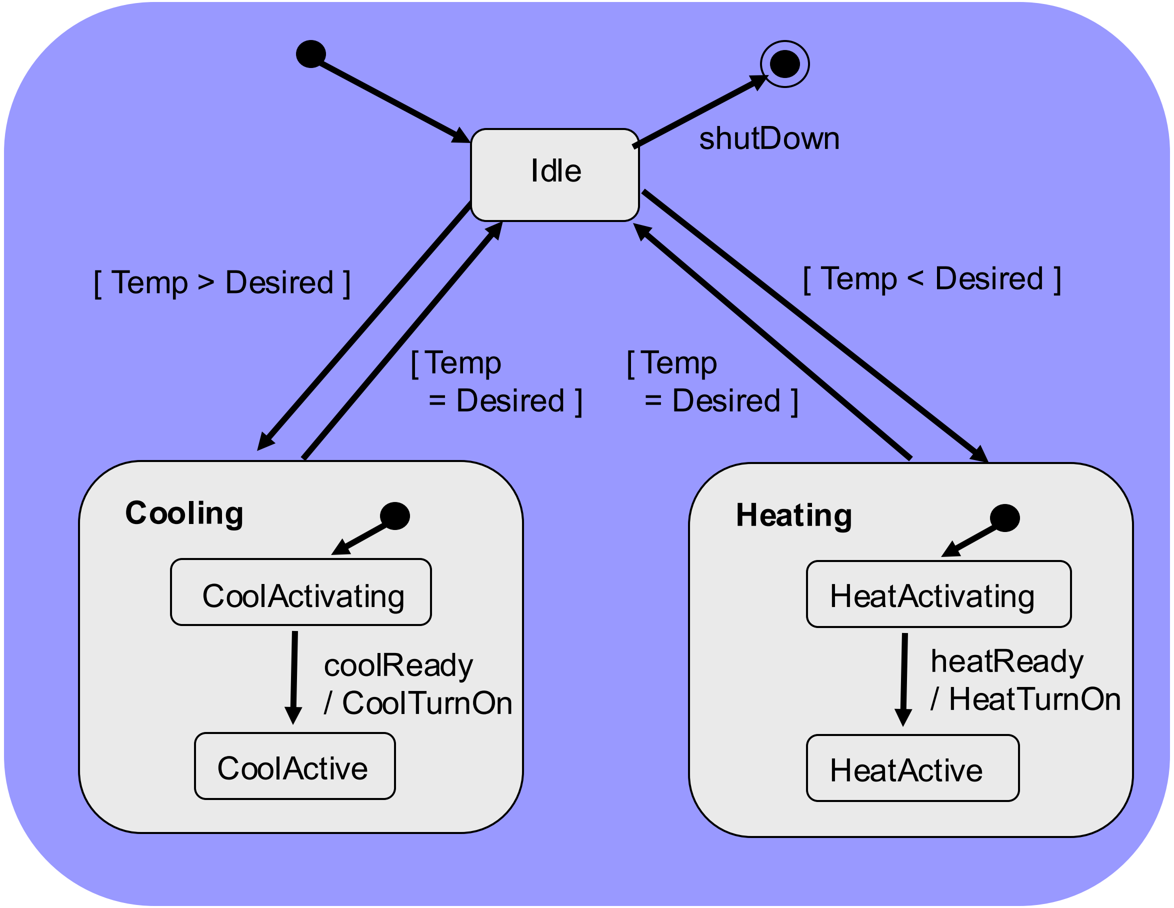
Sequential decomposition: vending machine ex.



Sequential decomposition: cash machine ex.



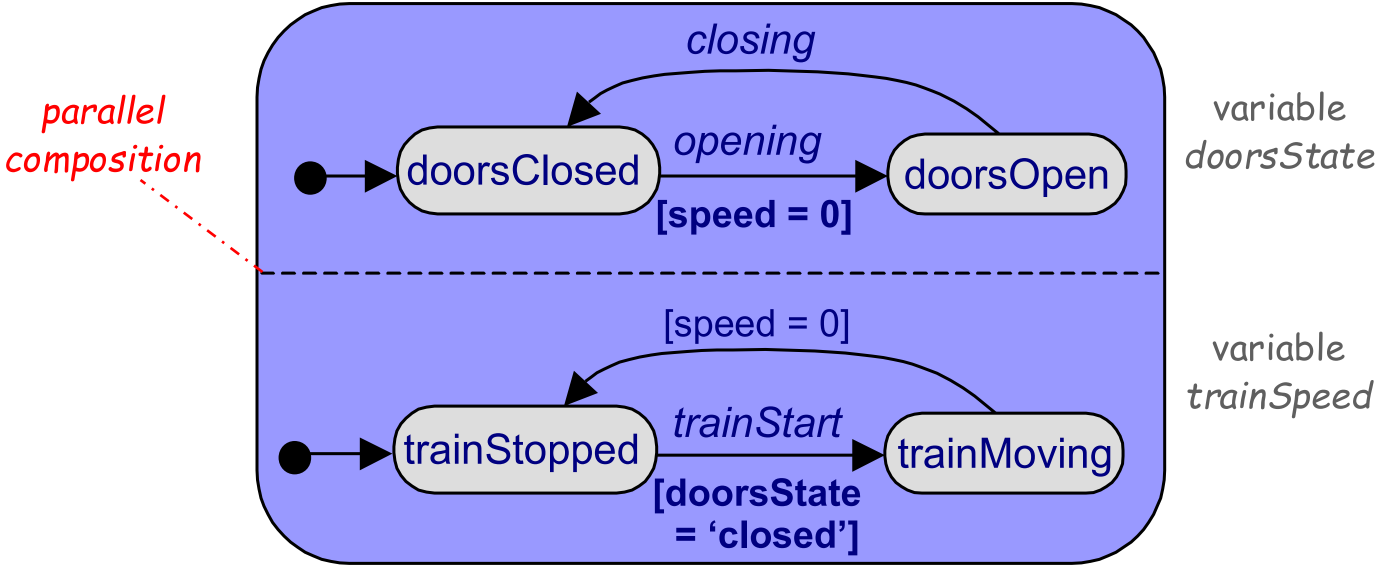
Sequential decomposition: thermostat controller



Parallel decomposition: concurrent behaviors

- Agents often control *multiple* items in parallel
- Problems with flat SM diagram:
  - N item variables each with *M* values => M^N states !
  - same SM state mixing up different variables
- **Statechart** = parallel composition of SM diagrams
  - one per variable evolving in parallel
  - statechart **state** = aggregation of concurrent substates
    - each may be recursively decomposed sequentially (or in parallel)
  - from M^N explicit SM states to **M × N** statechart states !
- Statechart trace = sequence of successive aggregated SM states up to some point
- Interleaving semantics: for 2 transitions firing in same state, one is taken after the other (non-deterministic choice)

Concurrent sub-states: example



- Trace example:

  < (doorsClosed, trainStopped); (doorsClosed, trainMoving);

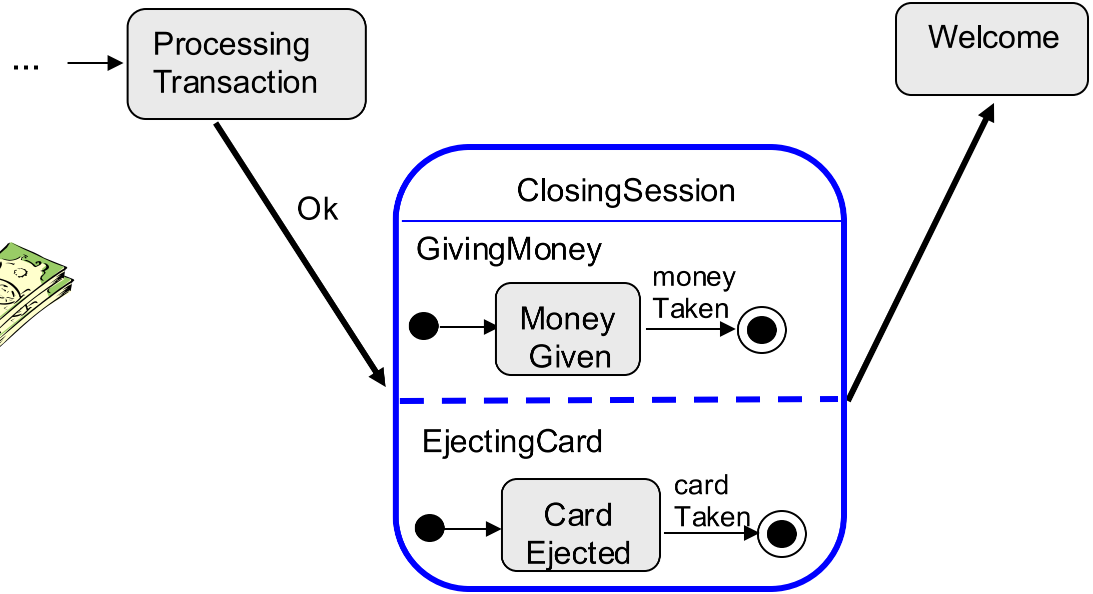
  (doorsClosed, trainStopped); (doorsOpen, trainStopped) >

