Input: Implementing Interaction Techniques as Finite State Machines

Administration

- HW4a due today
- HW5 set today

Interaction Techniques

- A method for carrying out a specific interactive task
  - Example: enter a number in a range
    - Could use … (simulated) slider
    - (simulated) knob
    - Type in a number (text edit box)
  - Each is a different interaction technique

How do we implement interaction techniques?

- Focus of today’s lecture
- Important for understanding existing techniques
- Important for designing and building your own:
  - Why not just use existing ones?

Suppose we wanted to implement an interaction for specifying a line

- **Could just specify two endpoints**
  - **click, click**
  - **not good: no affordance,no feedback**
- **Better feedback is to use "rubber banding"**
  - **stretch out the line as you drag**
  - **at all times, shows where you would end up if you "let go"**

Aside

- Rubber banding provides good feedback
- How would we provide better affordance?

Aside

- Rubber banding provides good feedback
- How would we provide better affordance?
  - Changing cursor shape is about all we have to work with

Implementing rubber banding

```
Accept the press for endpoint p1;
P2 = P1;
Draw line P1-P2;
Repeat
  Erase line P1-P2;
  P2 = current_position( );
  Draw line P1-P2;
Until release event;
Act on line input;
```

Implementing rubber banding

- Need to get around this loop absolute min of 5 times / sec
  - 10 times better
  - more would be better
- Notice we need “undraw” here

2^{nd} Aside: How do we do "undraw" in a frame buffer?

- **Writes to frame buffer memory are destructive** (old background lost)

2^{nd} Aside: How do we do "undraw" in a frame buffer?

- Writes to frame buffer memory are destructive (old background lost)

- Two major alternatives:
  - XOR
  - Completely redraw the image from some description (e.g., interactor tree)

What’s wrong with this code?

```
Accept the press for endpoint p1;
P2 = P1;
Draw line P1-P2;
Repeat
  Erase line P1-P2;
  P2 = current_position( );
  Draw line P1-P2;
Until release event;
Act on line input;
```

Not event driven

- Not in the basic event / redraw cycle form
  - don’t want to mix event and sampled
  - in many systems, can’t ignore events for arbitrary lengths of time
- How do we do this in a normal event / redraw loop?

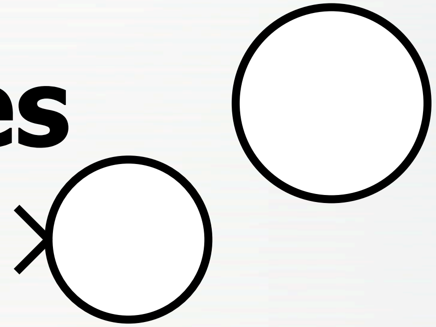
You don’t get to write control flow in event driven systems

- Control is in the hands of the user
- Basically have to chop up the actions in the code above and redistribute them in event driven form
  - “event driven control flow”
  - need to maintain “state” (where you are) between events and start up “in the state” you were in when you left off
- Examples from assignments?

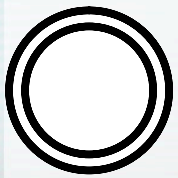
Finite state machine controllers

- One good way to maintain "state" is to use a state machine
  - Finite State Machine (FSM)
  - Has a collection of states the system could be "in"
    - One current state
  - Events cause you to move from current state to other states (or back to same state)
    - And execute actions as you move

FSM notation



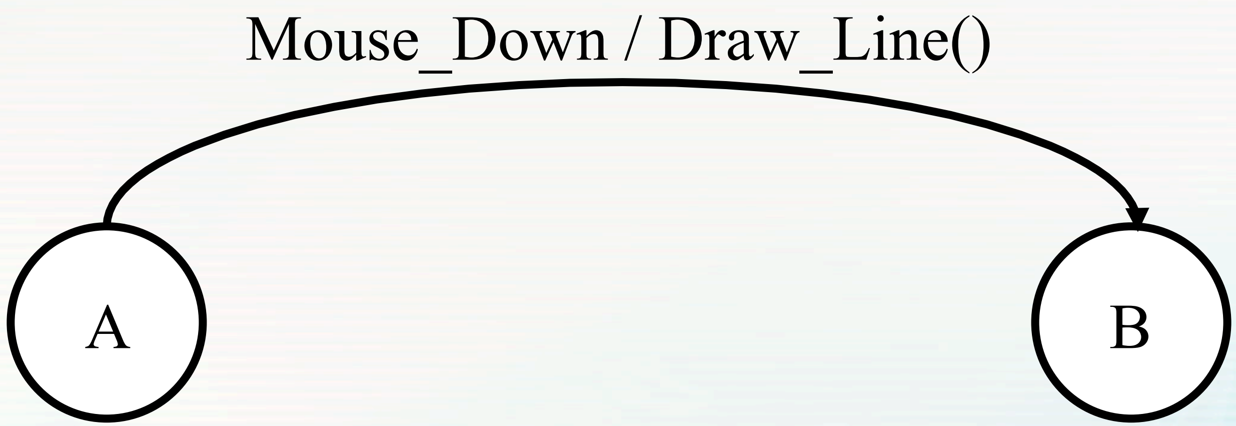
- Circles represent states
  - arrow for start state
    - Begin the interaction in this state
  - double circles for “final states”



