Outline

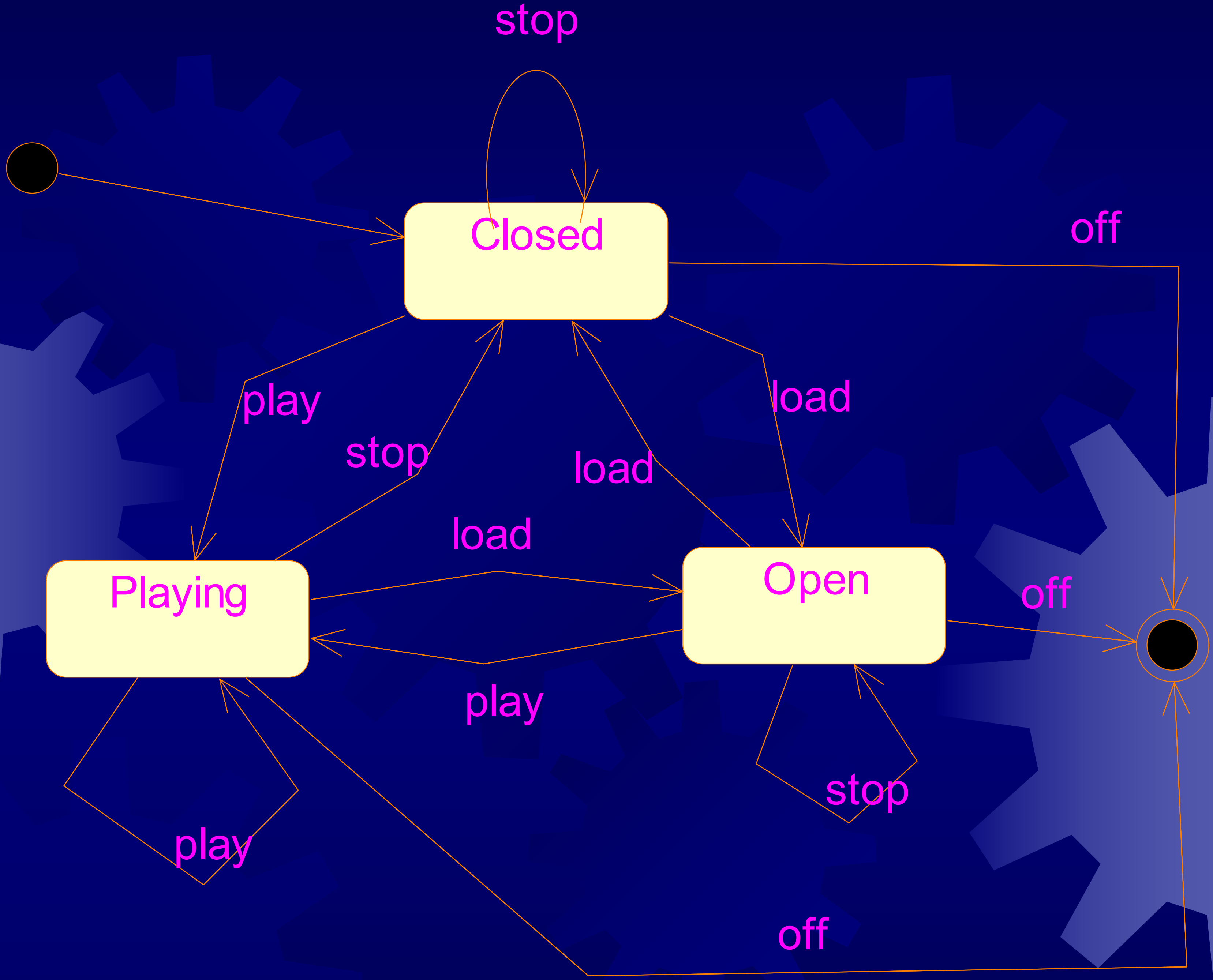
- Why statecharts
- State,transition, Initial and final states
- Actions, guard conditions
- Entry and exit actions
- Activities
- Composite states
- History states
- A “real” example
- Quiz on the automatic ticket machine
- Implementation of statecharts

Why Statecharts

- Interaction diagram Vs. statecharts
  - The former: a short period, a single user generated transaction, a certain sequence of messages
  - The later: an object's whole life, all possible messages it can accept.
- How to construct states
  An object in a particular state will respond differently to at least one event from the way in which it responds to that event in other states.
- (Active) state, event (triggers) and transitions