- Model-checking tools can generate counterexample traces leading to violation of desired property

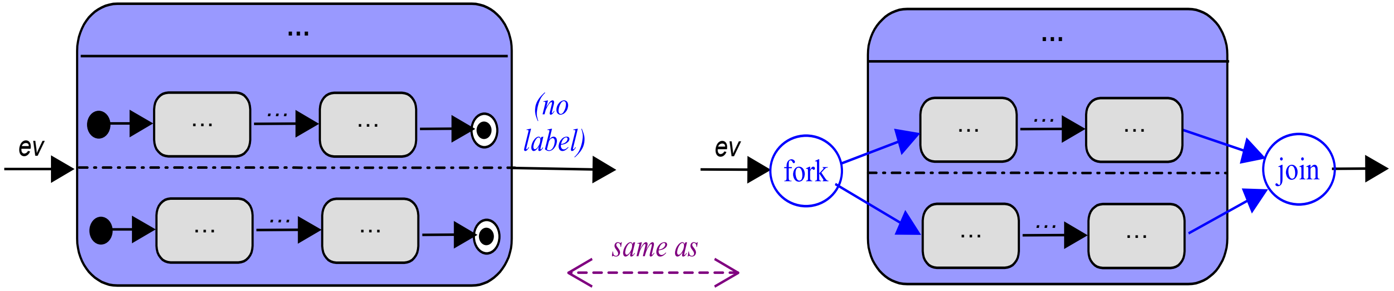
Parallel decomposition: semantic rules

- The object instance is in super-state iff it is in **each of** the nested concurrent sub-states
- Every incoming or outgoing transition of super-state is propagated to each concurrent sub-state:
  - for **incoming transition**: when it fires, an implicit transition to each concurrent substate is simultaneously fired (**FORK** mechanism)
  - for **outgoing transition**:
    - **if no label**: fires when implicit transitions from all concurrent substates were fired (in whatever order, **JOIN** mechanism)
    - **if label**: fires when event in label occurs with guard being true (forcing exit from all concurrent sub-states)

Fork / Join mechanisms: example



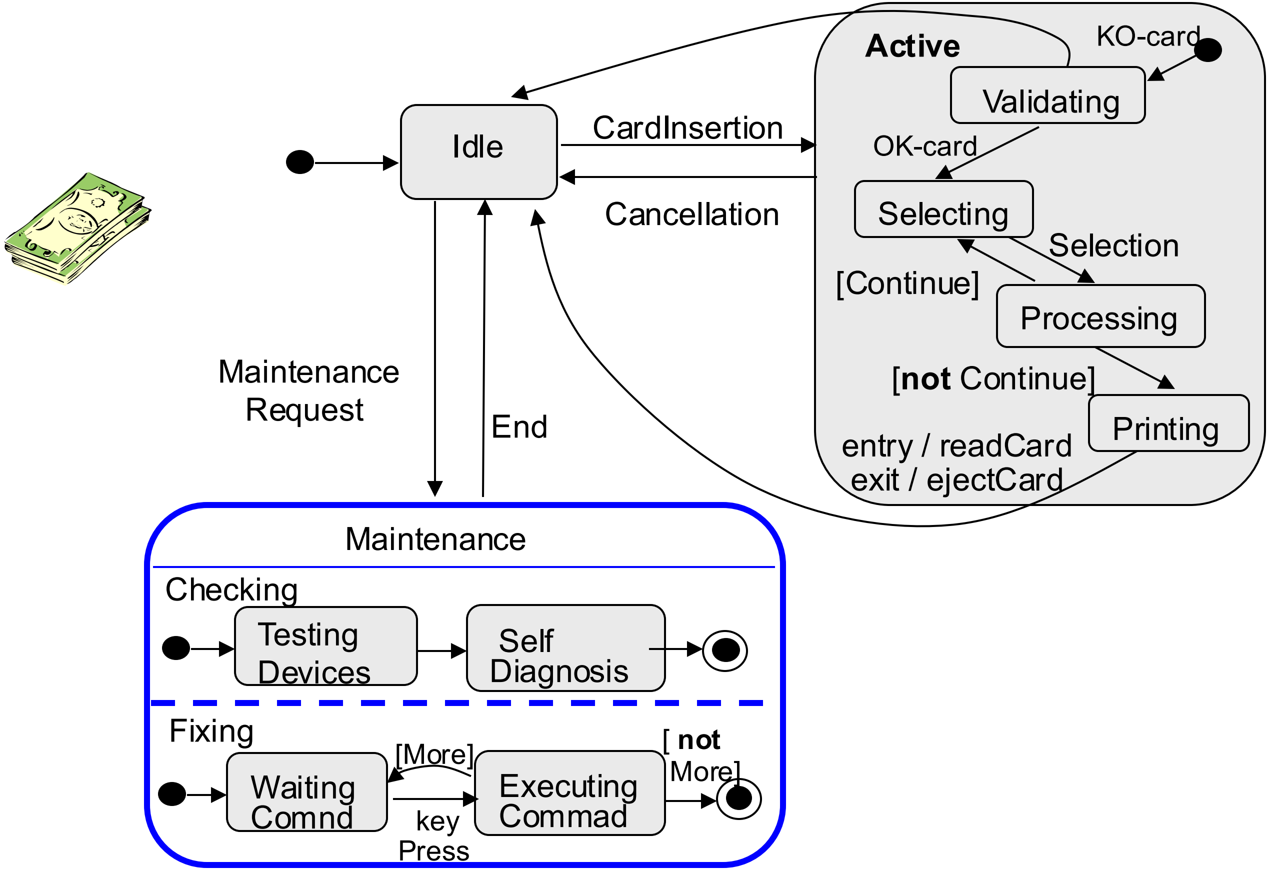
Initial and final substates in concurrent diagrams



Combining parallel & sequential decomposition: guidelines for manageable diagrams