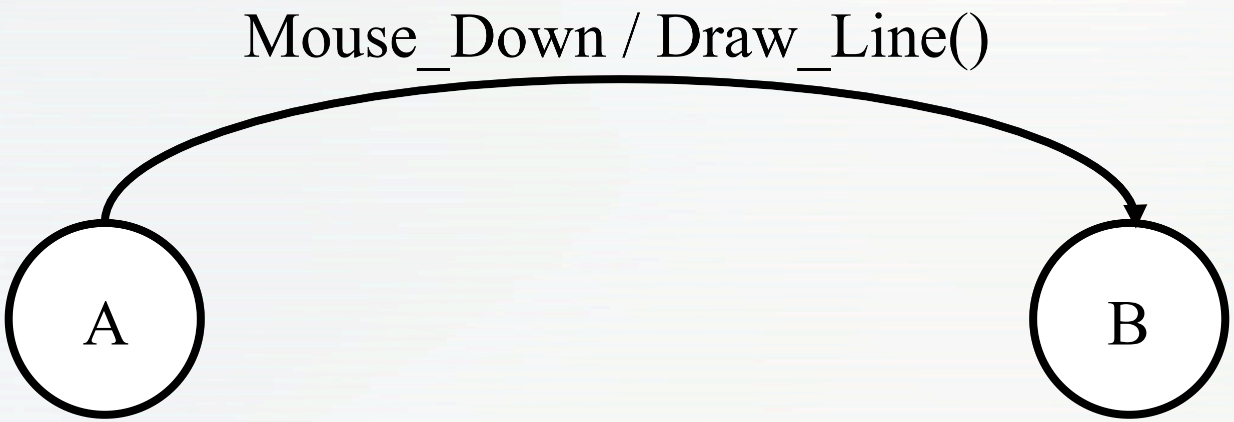
    - Typically not really “final”, just denoting end of part of interaction
    - Typically means you reset to start state

FSM notation

- **Transitions represented as arcs**
  - **Labeled with a "symbol"**
    - **for us an event (can vary)**
  - **Also optionally labeled with an action**



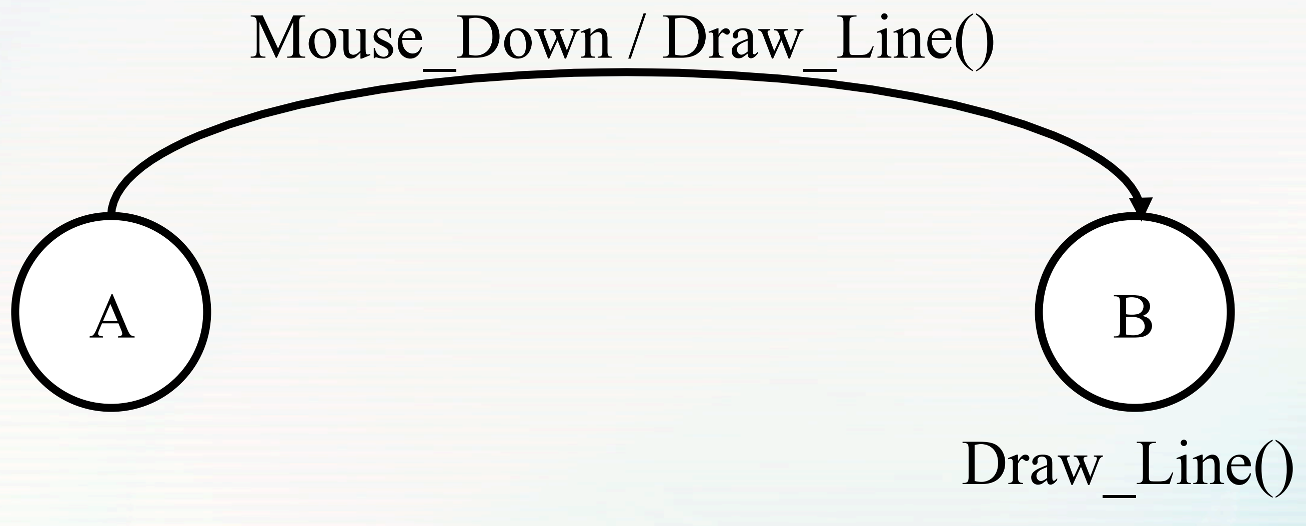
FSM Notation



- **Means: when you are in state A and you see a mouse down, do the action (call draw_line), and go to state B**

FSM Notation

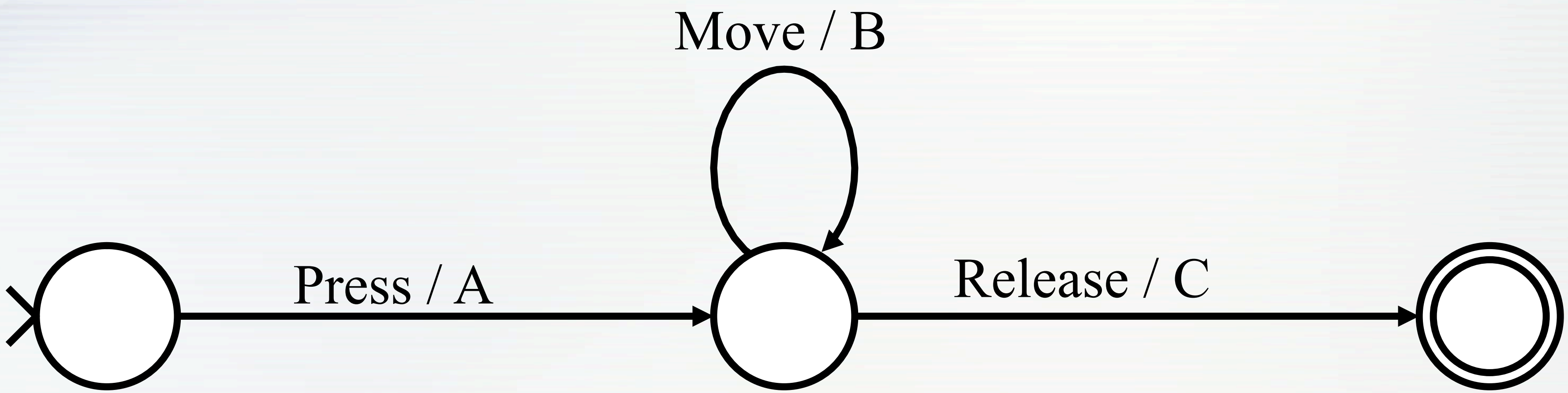
- **Sometimes also put actions on states**
  - **same as action on all incoming transitions**



Rubber banding again (cutting up the code)

```
      Accept the press for endpoint p1;
A:    P2 = P1;
      Draw line P1-P2;
      Repeat
B:        Erase line P1-P2;
          P2 = current_position();
          Draw line P1-P2;
      Until release event;
C:    Act on line input;
```

FSM control for rubber banding



A:
```
P2 = P1;
 Draw line P1-P2;
```

B:
```
Erase line P1-P2;
 P2 = current_position();
 Draw line P1-P2;
```

C:
```
Act on line input;
```

FSM control for rubber banding

How does this work:
demonstration!