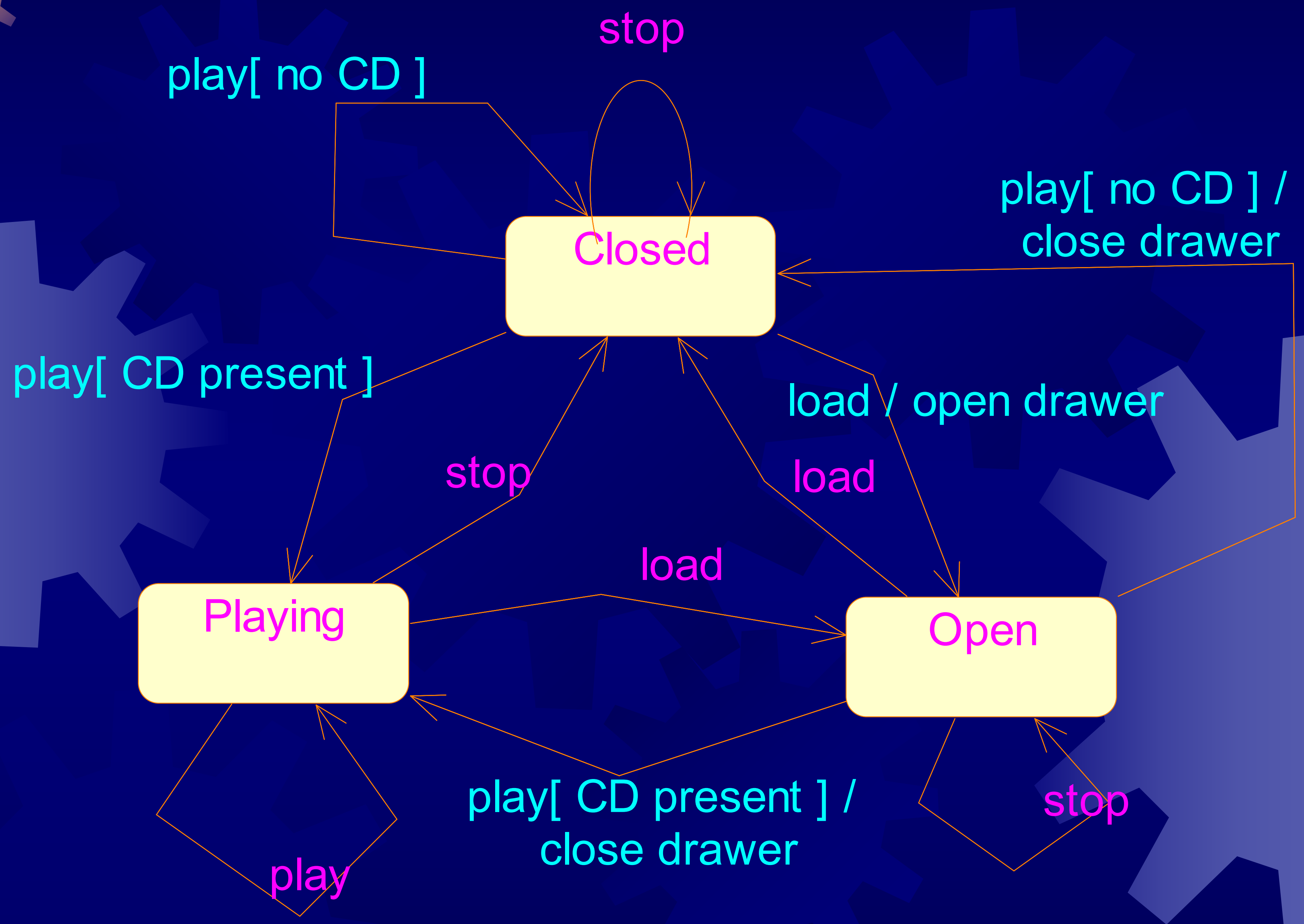
Components of a statechart

- State
- Event
- Transition
- Initial and final states
- Actions



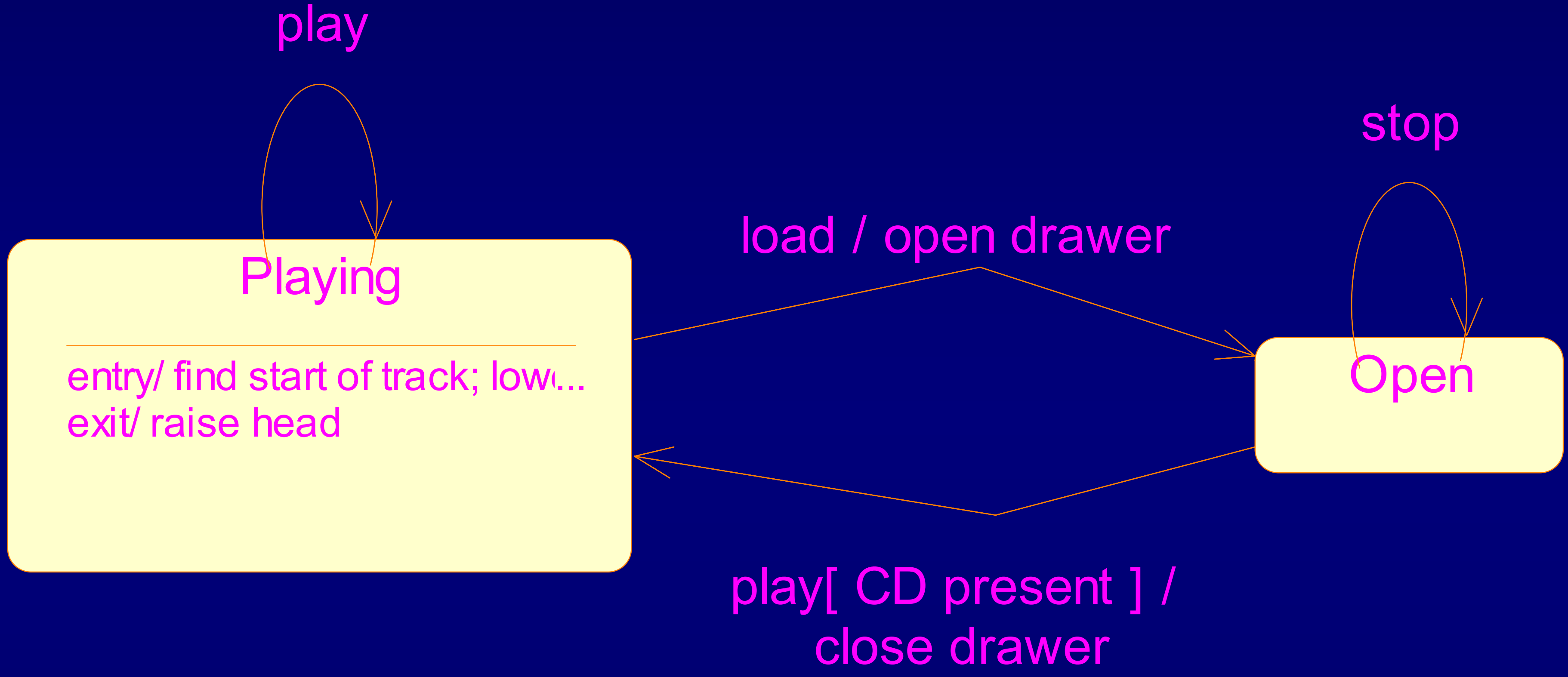
The CD player example

Guard condition and actions



Entry and exit actions

- Entry actions are performed every time a state becomes active, immediately after actions on transitions leading to the state have completed.
- Exit actions: are performed whenever a transition is fired to leave the state.

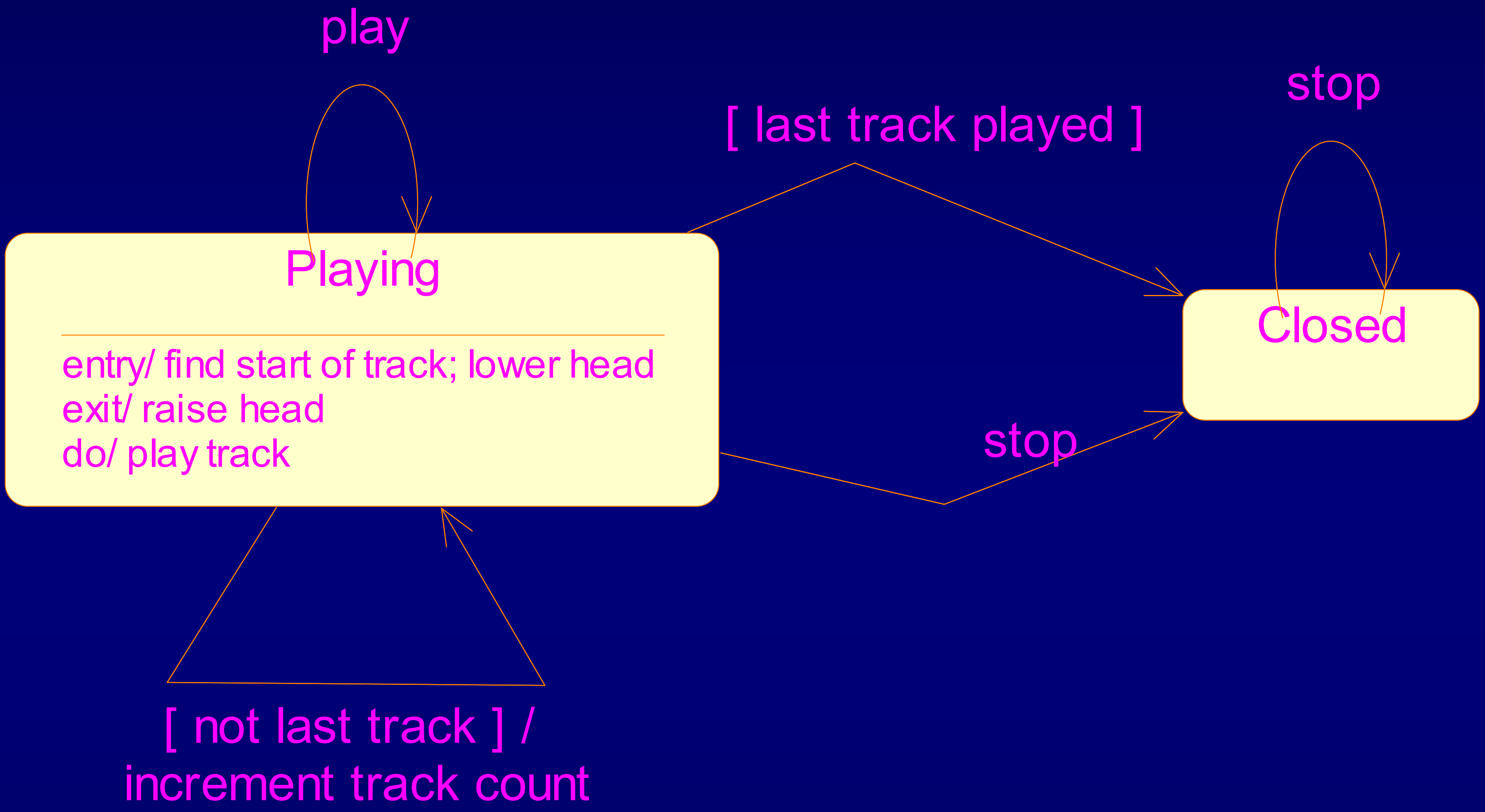

play
Playing
entry/ find start of track; low...
exit/ raise head
load / open drawer
stop
Open
play[CD present] / close drawer

Entry and exit actions

Applicable for self-transitions too.

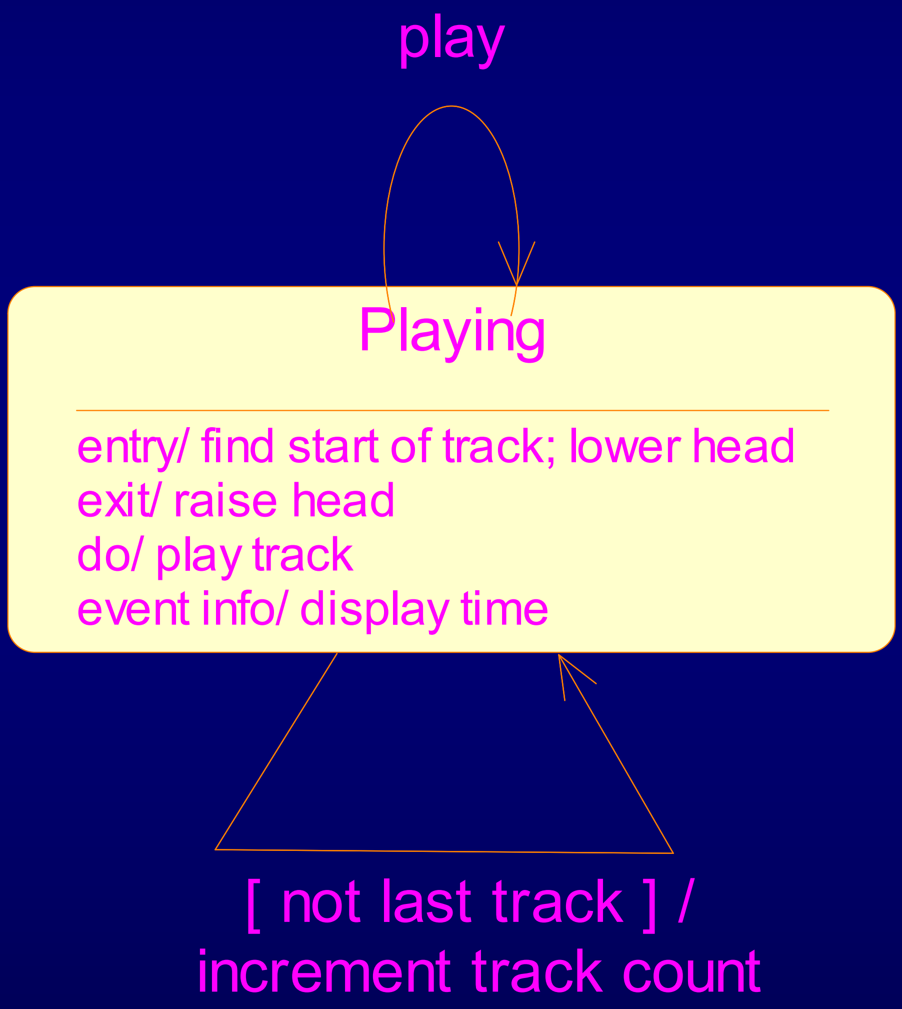
Activities

- Activities
  - The operation continues to run throughout the period when the state is active.
  - Can be interrupted by other events.
  - If no event interrupt it, completion transitions will be issued
- Actions
  - Can be though of as being instantaneous
  - Can not be interrupted by other events.