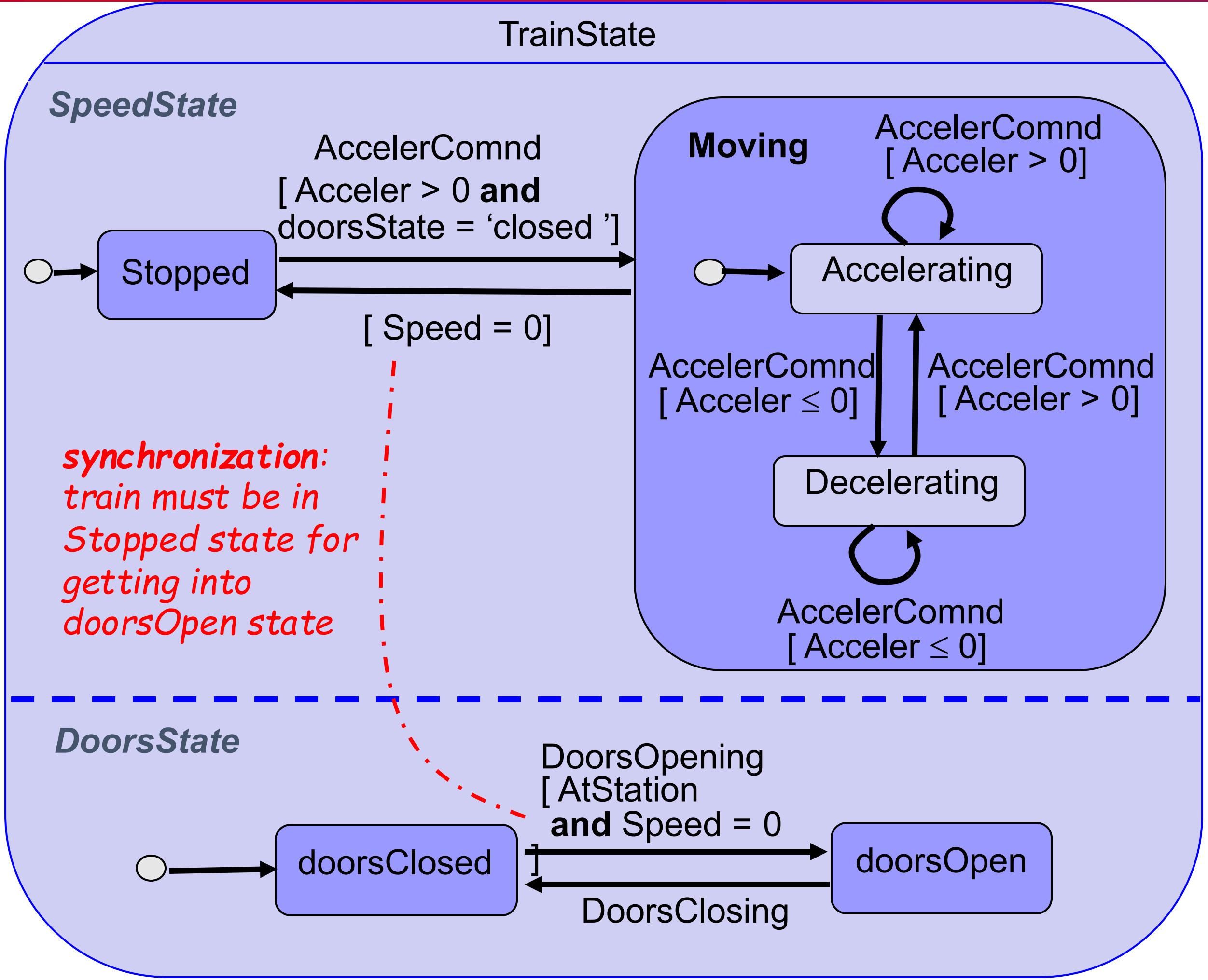
- Introduce sequential sub-states for:
  - refining complex, coarse-grained states
- Introduce concurrent states for:
  - state variables controlled by different agents
  - different state variables controlled by same agent
  - common events triggering multiple independent transitions
- Insert:
  - initial sub-state in each concurrent diagram, with outgoing transition to desired state
  - final sub-state in each concurrent diagram, with incoming transition from desired state
- Avoid “*spaghetti*” diagrams where ...
  - transitions connecting sequential sub-states of a concurrent state to sequential sub-states of other concurrent states (or to outer super-states)

Parallel & sequential decomposition: example



Combining parallel & sequential decomposition: guidelines for manageable diagrams

- When concurrent states are decomposed in sequential sub-states, check if any **synchronization** is required:
  - same event requiring transitions in multiple sub-diagrams
  - event causing transition in consumer diagram to be notified in producer diagram (**send** action)
  - synchronizing guards on same state variable in different sub-diagrams
  - guard in sub-diagram referring to state variable modified by transition in other sub-diagram
- Check lexical consistency of event names in order to avoid:
  - undesired firing by different events with same name
  - non-firing due to same event having different names



TrainState

SpeedState

Stopped

AccelerComnd
[Acceler > 0 **and**
doorsState = 'closed ']

[Speed = 0]

Moving

AccelerComnd
[Acceler > 0]

Accelerating

AccelerComnd
[Acceler ≤ 0]

AccelerComnd
[Acceler > 0]

Decelerating

AccelerComnd
[Acceler ≤ 0]

***synchronization**:*
train must be in
Stopped state for
getting into
doorsOpen state

DoorsState

doorsClosed

DoorsOpening
[AtStation
and Speed = 0
]

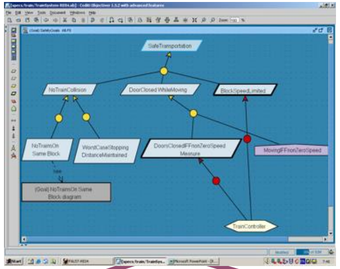
doorsOpen

DoorsClosing

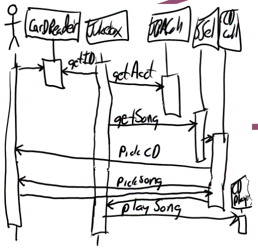
Modeling system behaviors: outline

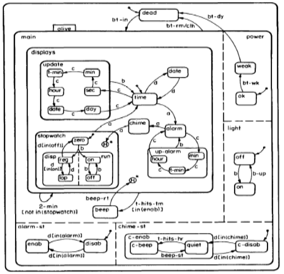
- Modeling instance behaviors
  - Scenarios as UML sequence diagrams
  - Scenario refinement: episodes and agent decomposition
- Modeling class behaviors
  - State machines as UML state diagrams
  - State machine refinement: sequential & concurrent substates
- Building behavior models
  - Elaborating relevant scenarios for good coverage
  - Decorating scenarios with state conditions
  - From scenarios to state machines
  - From scenarios to goals
  - From operationalized goals to state machines

Goals, scenarios, state machines are complementary



- ☺ declarative, satisfaction arguments
- ☺ functional & non-functional, options
- ☺ many behaviors ☹ but implicit
- ☺ early analyses
- ☹ too abstract? hard to elicit?





- ☺ concrete examples, narrative
- ☺ easier to elicit, validate
- ☺ explicit behaviors
- ☺ acceptance test data
- ☹ partial, few behaviors: coverage?
- ☹ implicit reqs, premature choices?

- ☺ visual abstraction
- ☺ explicit behaviors (entire classes)
- ☺ verifiable, executable
- ☺ code generation
- ☹ implicit reqs ... too operational?
- ☹ hard to build & understand

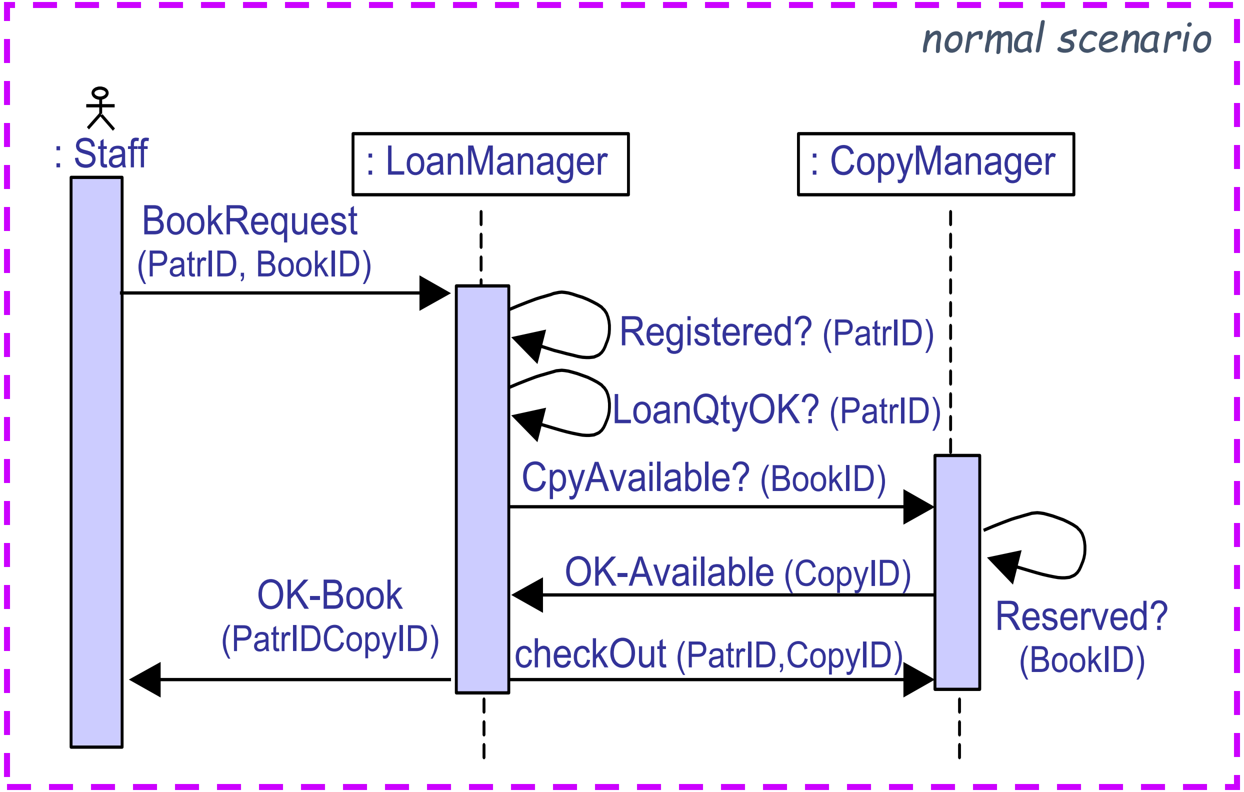
Elaborating relevant scenarios for good coverage