5 volunteers:
- 3 states
- 1 event actor
- 1 user

Example #2: Button

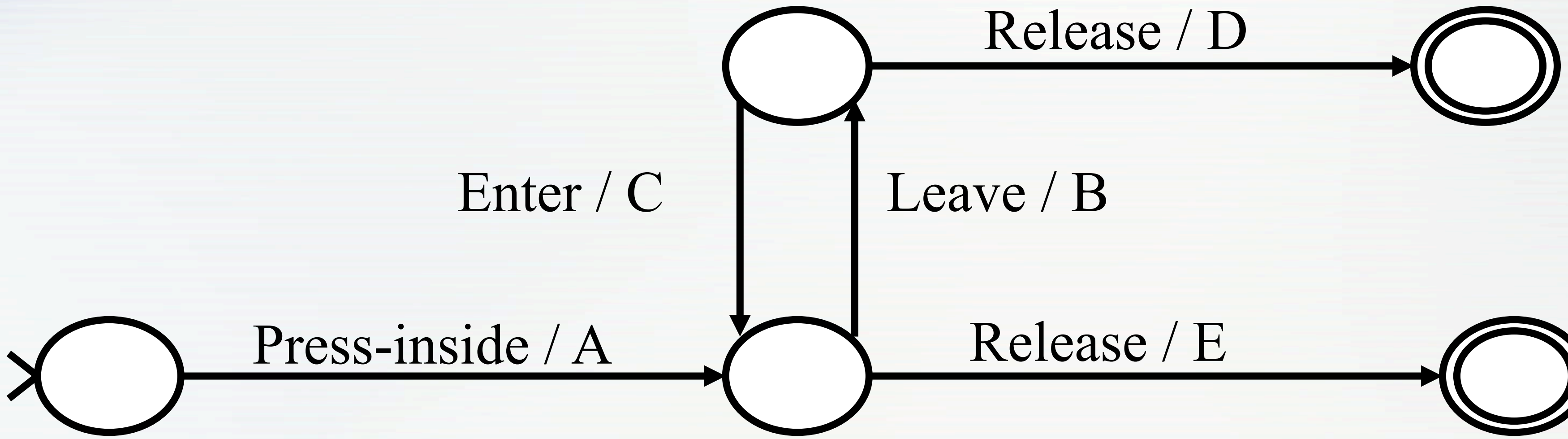
- For drawing a line, had to represent
  - Clicking the first point
  - Moving the cursor
  - Clicking the second point
- What kinds of things do we need to represent for buttons?

Second example: button

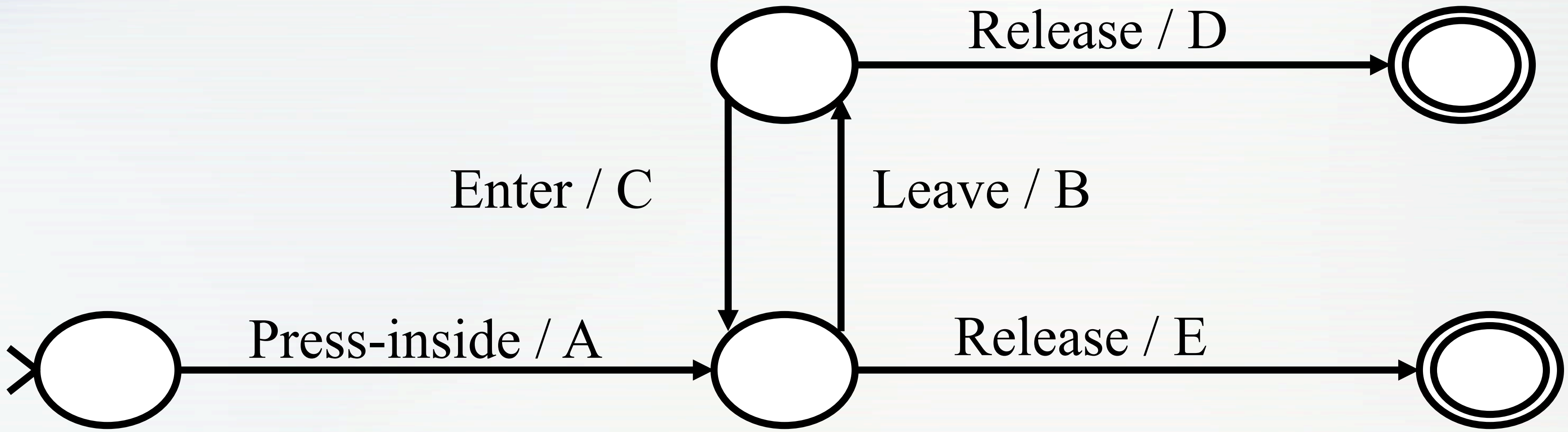
Press inside	**=> highlight**
Move in/out	**=> change highlight**
Release inside	**=> act**
Release outside	**=> do nothing**

FSM for a button?

FSM for a button



FSM for a button



A: highlight button
B: unhighlight button
C: highlight button
D: <do nothing>
E: unhighlight; do button action

FSM control for buttons

How does this work:
demonstration!

7 volunteers:

- 5 states
- 1 event actor
- 1 user

Now your turn!

- Document window with text in it and a scrollbar on one side
- What’s the FSM for the scrollbar thumb?