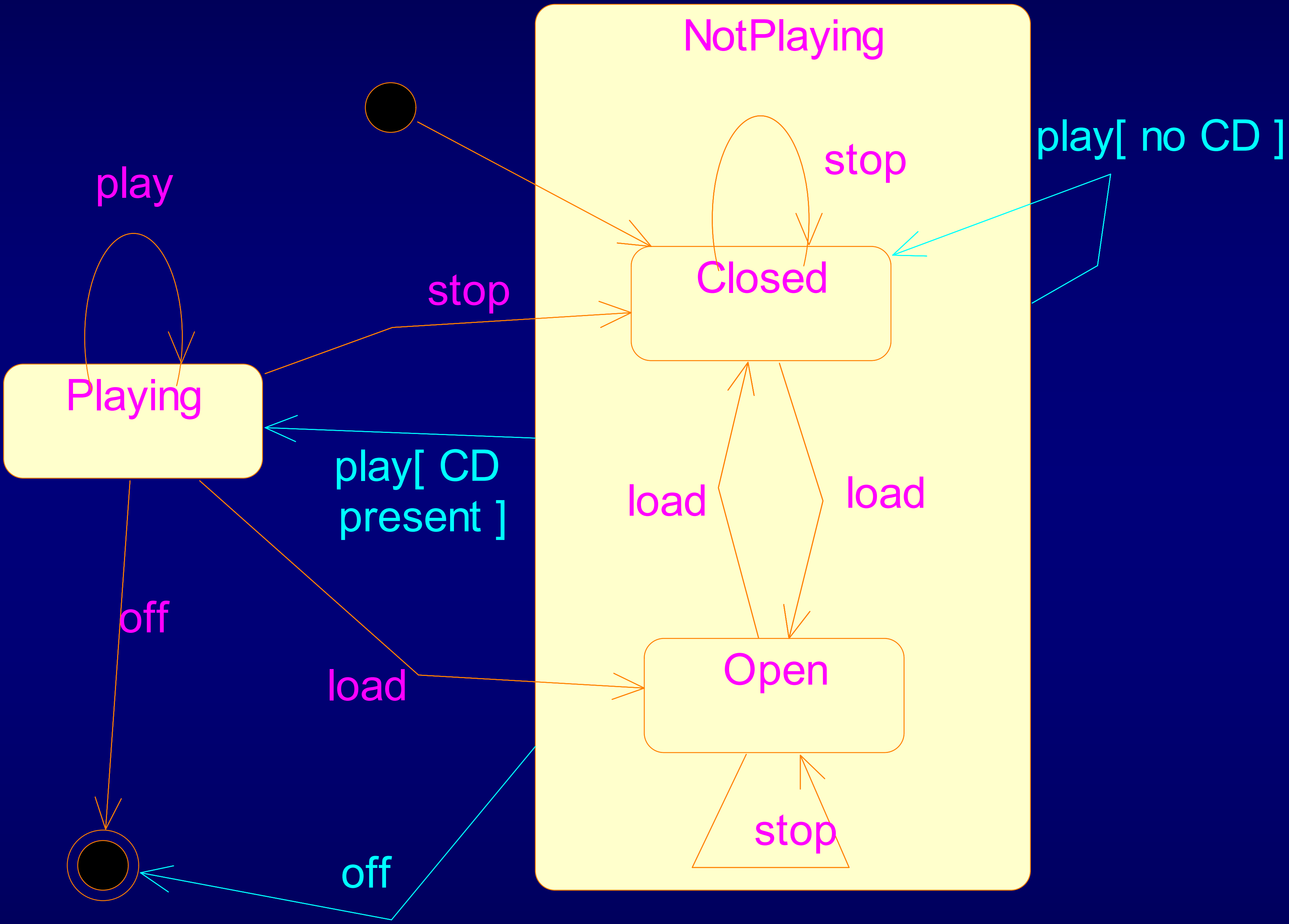
Activities and completion transitions

Internal transition: Some events cause transitions to the current state, but without triggering the execution of the entry and exit actions.



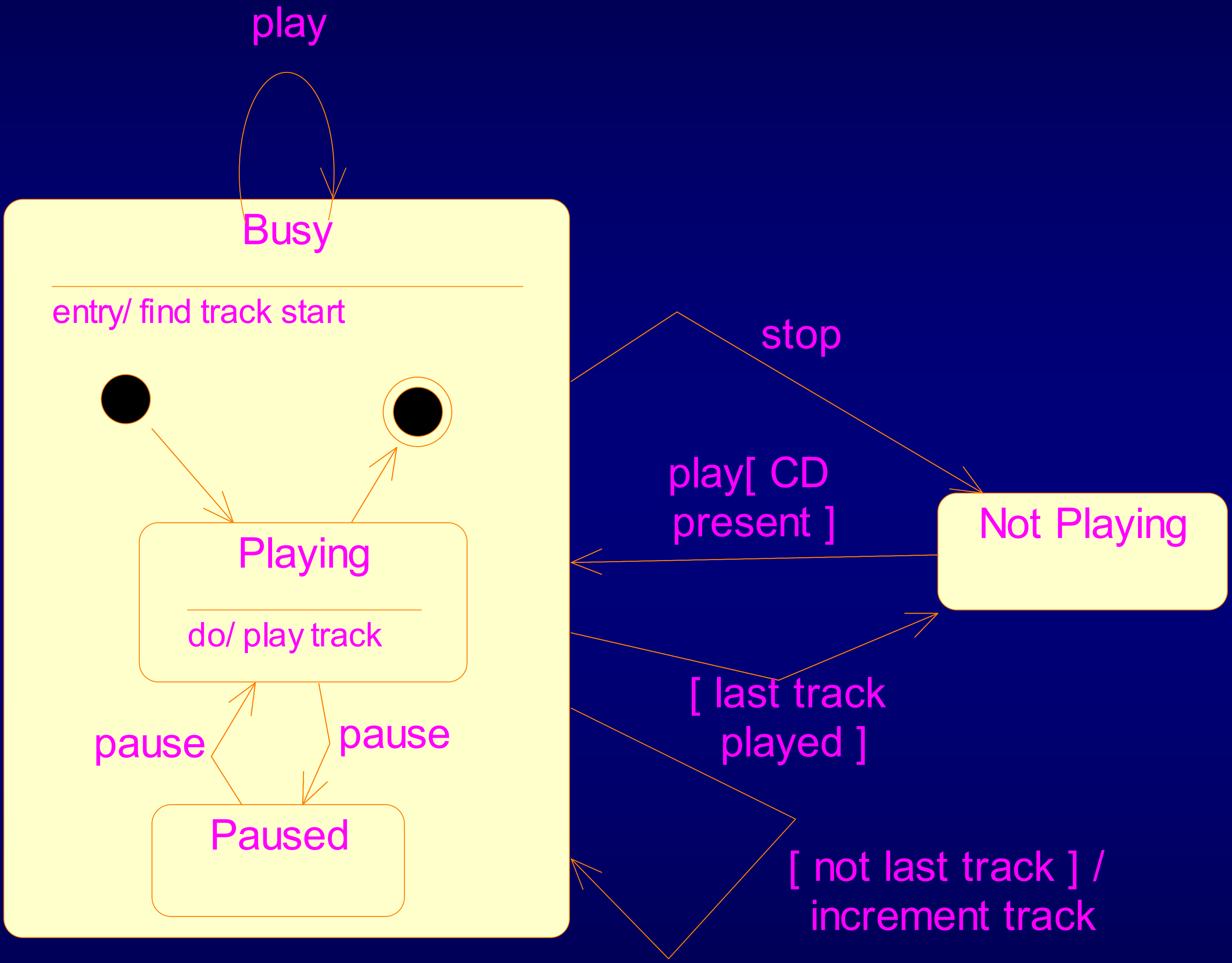
Composite states

- For complex systems, statecharts are often too complex for producing and understanding.
- Composite states are used just for simplifying the statecharts.
- Applicability: for states which have the common transitions (outgoing or incoming)
- The states above are grouped together to form a composite state; the individual states are called substates.


NotPlaying
play[no CD]
stop
play
Closed
stop
Playing
play[CD present]
load
load
off
load
Open
stop
off