- Work pairwise: one agent pair after the other
- Ensure goal coverage by *positive* scenarios
- Ensure obstacle coverage by *negative* scenarios
- Identify auxiliary episodes
  - required for next interaction
  - info acquisition, agent authentication, help request, ...
- Explore stimulus - response chains
  - if interaction = stimulus sent... what response is required?
- Check scenarios for clean-up ...
  - split scenarios with unrelated concerns
  - remove irrelevant events
  - refine unmonitorable or uncontrollable interaction events
  - for related episodes: ensure common granularity

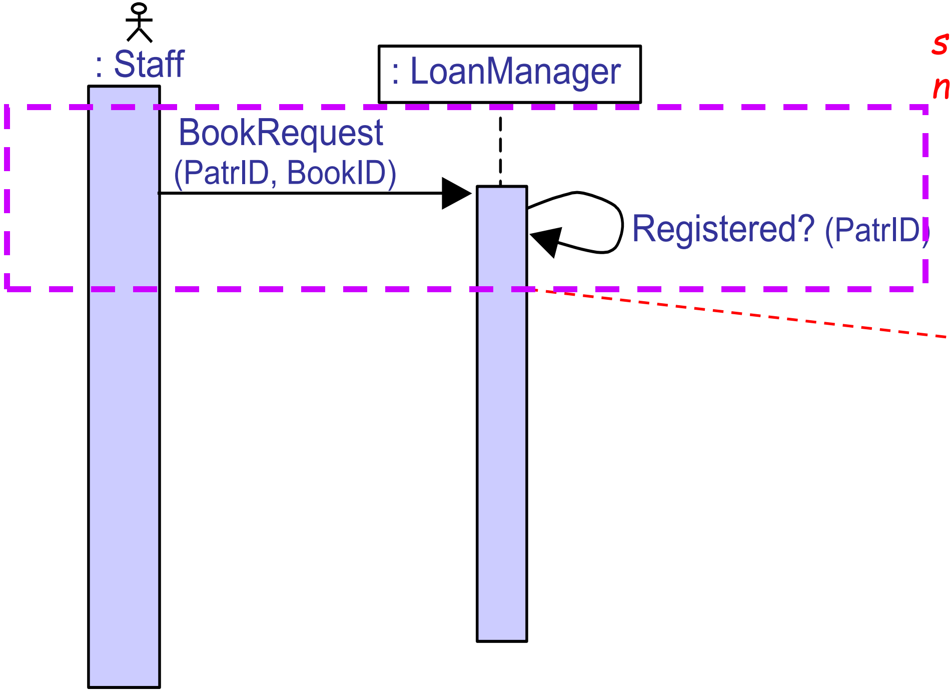
Elaborating relevant scenarios for good coverage

- Look for abnormal scenarios associated with normal ones:
  - "associated" = sharing common prefix episode, then differing by possible exception case
  - look at these **systematically**
    - take all prefix episodes of normal scenario by increasing size
    - for each prefix: possible exceptions at the end ?
      - possible use of exception patterns:
        - invalid data, unsatisfiable request,
        - no response, too late response, inadequate response,
        - cancel events, ...
    - for each exception: what suitable course of action?

Looking for associated *abnormal* scenarios: ex.



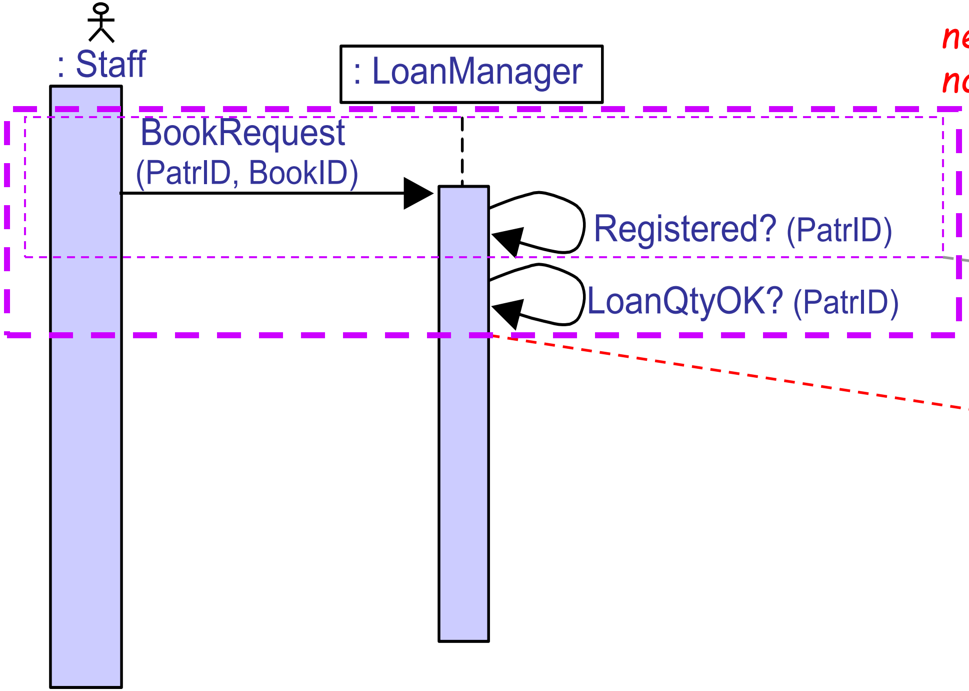
Looking for associated *abnormal* scenarios: ex.



smallest prefix of
normal scenario

exception:
UnRegistered

Looking for associated *abnormal* scenarios: ex.

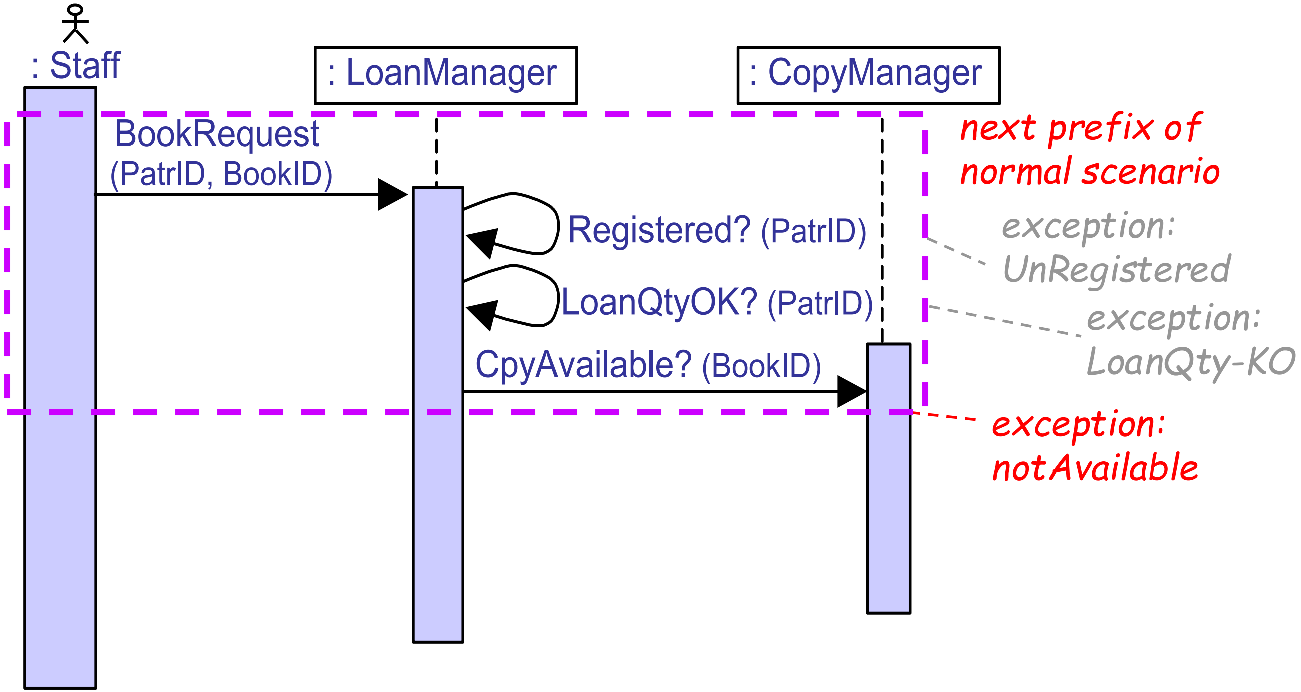


next prefix of normal scenario

exception: UnRegistered

exception: LoanQty-KO

Looking for associated *abnormal* scenarios: ex.