- 1 user
- 1 event actor
- N(?) states

- **What’s the FSM for the scrollbar if the user just clicks on the scrollbar?**

- **1 user**
- **1 event actor**
- **N(?) states**

In general...

- **Machine states represent context of interaction**
  - **"where you are" in control flow**
- **Transitions indicate how to respond to various events**
  - **what to do in each context**

"Events" in FSMs

- What constitutes an "event" varies
  - may be just low level events, or
  - higher level (synthesized) events
    - e.g. region-enter, press-inside
    - Also things you might not think of like time passing

Guards on transitions

- Sometimes also use "guards"
  - predicate (bool expr) before event
  - adds extra conditions required to fire
  - typical notation:

    expression: event / action

    - e.g. button.enabled: press-inside / A

FSM are a good way to do control flow in event driven systems

- Can do (formal or informal) analysis or reasoning about UI
  - are all possible inputs (e.g. errors) handled from each state?
  - what are next legal inputs
    - can use to enable / disable

Implementing FSMs

```
state = start_state;
for (;;) {
  raw_evt = wait_for_event();
  events = transform_event(raw_evt);
  for each evt in events {
     state = fsm_transition(state, evt);
  }
}
```

- **Note that this is basically the normal event loop**

Implementing FSMs

```
fsm_transition(state, evt)
   switch (state)
     case 0:   // case for each state




     case 1: // case for next state


return state;
```

Implementing FSMs

```
fsm_transition(state, evt)
  switch (state)
    case 0:   // case for each state
      switch (evt.kind)
        case loc_move: // trans evt
          … action …   // trans action
          state = 42;  // trans target
        case loc_dn:
          ...
    case 1: // case for next state
      switch (evt.kind) …

return state;
```

Implementing FSMs

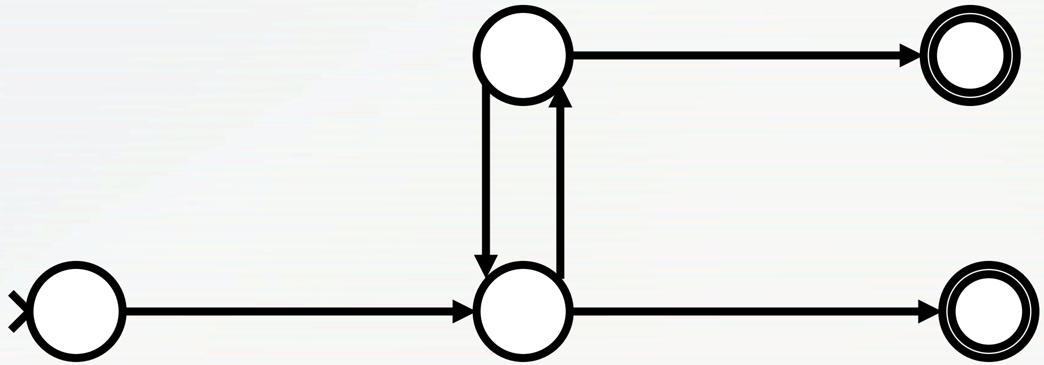
```
fsm_transition(state, evt)
  switch (state)
    case 0:   // case for each state
      switch (evt.kind)
        case loc_move: // trans evt
          … action …   // trans action
          state = 42;  // trans target
        case loc_dn:
          ...
    case 1: // case for next state
      switch (evt.kind) …

return state;
```

FSM Issues

- Notation
  - Graphical notation is nice for small things, but doesn't scale (spaghetti)
  - Textual notation is not nice
    - Like all GOTO control flow
- Handles sequencing well, but not independent action
  - State explosion problems

State explosion problems

- Suppose you had a button



- And you want to add an option to modify its action with ctrl key
  - Changes label and action

Modified button example

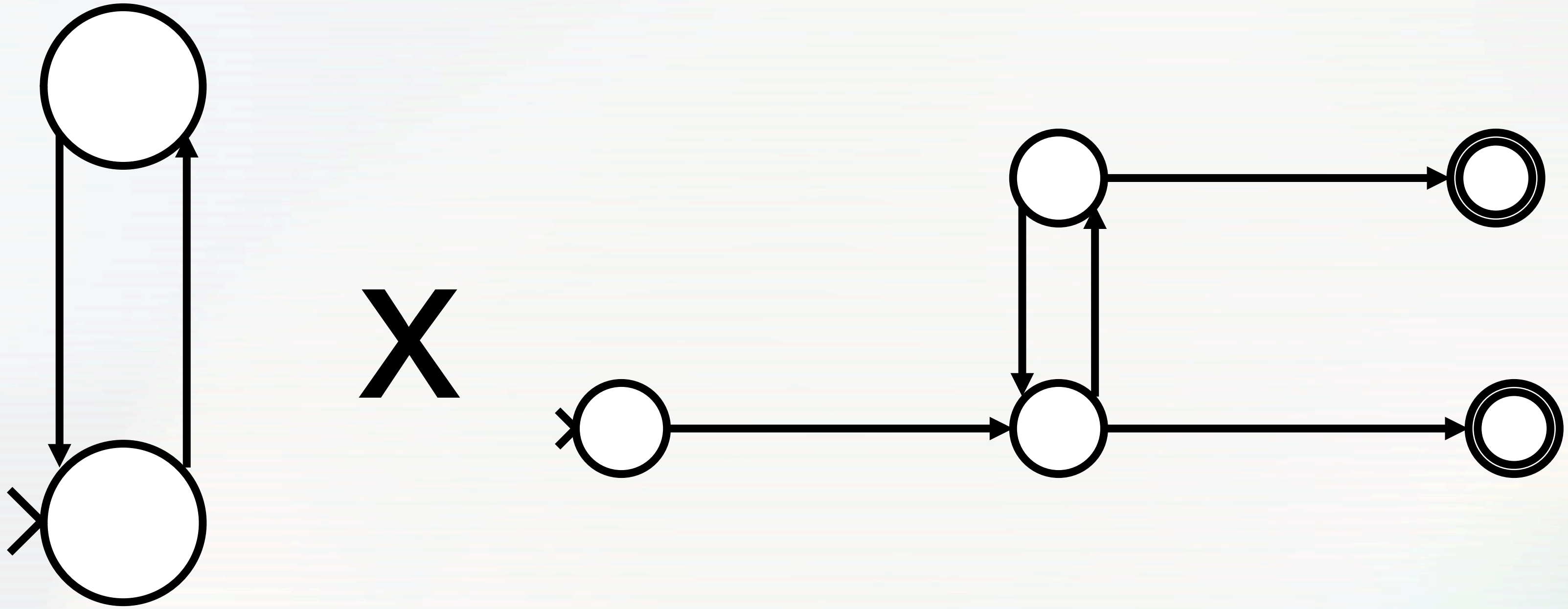
- What does tracking the control key look like?