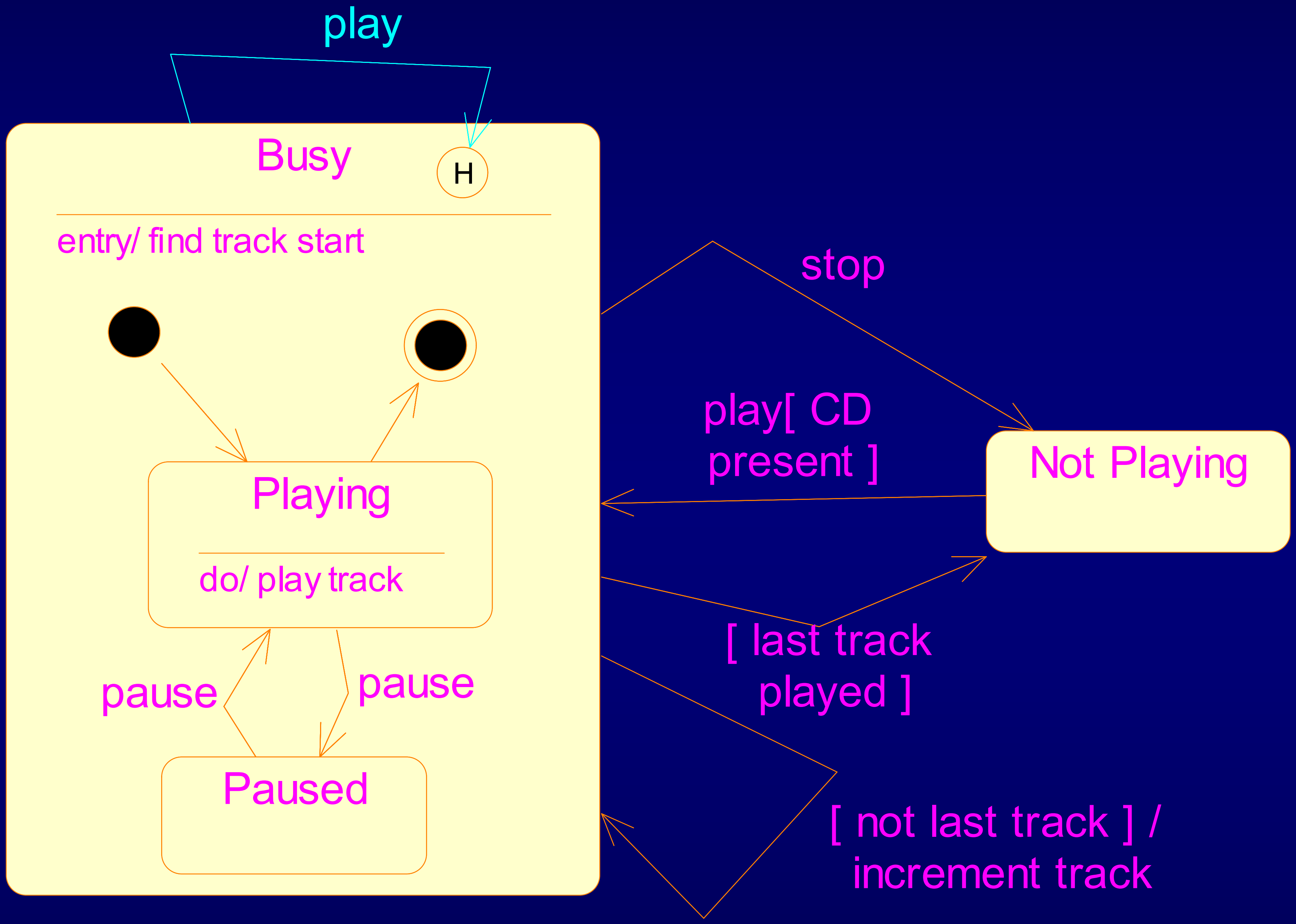
Properties of composite states

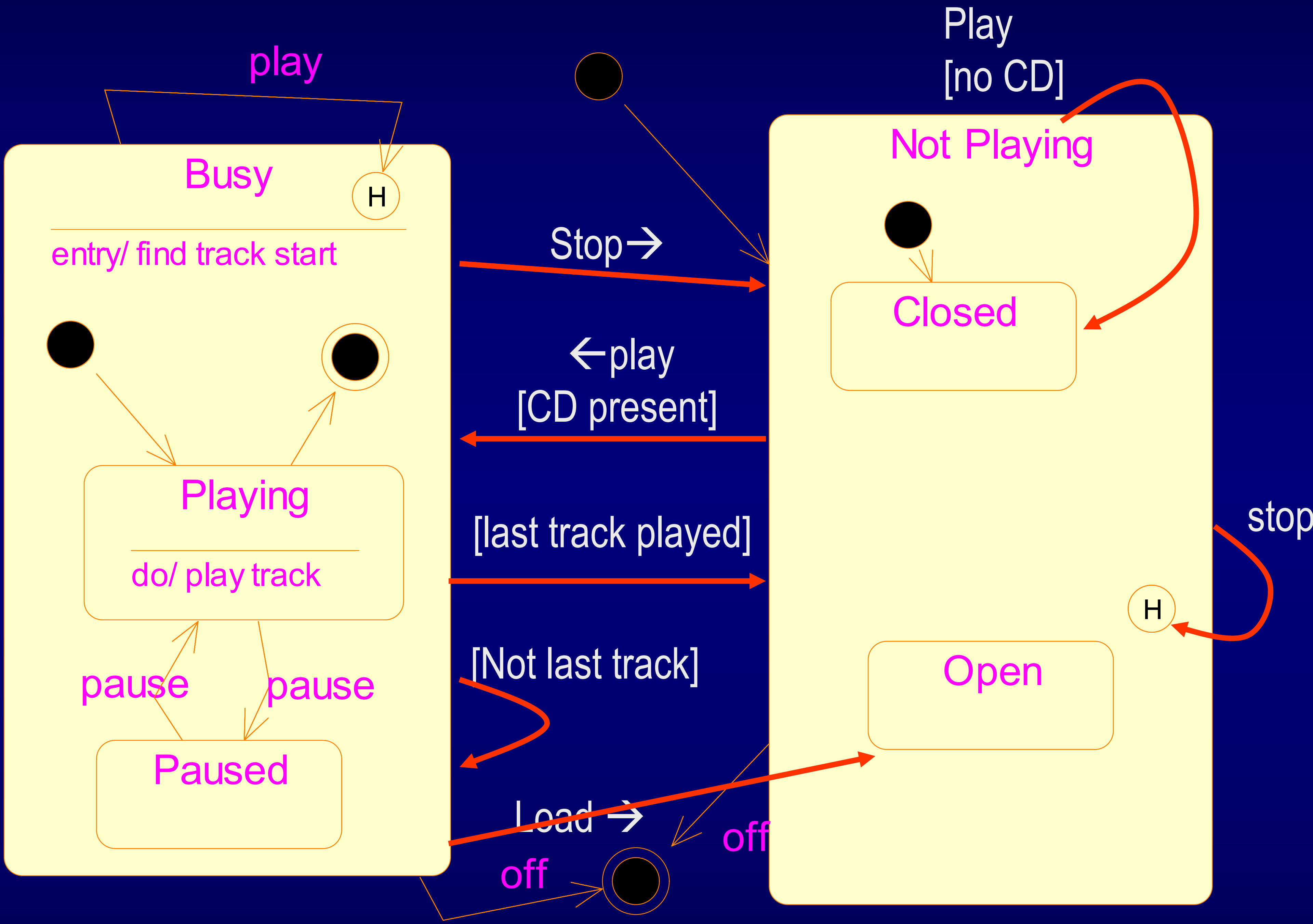
- If a composite state is active, exactly one of its substate must also be active
- Outgoing transition: can flow from the composite state, or from a substate
- Incoming transition: can flow to the composite state, or to a substate
- Initial state: becomes active when an incoming transition reaches at the boundary
- Final state: when ongoing activity has finished; issue completion transitions
- Entry/exit actions


play
Busy
entry/ find track start
Playing
do/ play track
pause
pause
Paused
stop
play[CD present]
Not Playing
[last track played]
[not last track] / increment track

History states

- Requirement: if the “play” button is pressed, return to the CD player’s previous state, either “playing” or “paused”
- The composite state can remember a history state, denoted by an “H” state.


play
Busy
H
entry/ find track start
Playing
do/ play track
pause
pause
Paused
stop
play[CD present]
Not Playing
[last track played]
[not last track] / increment track

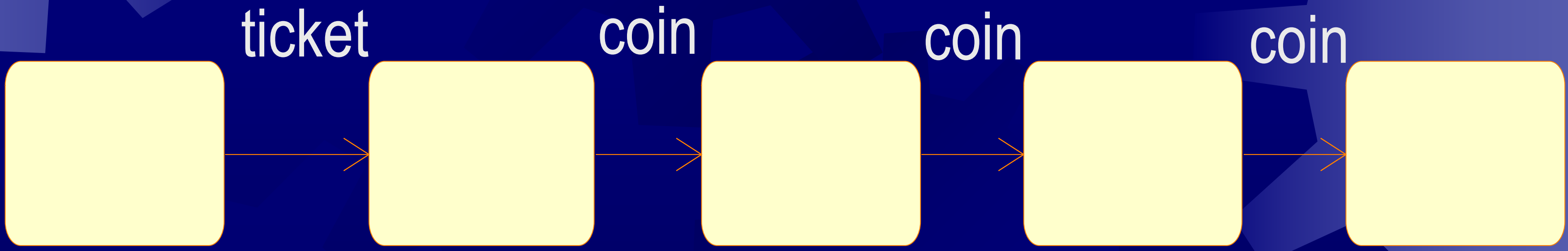

play
Busy
H
entry/ find track start
Playing
do/ play track
pause
pause
Paused
Stop→
←play
[CD present]
[last track played]
[Not last track]
Load →
off
off
Play
[no CD]
Not Playing
Closed
stop
H
Open

A “real” example

- Description of the automatic ticket machine
  - You select a type of ticket
  - The machine displays how much money you should continue to pay.
  - You insert coins
  - Two modes: “Change available” or “Exact money required”
  - You have the option of entering money before selecting a ticket type
  - Cancel: press the “cancel” button or keep silence for 30 seconds

Preliminary statechart

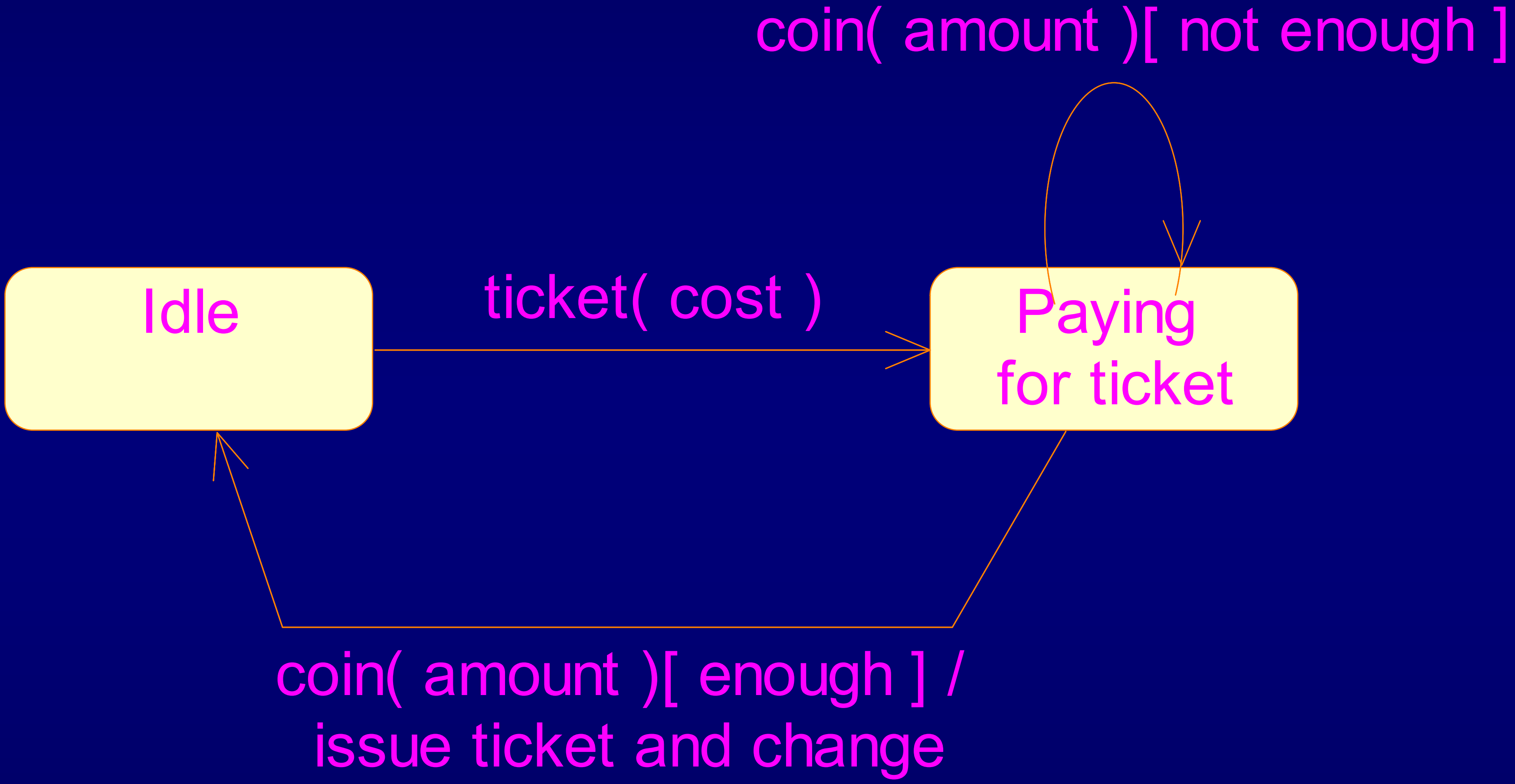
- A single (the most common) transaction
- Every event leads to a transition between two states
- It is unnecessary to name the states



Analysis of the preliminary statecharts

- The process can be divided into three stages:
  1. The machine waits for the user to select a ticket type;
  2. The user inserts coins into the machine;
  3. When the inserted coins is enough, the machine issues the ticket. Return to stage 1.