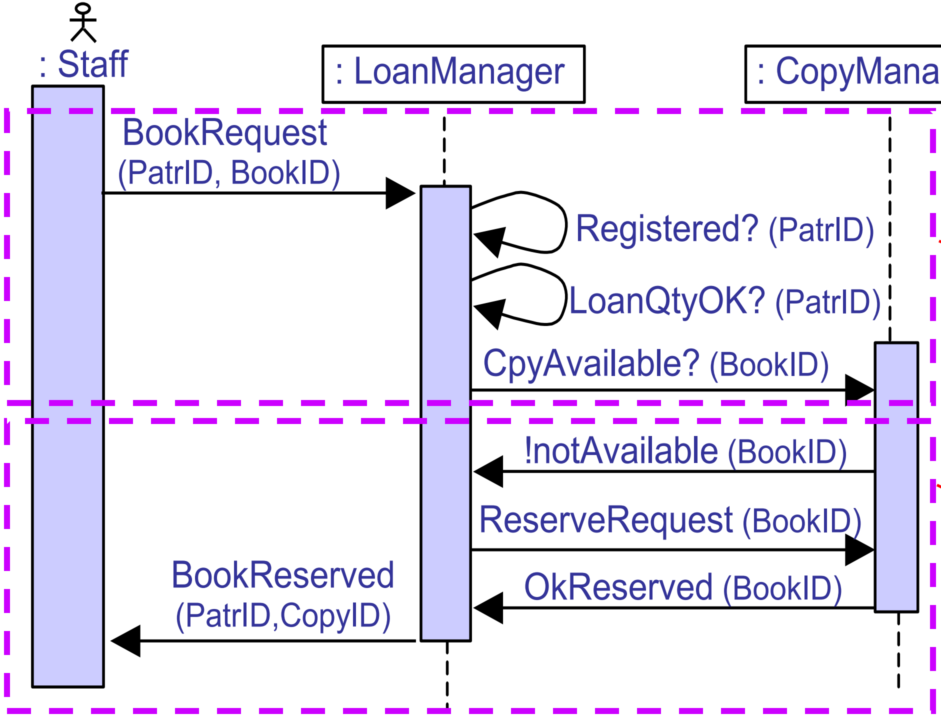
next prefix of normal scenario

exception: UnRegistered

exception: LoanQty-KO

exception: notAvailable

Looking for associated *abnormal* scenarios: ex.



common prefix episode

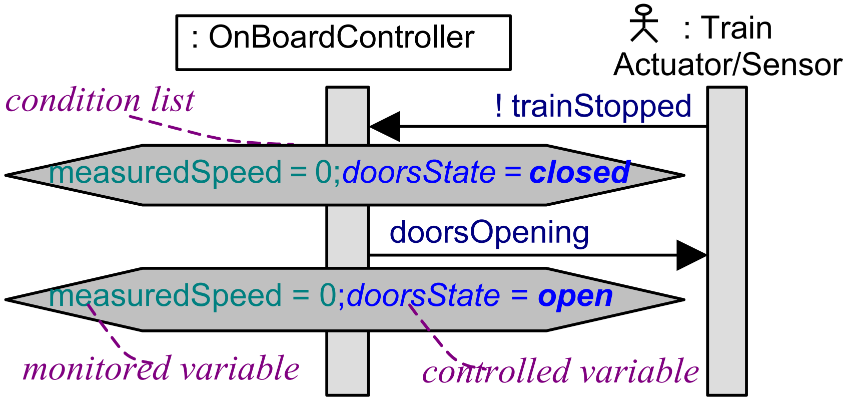
differing postfix from exception point

Modeling system behaviors: outline

- Modeling instance behaviors
  - Scenarios as UML sequence diagrams
  - Scenario refinement: episodes and agent decomposition
- Modeling class behaviors
  - State machines as UML state diagrams
  - State machine refinement: sequential & concurrent substates
- Building behavior models
  - Elaborating relevant scenarios for good coverage
  - Decorating scenarios with state conditions
  - From scenarios to state machines
  - From scenarios to goals
  - From operationalized goals to state machines

Decorating scenarios with state conditions

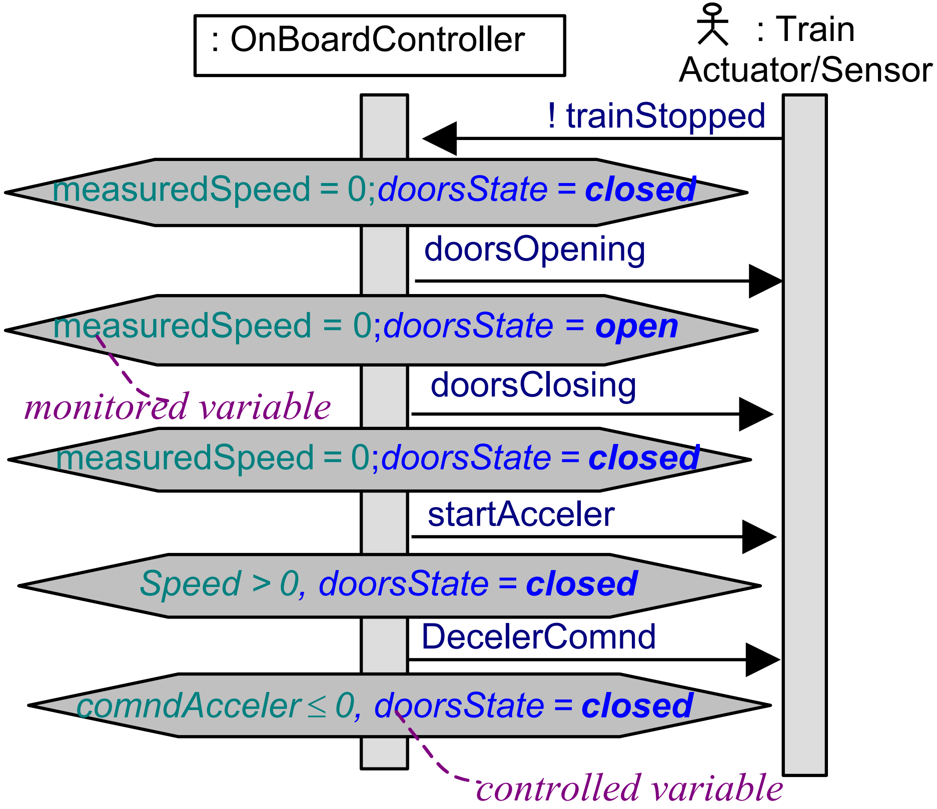
- Helpful for state-based reasoning from scenarios
- **State condition** at timepoint on agent lifeline:
  - captures snapshot state of dynamic variables at this point
- Structured as **condition list**: implicitly conjoined:
  - monitored conditions: state of variables monitored by the agent
  - controlled conditions: state of variables controlled by the agent



Decorating scenarios with state conditions

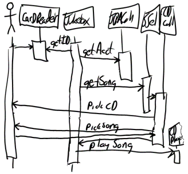
- Condition lists are computed by down propagation along lifeline
- From *DomPre*, *DomPost* of operations corresponding to interaction events (available from operation model)
- For **outgoing** event:
  - add its *DomPre* to list of *controlled* conditions before it
  - add its *DomPost* to list of *controlled* conditions after it
  - remove any invalidated condition
- For **incoming** event:
  - add its *DomPost* to list of *monitored* conditions after it
  - remove any invalidated condition

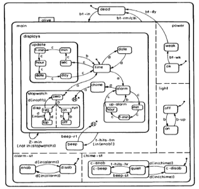
Propagating condition lists: example


: OnBoardController
: Train
Actuator/Sensor
! trainStopped
measuredSpeed = 0;doorsState = closed
doorsOpening
measuredSpeed = 0;doorsState = open
monitored variable
doorsClosing
measuredSpeed = 0;doorsState = closed
startAcceler
Speed > 0, doorsState = closed
DecelerComnd
comndAcceler ≤ 0, doorsState = closed
controlled variable

From scenarios to state machines

- State machines can be built incrementally from scenarios:
  - so as to cover all behaviors captured by positive scenarios
  - while excluding all behaviors captured by negative scenarios
- For building state diagrams from sequence diagrams, 3 steps:
  - Decorate scenarios with state conditions (as just seen)
  - Generalize scenarios into state machines
  - Check, extend & restucture resulting SMs