Modified button example

- **Control key**

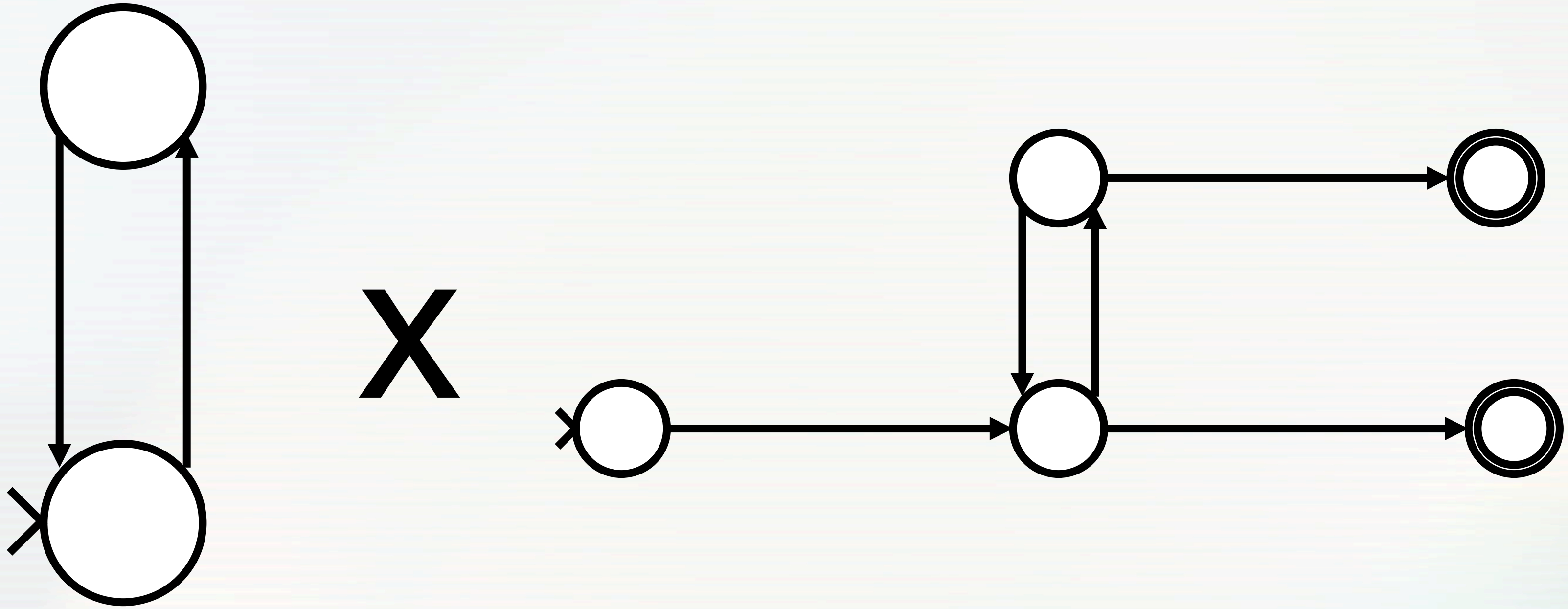
Modified button example

- **Control key x Button**



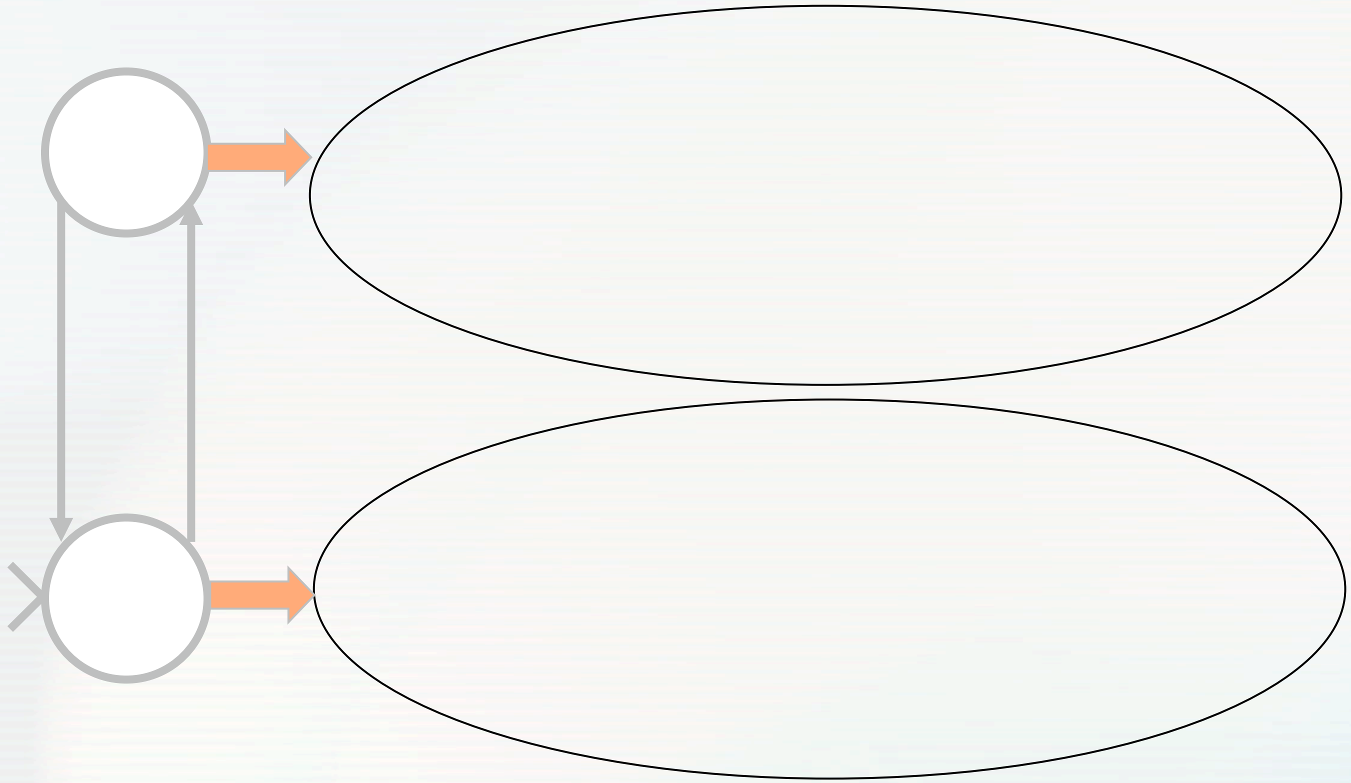
Modified button example

- **Transitions are really independent**