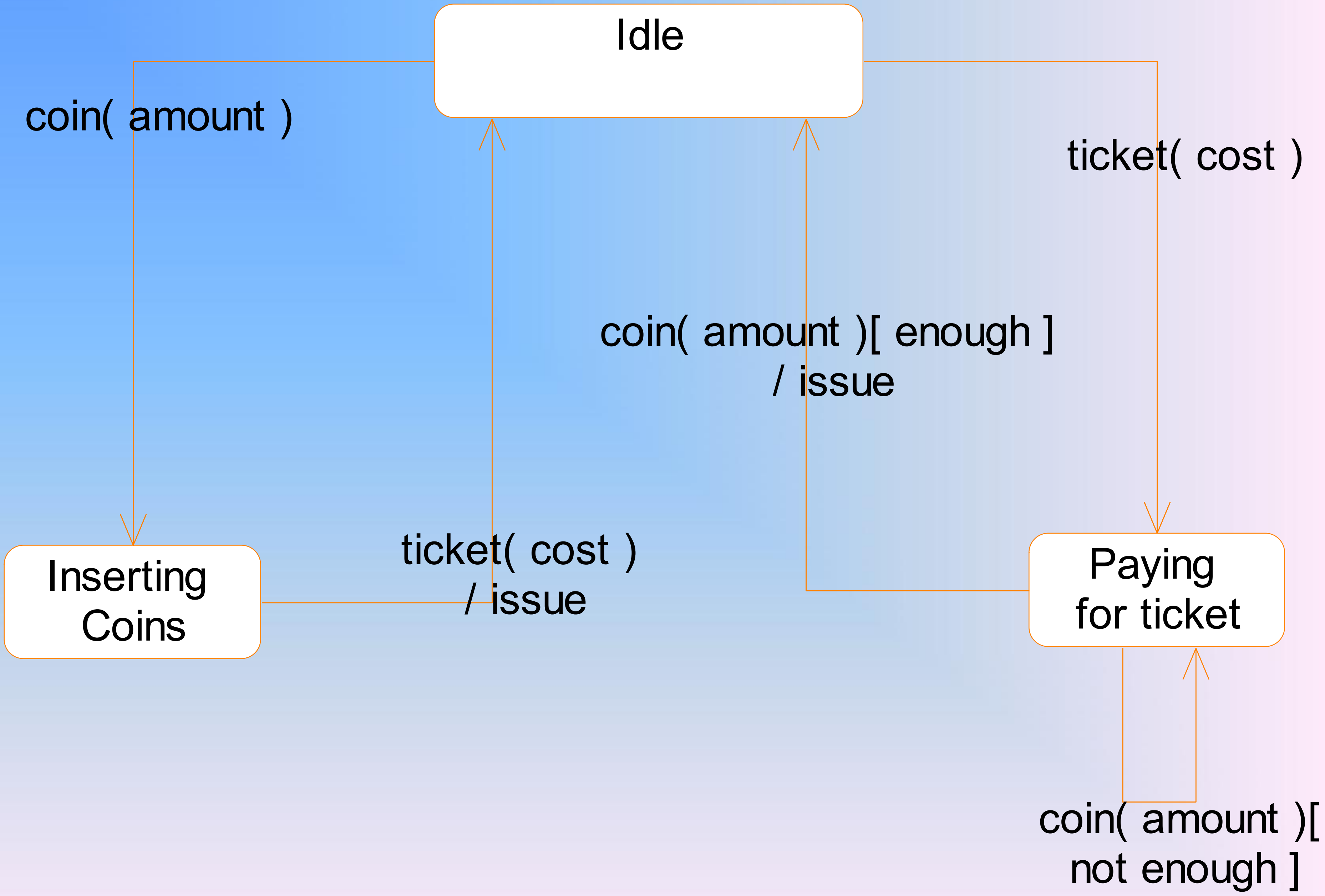
- Most stages correspond to states directly. But for the stages (e.g. stage 3) that only span short periods of duration, we can model them by actions.


coin(amount)[not enough]
Idle
ticket(cost)
Paying for ticket
coin(amount)[enough] / issue ticket and change

To model the alternative way: entering money before selecting a ticket type

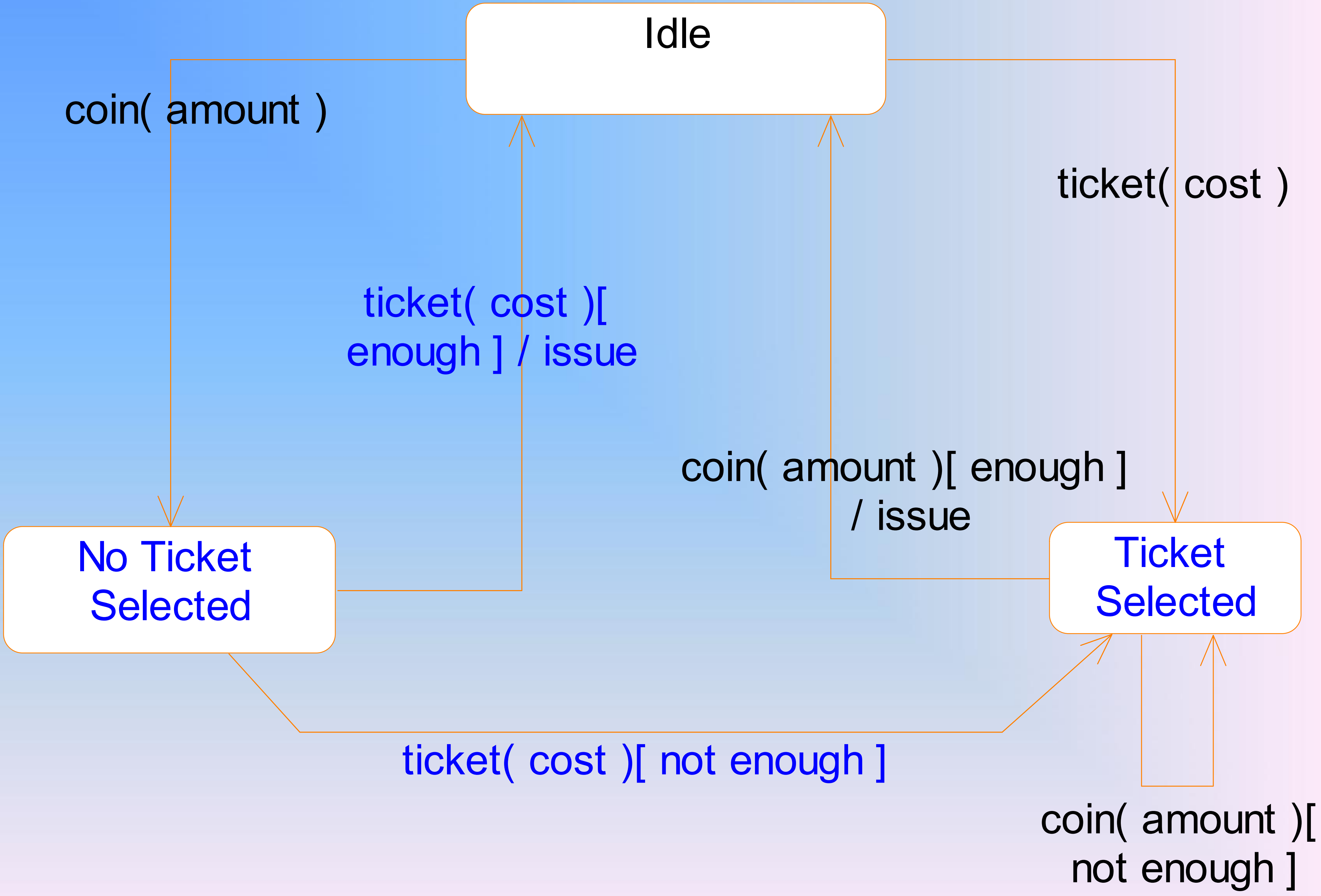
- Starting from the “idle” state, there is no transition labeled “coin”.