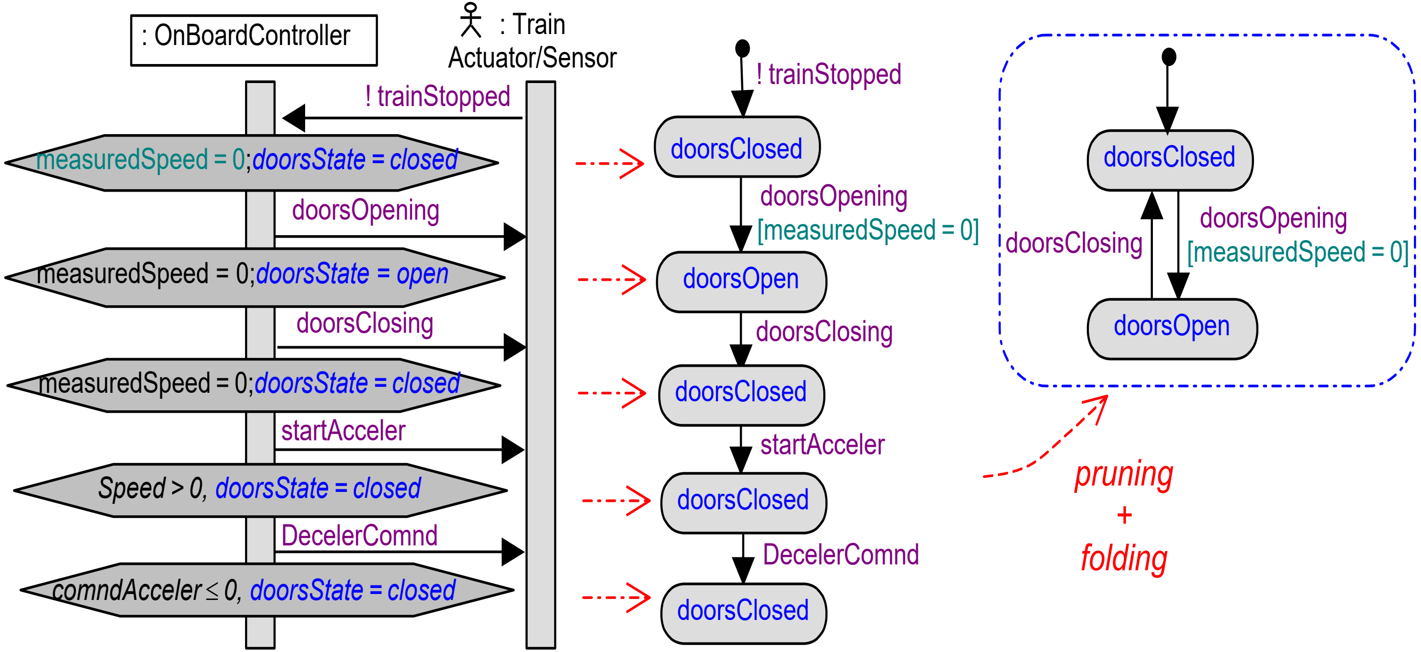




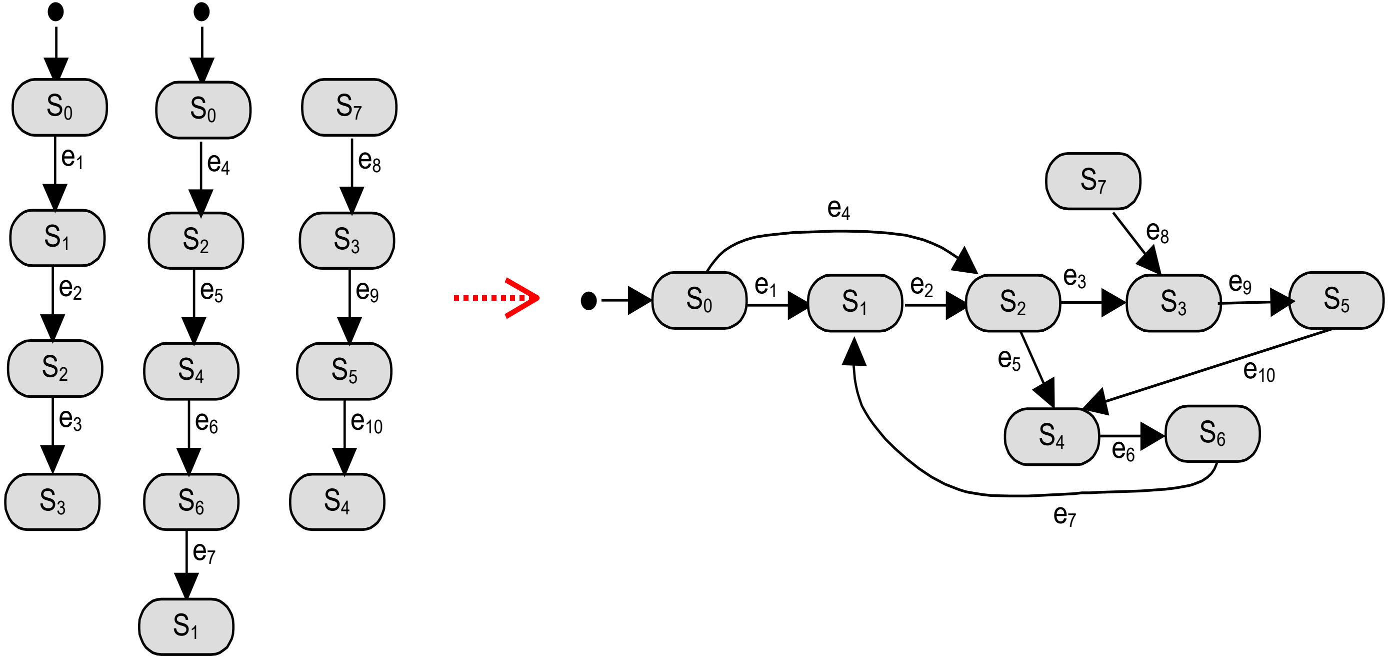
Generalize scenarios into state machines

- One concurrent SM per variable controlled by the agent, whose paths cover all corresponding scenario lifelines
  - **Lifeline selection**: all lifelines referring to this controlled variable
  - SM path derivation:
    - sequence of **states** =   sequence of lifeline state conditions on this controlled variable
    - **transitions** labelled with corresponding interaction event
    - add initial state, conditions on monitored variables as guards
    - remove transitions with no state change for this controlled variable
    - merge multiple occurrences of same state by folding   => cycles
  - **SM path merge**:
    - take a path with initial state as first path
    - merge new path from its start: for each next state:
      - if already there: add incoming transition
      - if not already there: add it + incoming transition

SM path derivation

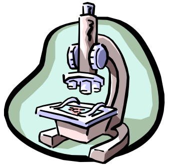


SM path merge



Check resulting concurrent SM

- **Within** concurrent state, for one controlled variable:
  - Unreachable states ? (from initial state)
  - Missing states ? (incl. final state)
  - Missing or inadequate transitions ? (events, guards)
  - Missing actions ?
- **Between** concurrent states, for different controlled variables
  - Synchronization needed? (as seen before)
    - Shared events? Synchronizing guards? Event notification ?
  - Lexical consistency of event names? (as seen before)



Modeling system behaviors: outline

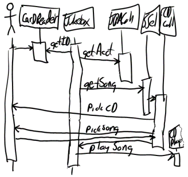
- Modeling instance behaviors
  - Scenarios as UML sequence diagrams
  - Scenario refinement: episodes and agent decomposition
- Modeling class behaviors
  - State machines as UML state diagrams
  - State machine refinement: sequential & concurrent substates
- Building behavior models
  - Elaborating relevant scenarios for good coverage
  - Decorating scenarios with state conditions
  - From scenarios to state machines
  - From scenarios to goals
  - From operationalized goals to state machines

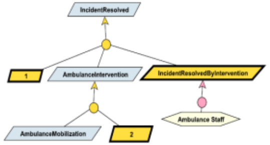
From scenarios to goals

- Goals can be identified & specified from scenarios ...
  - generalize positive scenarios by covering more behaviors
  - while excluding all behaviors captured by negative scenarios
- By asking **WHY?** questions about positive scenarios,