  - ➔ **"Cross-product" machine**



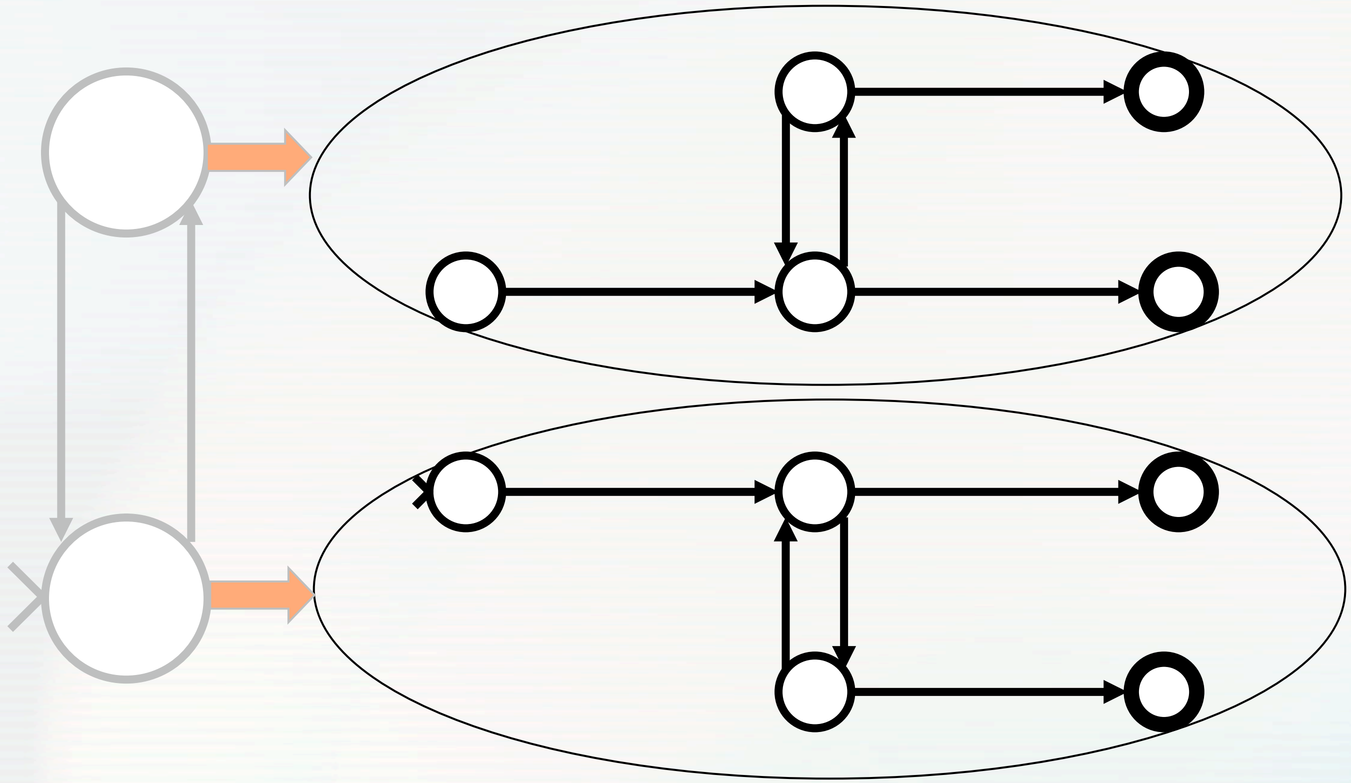
Cross product machines

- **Replicate machine A once for every state in machine B**



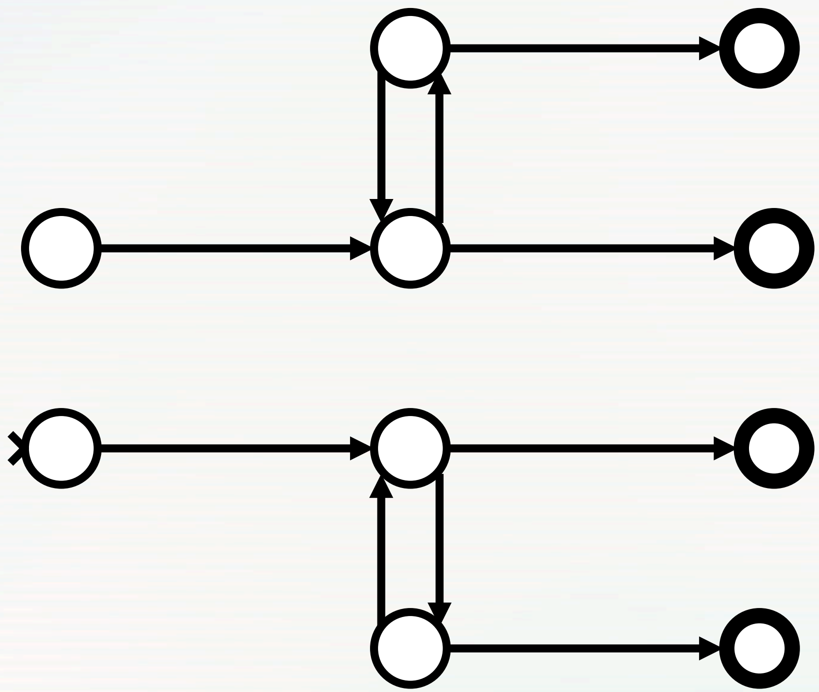
Cross product machines

- **Replicate machine A once for every state in machine B**