  → We need to create a new state


Idle
coin(amount)
ticket(cost)
coin(amount)[enough]
/ issue
Inserting
Coins
ticket(cost)
/ issue
Paying
for ticket
coin(amount)[
not enough]

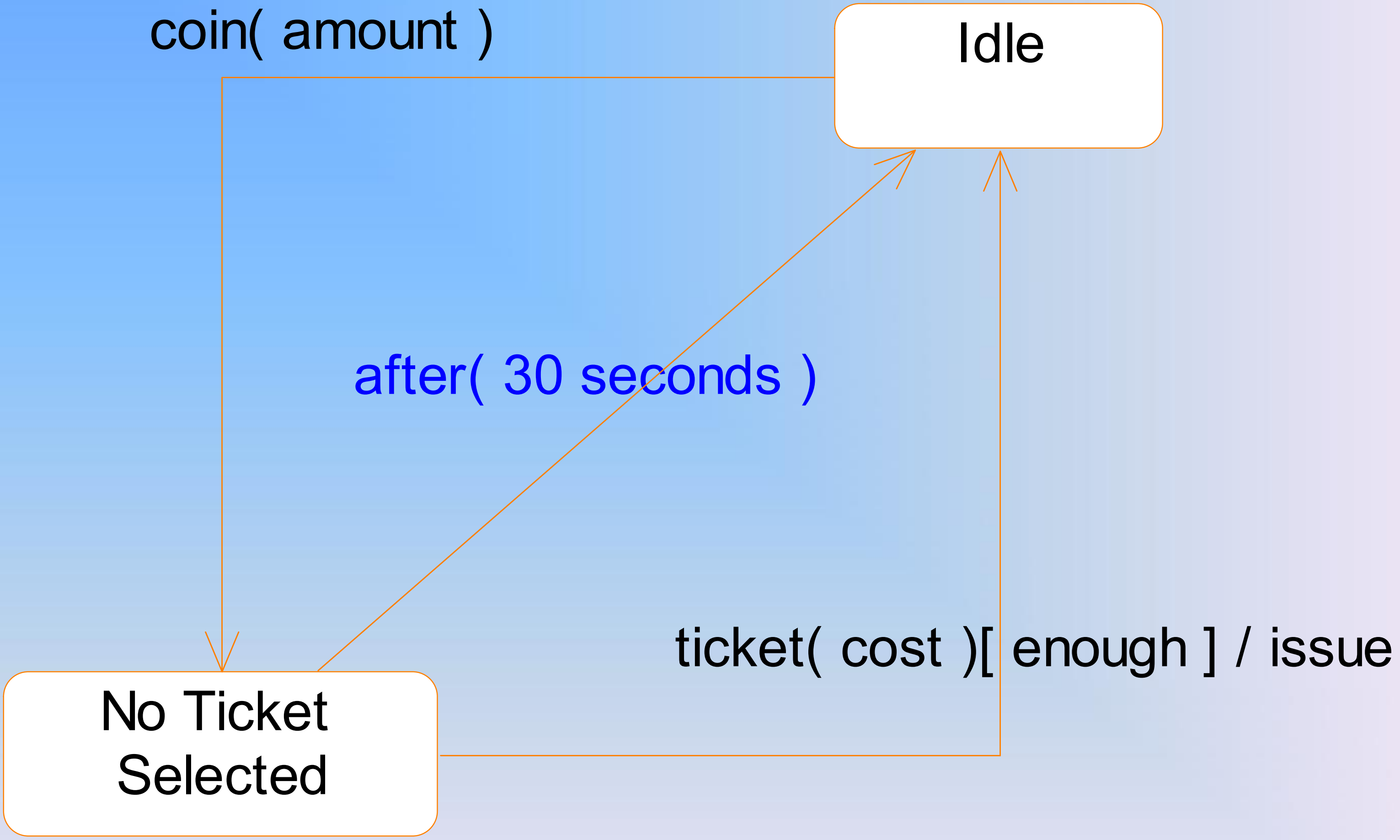
Integrating the transactions

- To model the sequence:
  - Entering some coins (but not enough)
  - Select a ticket type
  - Continue to enter coins
- We could create a new state.
  But eventually we find the new state is as the same as the existing one


Idle
coin(amount)
ticket(cost)
ticket(cost)[enough] / issue
coin(amount)[enough] / issue
No Ticket Selected
Ticket Selected
ticket(cost)[not enough]
coin(amount)[not enough]

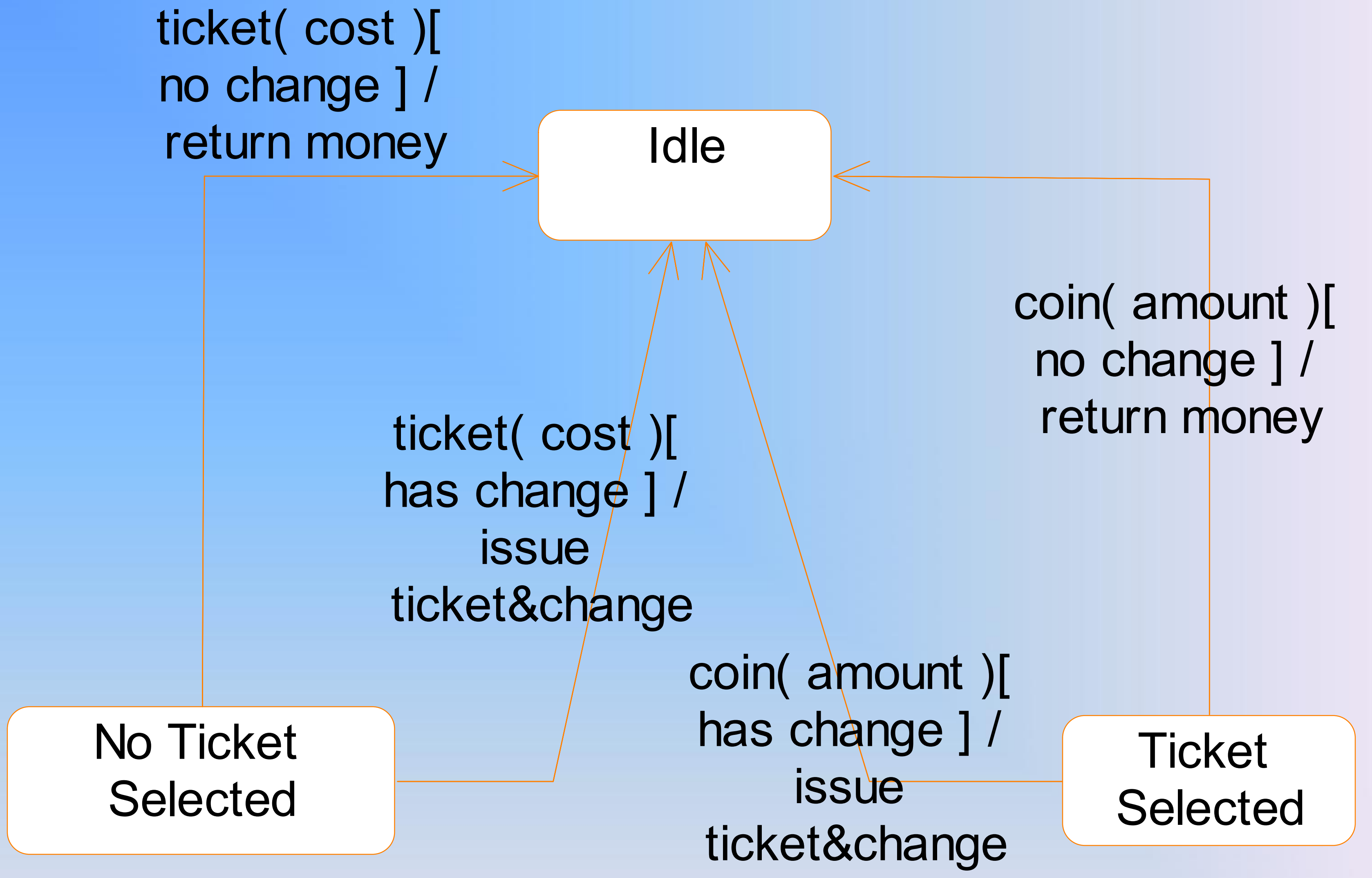
Time events

- Each state is supposed to contain an internal timer.
- The timer is reset every time the state is entered.
- The “after” event
- The “when” event


coin(amount)
Idle
after(30 seconds)
ticket(cost)[enough] / issue
No Ticket Selected