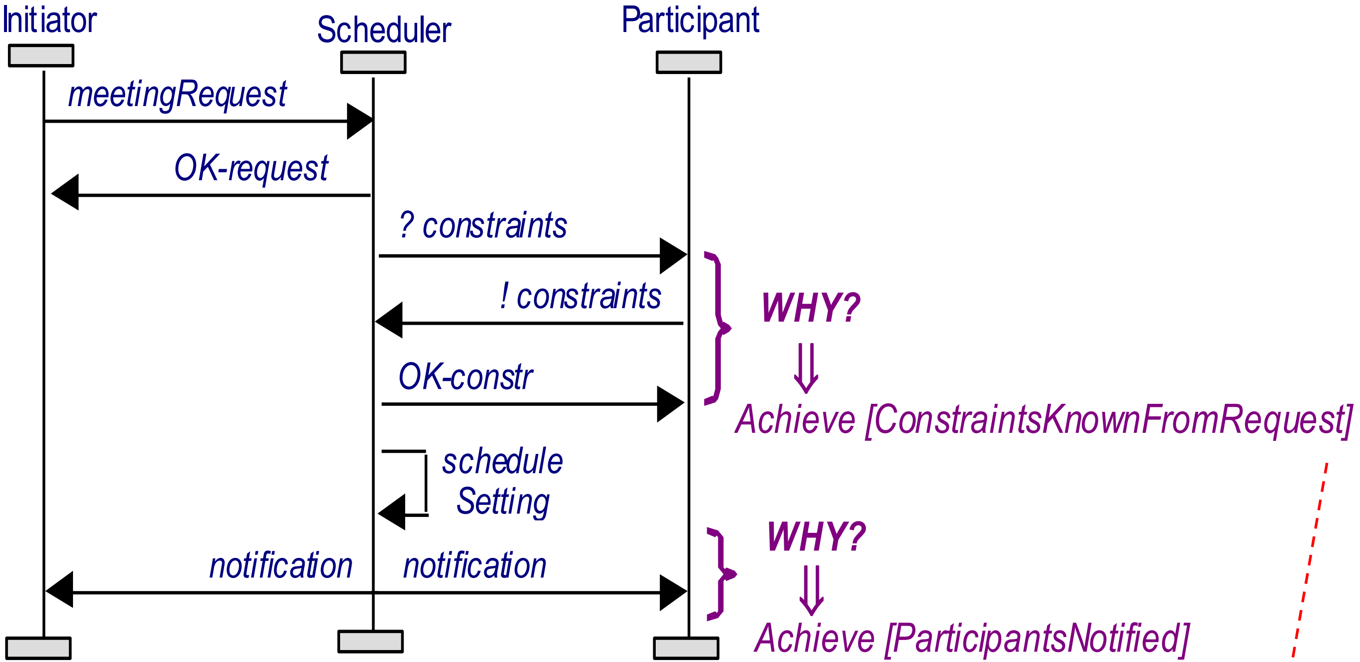
  WHY NOT? questions about negative scenarios
  - Scenario decomposed in episodes

    => milestone refinement of scenario goal into episode subgoals
- By mining behavioral goals from decorated scenarios



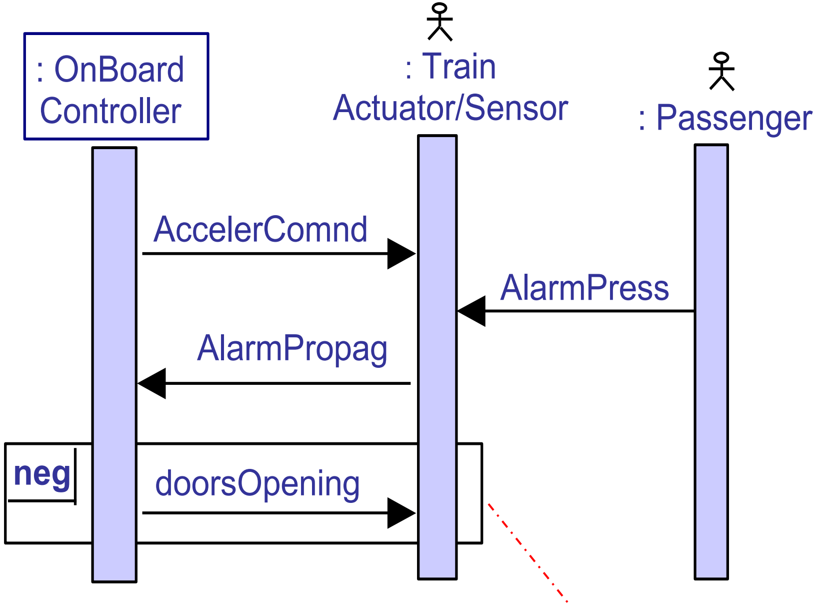


Identifying goals from scenario episodes: WHY questions and milestones



Milestone subgoals of
Achieve [MaximumAttendance]

Identifying goals from scenario episodes: WHY NOT questions



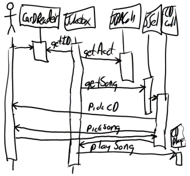
WHY NOT?

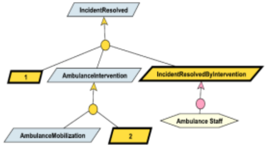
⇓

Maintain [DoorsClosedWhileMoving]

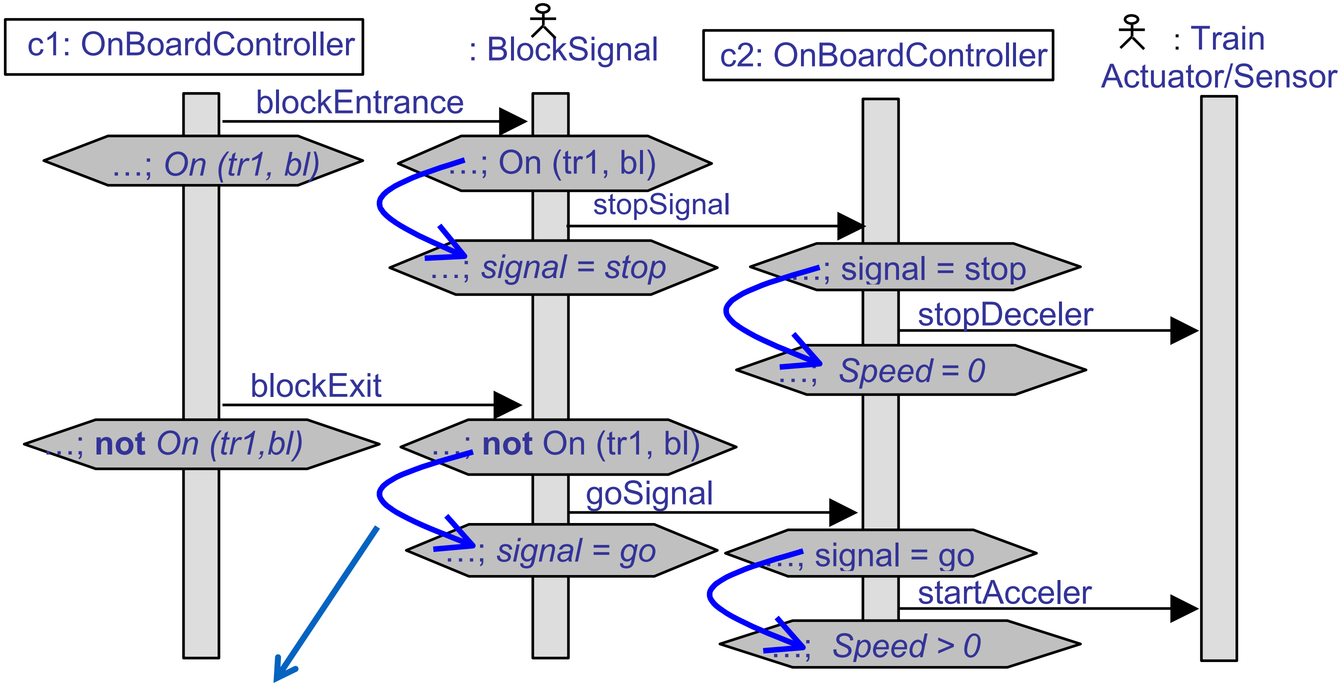
Mining behavioral goals from decorated scenarios

- Based on **state conditions** along timelines of decorated scenario
- For Achieve goals:
  - consider **stimulus-response** interaction patterns
  - if Condition*Before*Interaction **then sooner-or-later** ConditionAfter
- For Maintain goals:
  - consider **invariance** patterns from state transitions *ST*
  - **if** *ST* **then always** *Invariant*Condition unless *New*Condition
- These are leaf goals under responsibility of the agent associated with the timeline