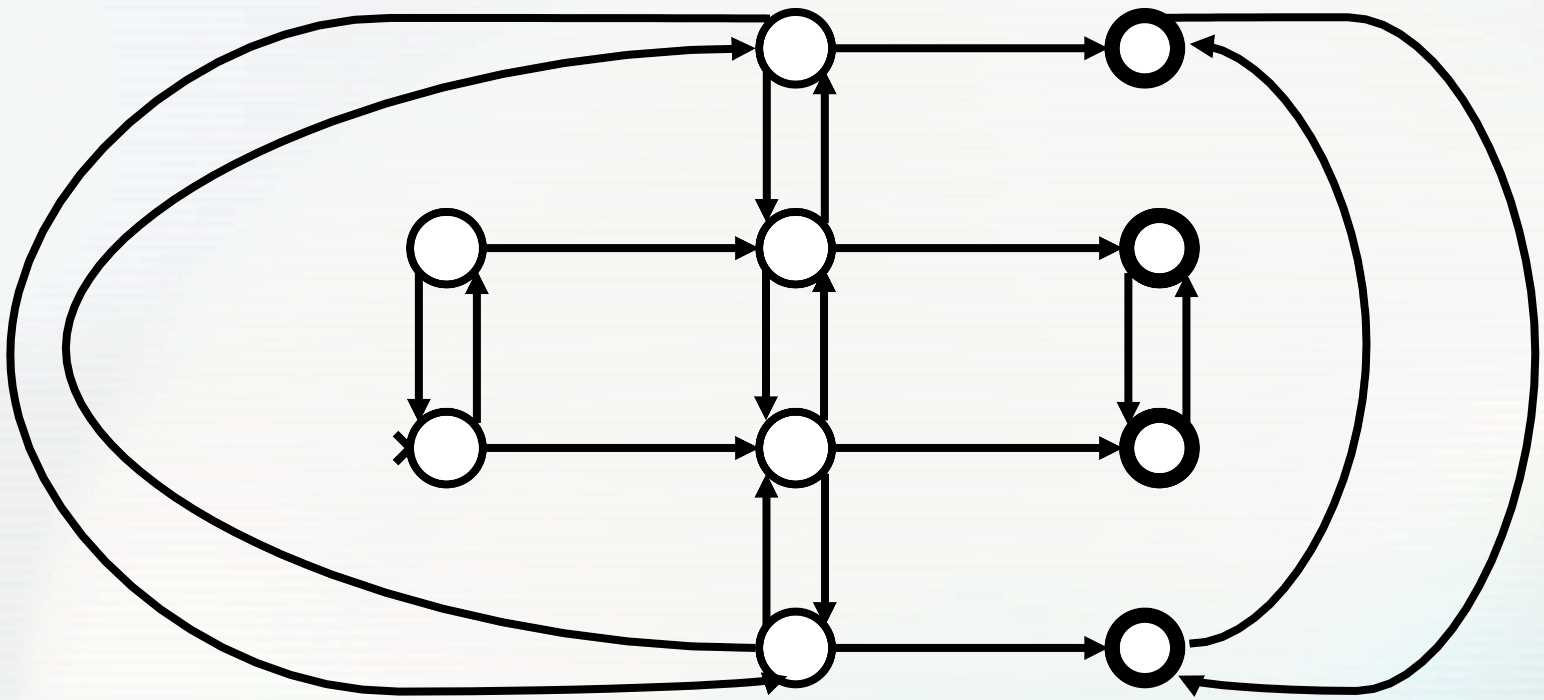
Cross product machines

- **Replicate machine A once for every state in machine B**



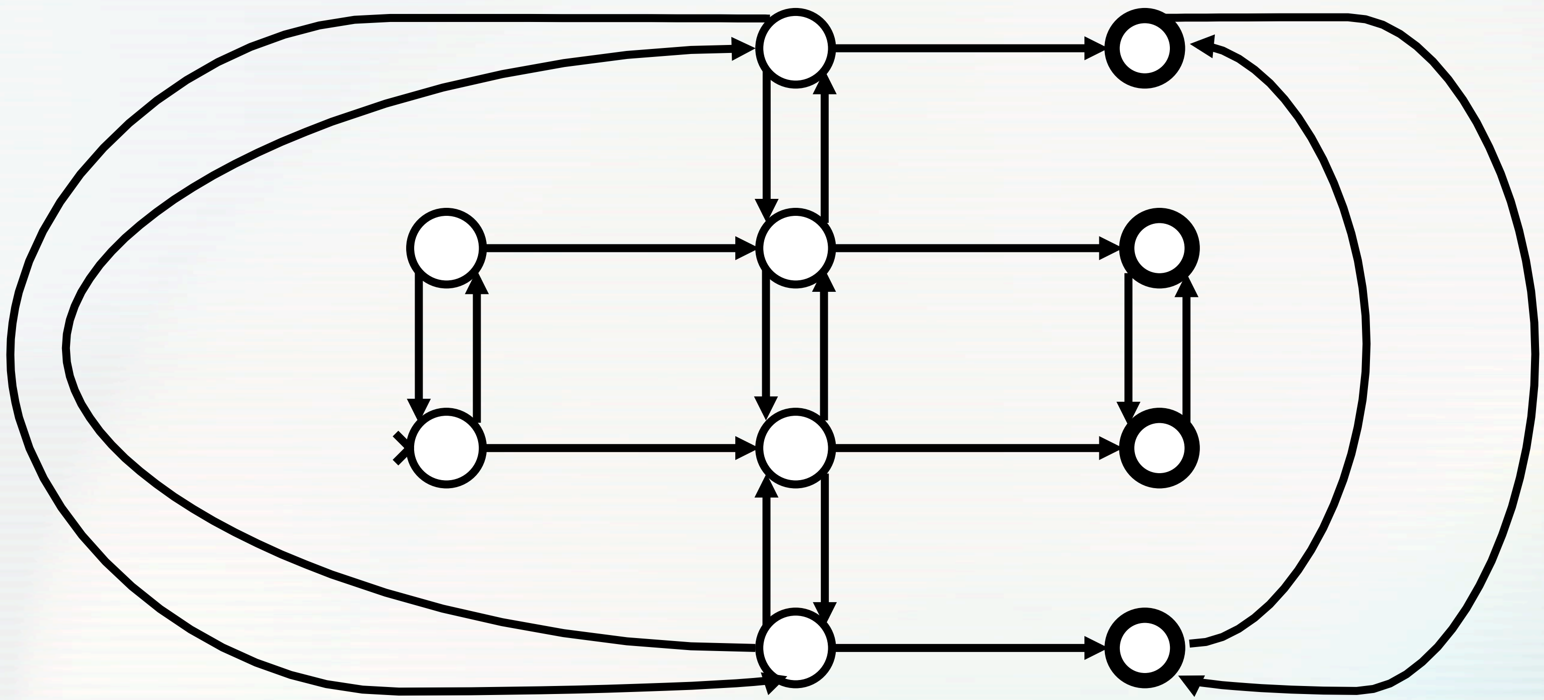
Cross product machines

- **Add transitions from machine B between corresponding states**