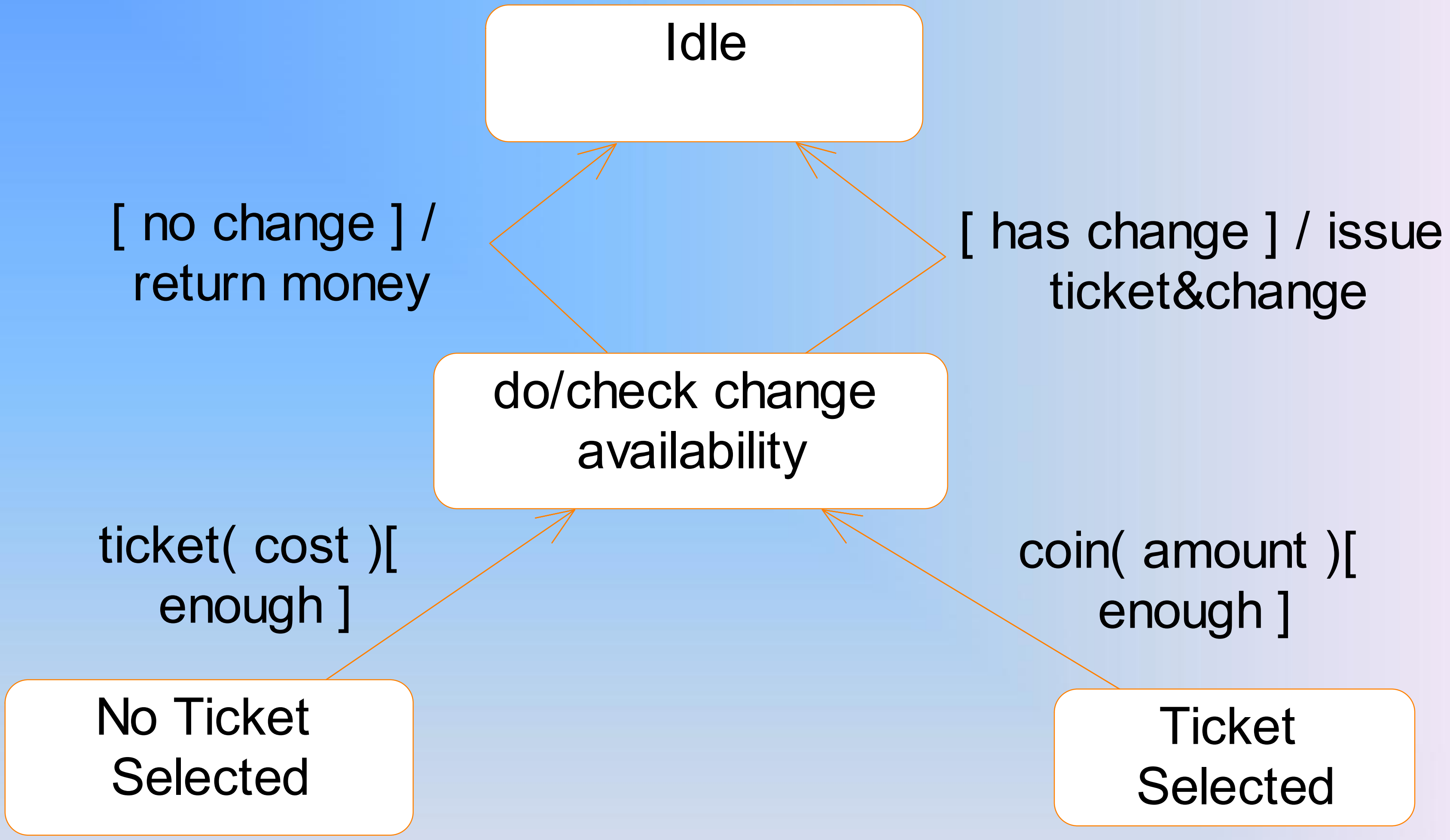
Action states

- Purpose of using action states: to avoid repetitions of transitions.
- An action state represents a period of time during which an object is performing some internal processing
- It only contains an activity
- It can not respond to external events. The transitions leaving an action state can only be completion transitions.
- Do not overuse them.

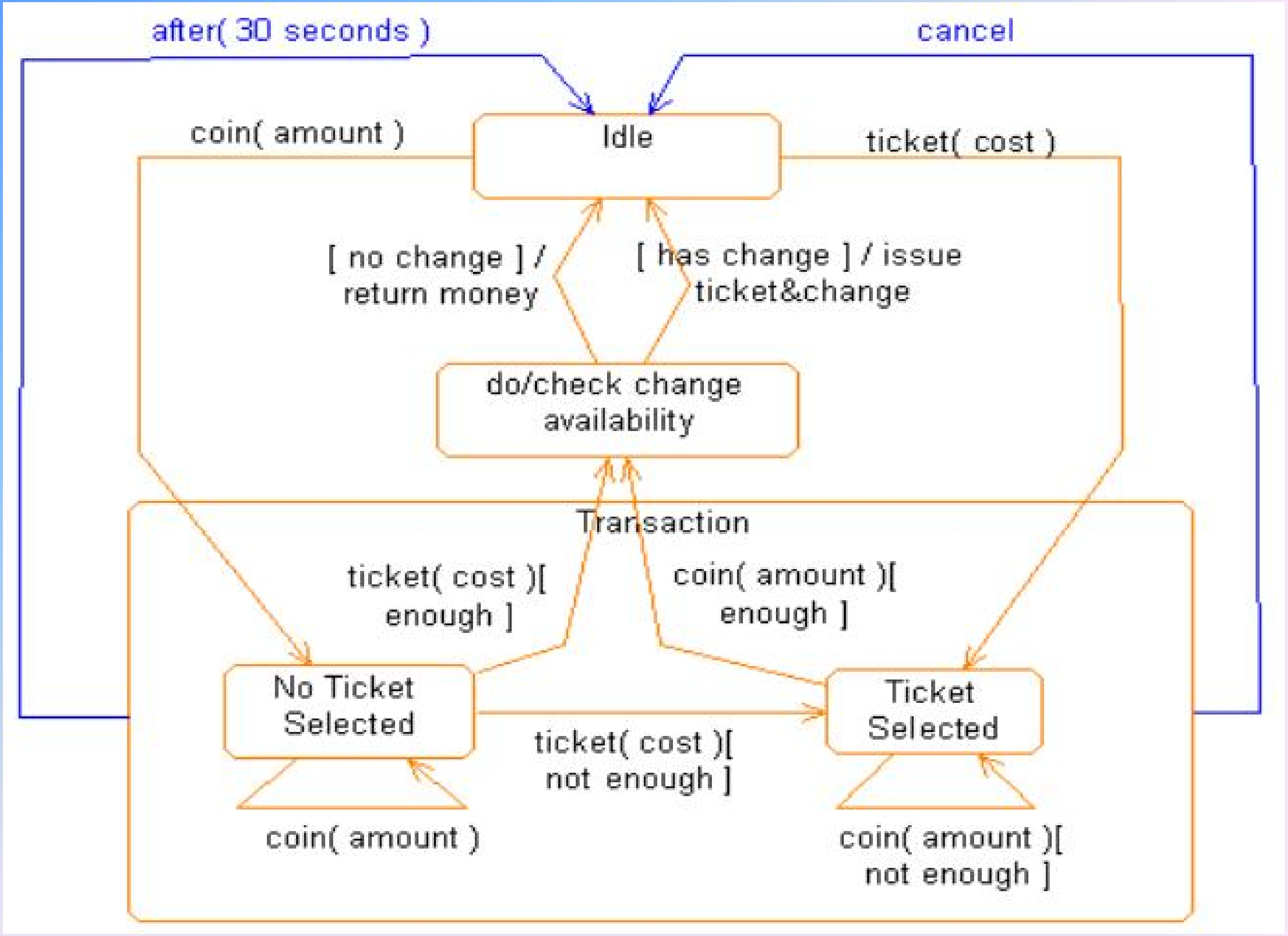


Idle
No Ticket Selected
Ticket Selected
ticket(cost)[no change] / return money
coin(amount)[no change] / return money
ticket(cost)[has change] / issue ticket&change
coin(amount)[has change] / issue ticket&change

Without action states, some transitions must be repeated.


Idle
[no change] / return money
[has change] / issue ticket&change
do/check change availability
ticket(cost)[enough]
coin(amount)[enough]
No Ticket Selected
Ticket Selected

With action states, we get a clearer figure.



The complete statechart for the ticket machine

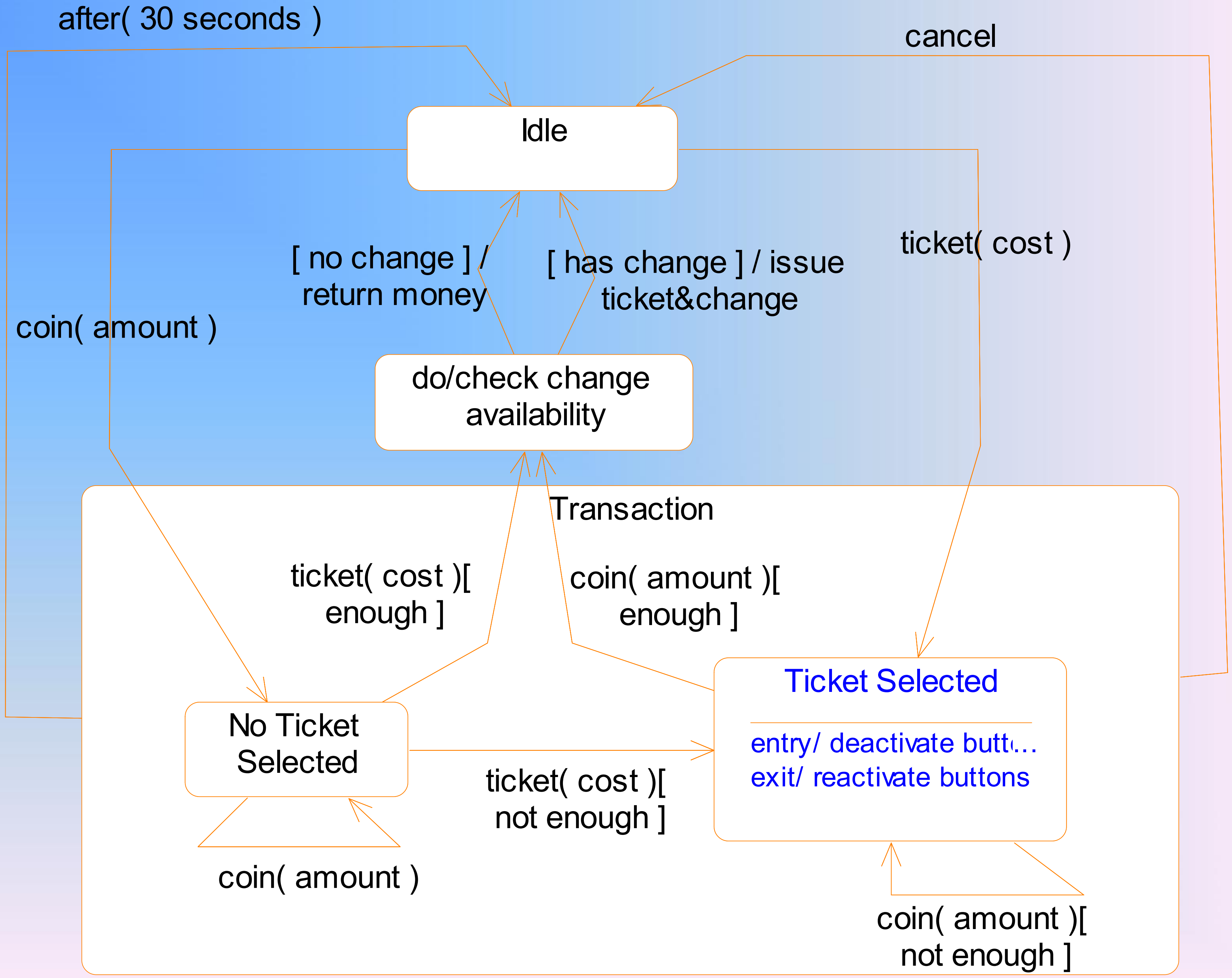
Question

- Page 203, exercise 8.6.
  1. Suppose that the ticket selection buttons are deactivated once a ticket type has been selected, and only reactivated at the end of a transaction.

  2. Suppose that once enough money has been entered to pay for the required ticket, the coin entry slot is closed, and only reopened once any ticket and change has been issued.