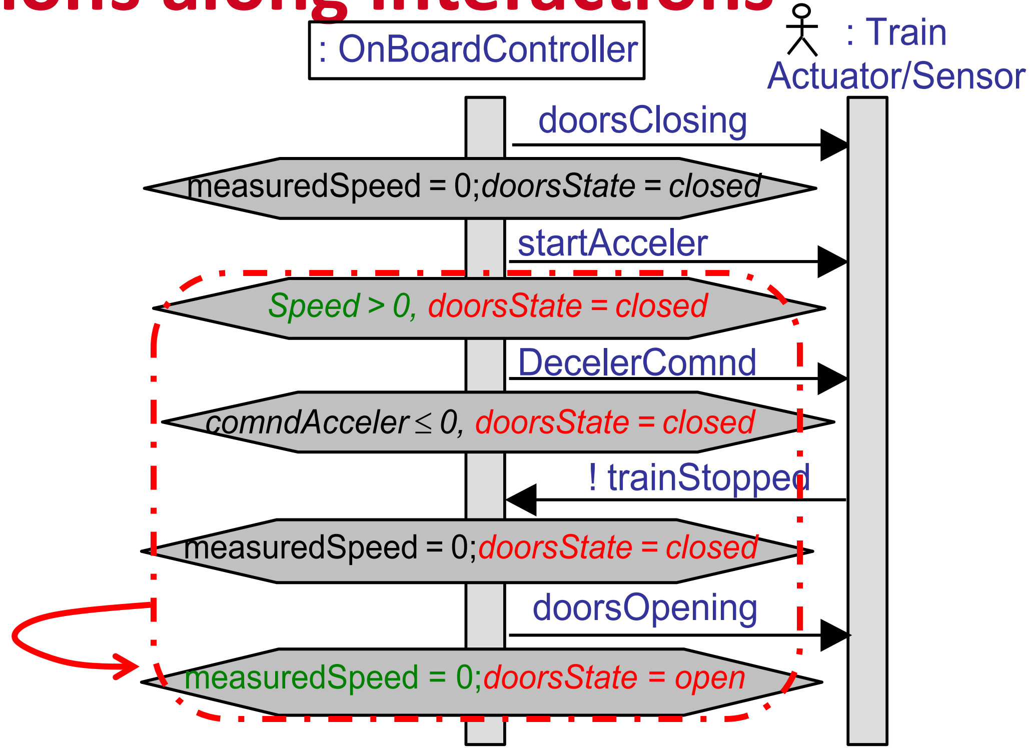


Mining *Achieve* goals from stimulus-response interactions



Achieve [SignalSetToGo **If** NoTrainOnBlock]

Mining *Maintain* goals from invariant conditions along interactions



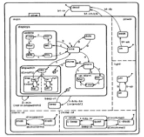
: OnBoardController
: Train
Actuator/Sensor
doorsClosing
measuredSpeed = 0;*doorsState = closed*
startAcceler
Speed > 0, doorsState = closed
DecelerComnd
comndAcceler ≤ 0, doorsState = closed
! trainStopped
measuredSpeed = 0;*doorsState = closed*
doorsOpening
measuredSpeed = 0;*doorsState = open*

If Speed > 0 **and** doorsState = closed **then**
always doorsState = closed **unless** measuredSpeed = 0

Modeling system behaviors: outline

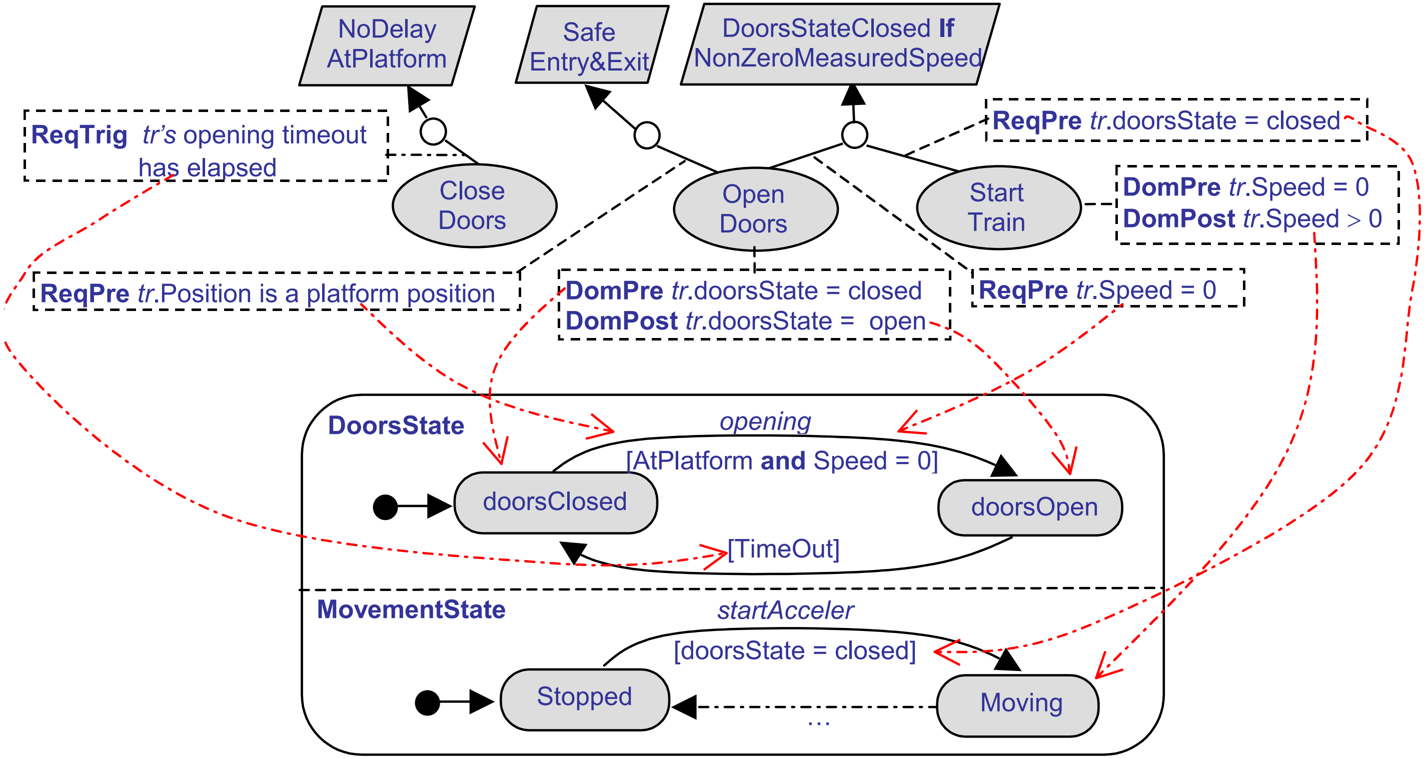
- Modeling instance behaviors
  - Scenarios as UML sequence diagrams
  - Scenario refinement: episodes and agent decomposition
- Modeling class behaviors
  - State machines as UML state diagrams
  - State machine refinement: sequential & concurrent substates
- Building behavior models
  - Elaborating relevant scenarios for good coverage
  - Decorating scenarios with state conditions
  - From scenarios to state machines
  - From scenarios to goals
  - From operationalized goals to state machines

Deriving state diagrams from operationalized goals



- One concurrent SM per agent; one concurrent sub-SM per variable controlled by the agent
- For each dynamically relevant state variable:
  - **Operationalization selection**: take all leaf goals constraining it + associated operations including it in their **Output** list
  - **SM derivation**: for each selected operation Op, derive a transition
    - (DomPre, DomPost) $\Rightarrow$ (sourceState, TargetState)
      with transition label *Op*
    - Conjunction of all ReqPre $\Rightarrow$ transition guard
    - Disjunction of all ReqTrig $\Rightarrow$ guard on event-free transition
  - Add label-free transition from initial state (based on variable's **Init**)
  - Check, extend, restructure this SM as needed
- Can be used the other way round to find missing ReqPre, ReqTrig, Op

Deriving state diagrams from operationalized goals: example


NoDelay AtPlatform
Safe Entry&Exit
DoorsStateClosed **If** NonZeroMeasuredSpeed
ReqTrig *tr's* opening timeout has elapsed
ReqPre *tr*.doorsState = closed
Close Doors
Open Doors
Start Train
DomPre *tr*.Speed = 0
DomPost *tr*.Speed > 0
ReqPre *tr*.Position is a platform position
DomPre *tr*.doorsState = closed
DomPost *tr*.doorsState = open
ReqPre *tr*.Speed = 0
DoorsState
opening
[AtPlatform **and** Speed = 0]
doorsClosed
doorsOpen
[TimeOut]
MovementState
startAcceler
[doorsState = closed]
Stopped
Moving
…