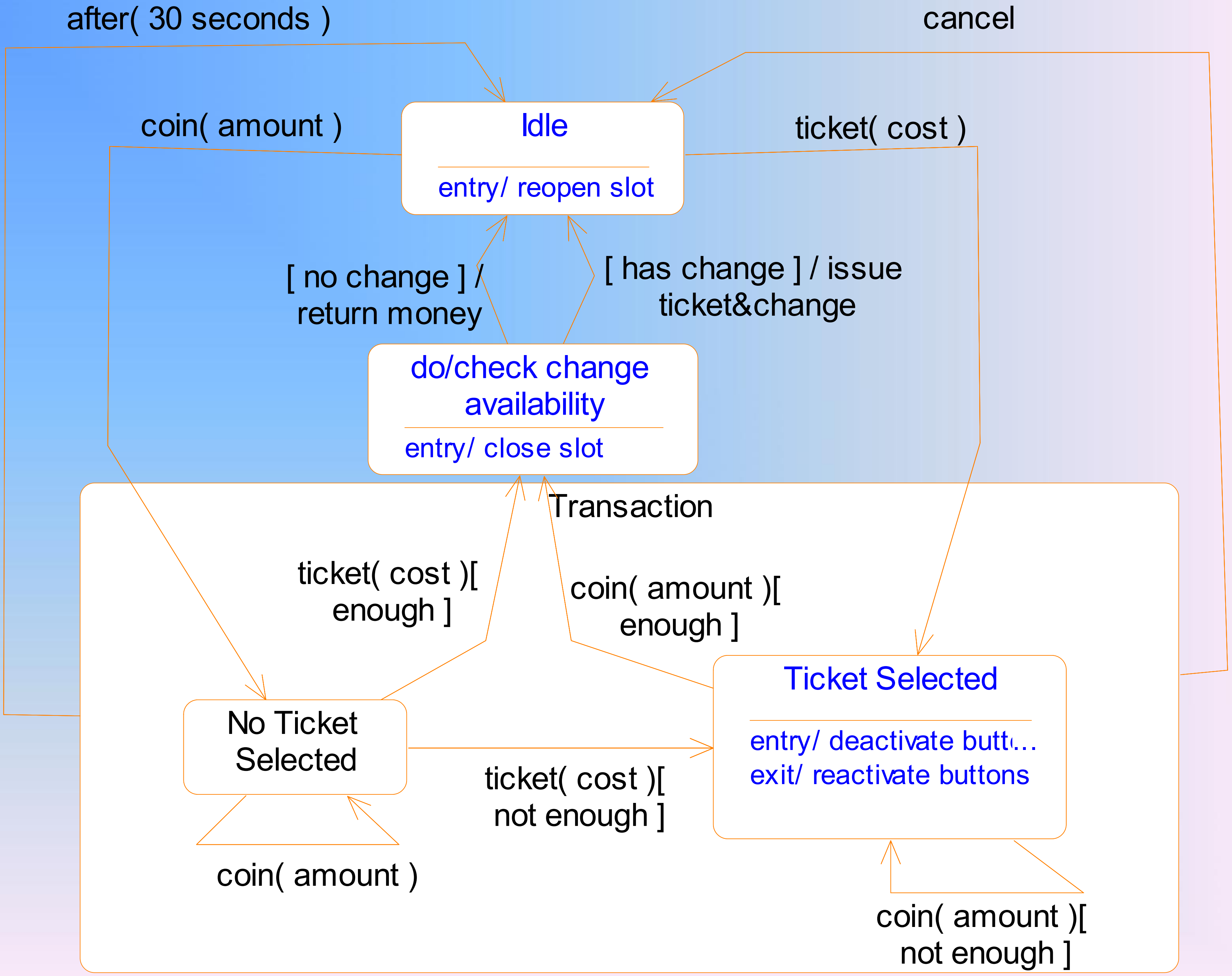
Analysis of the requirement #1

- In *Idle* or *No ticket selected* state, a single press of the ticket type button leads to transitions ending at the *ticket selected* state. Thus, we need only to study the last state.
- Add an entry action to deactivate the buttons
- Since all transitions leaving from the *ticket selected* state end at the *Idle* state, we can add an exit action into the state to reactivate the buttons.


after(30 seconds)
cancel
Idle
[no change] / return money
[has change] / issue ticket&change
ticket(cost)
coin(amount)
do/check change availability
Transaction
ticket(cost)[enough]
coin(amount)[enough]
Ticket Selected
entry/ deactivate butt...
exit/ reactivate buttons
No Ticket Selected
ticket(cost)[not enough]
coin(amount)
coin(amount)[not enough]

Analysis of the requirement #2

- In *Idle* or *No ticket selected* state, the machine does not know what type of ticket is required by the user, so it is impossible to judge whether the entered money is enough. Thus, the slot is kept open.
- In the *ticket selected* state, when enough money has been entered, the machine jumps into the action state.
- Thus, we add an entry action into the action state to close the slot.
- Since the slot is reopened only after the ticket and change is issued, we add an entry action into the idle state.

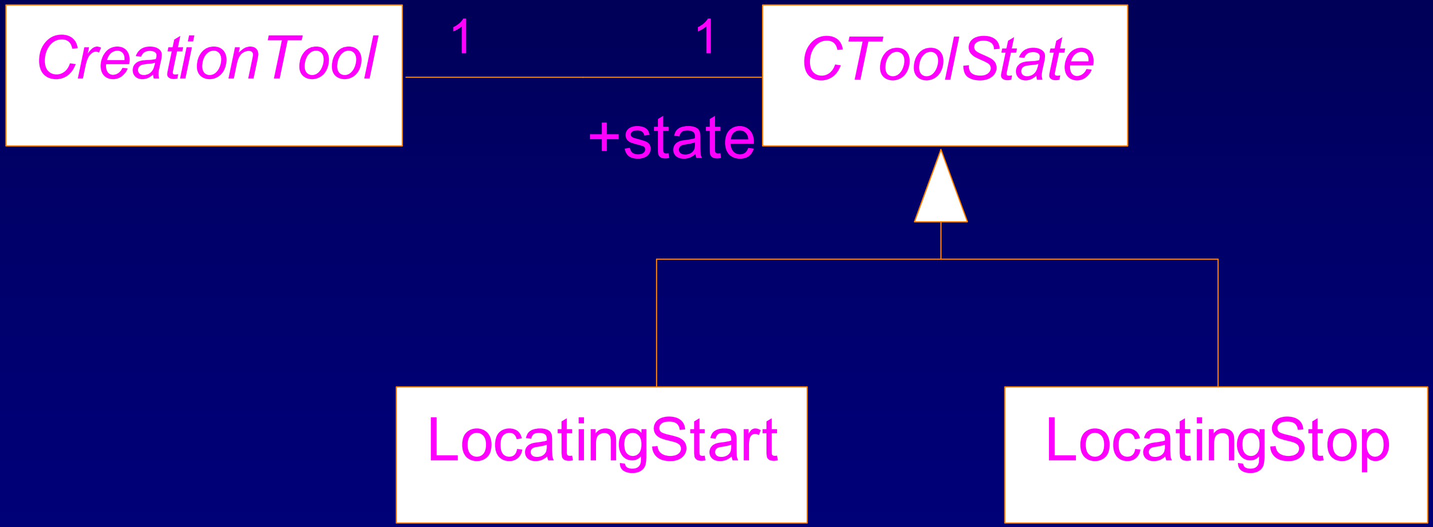

after(30 seconds)
cancel
coin(amount)
Idle
entry/ reopen slot
ticket(cost)
[no change] / return money
[has change] / issue ticket&change
do/check change availability
entry/ close slot
Transaction
ticket(cost)[enough]
coin(amount)[enough]
No Ticket Selected
Ticket Selected
entry/ deactivate butt...
exit/ reactivate buttons
ticket(cost)[not enough]
coin(amount)
coin(amount)[not enough]

Implementation of statecharts

- Solution 1
  - The states are modeled by an enumeration variable.
  - “Switch” statements are used for the member functions whose behaviors varies with the state.
- Drawbacks
  - When a new state is added/removed, all functions should be changed.

Implementation of statecharts

- Solution 2:
  - Suppose class A has *n* states. Each state is modeled by a *state class*.
  - All the state classes are generalized to a root class.
  - Class A holds a pointer (to the root class).
  - For incoming messages, class A simply pass them on to the object representing the current state.



```
class CToolState {
  virtual void Press()=0;
};
class CreationTool{
   CToolState* state;
  void Press() {state->Press();}
}
```

Solution 2 (continued)

- The root class has an interface to process all messages sent to class A.
- Each state class only re-define functions for the messages that interest it.
- Each state is capable of destroying itself, creating a new object representing a new state, and setting the current state to the new state.
  ➔ bidirectional association between class A and the root class is necessary.

```
class CToolState;
class CreationTool{
    CToolState* pCurState; // points to the current state
public:
   void Press();
   void ChangeState( CToolState * pNewState);
};

class CToolState {
protected:
   CreationTool * pCreationTool; // bidirectional
```

<u>Skeleton of the implementation:1/4</u>

```
public:
   CToolState(CreationTool * p);
   void ChangeState(CToolState * s);
   virtual void Press();
};
void CreationTool::Press() {  pCurState->Press(); }
void CreationTool::ChangeState(CToolState *pNewState)
{  delete pCurState;
   pCurState = pNewState;
}

CToolState::CToolState(CreationTool * p)
{  pCreationTool = p;
}
```

Passing message

Skeleton of the implementation:2/4

```
void CToolState::ChangeState(CToolState * s)
{  pCreationTool->ChangeState(s);
}
void CToolState::Press() {}

class LocatingStop : public CToolState {
public:
   LocatingStop( CreationTool * p): CToolState(p) {}
};
```

Skeleton of the implementation:3/4

```
class LocatingStart: public CToolState {
public:
   LocatingStart( CreationTool * p): CToolState(p) {}
    void Press() {
        // set start position
        // draw faint image
        ChangeState(new LocatingStop(pCreationTool) );
   }
}

main() { }
```

Skeleton of the implementation:4/4