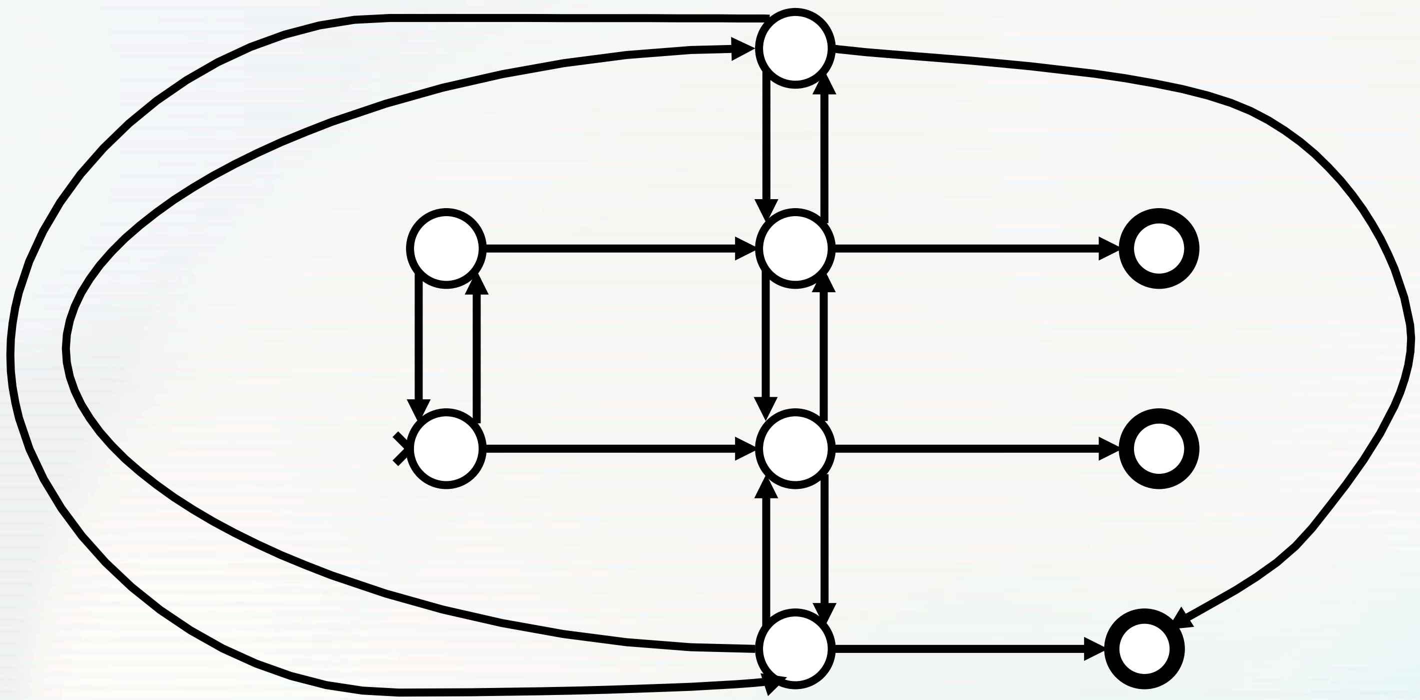
Cross product machines

- **Correct and simplify based on semantics**



Cross product machines

- **Correct and simplify based on semantics**



Now suppose we add another independent action (shift key?)

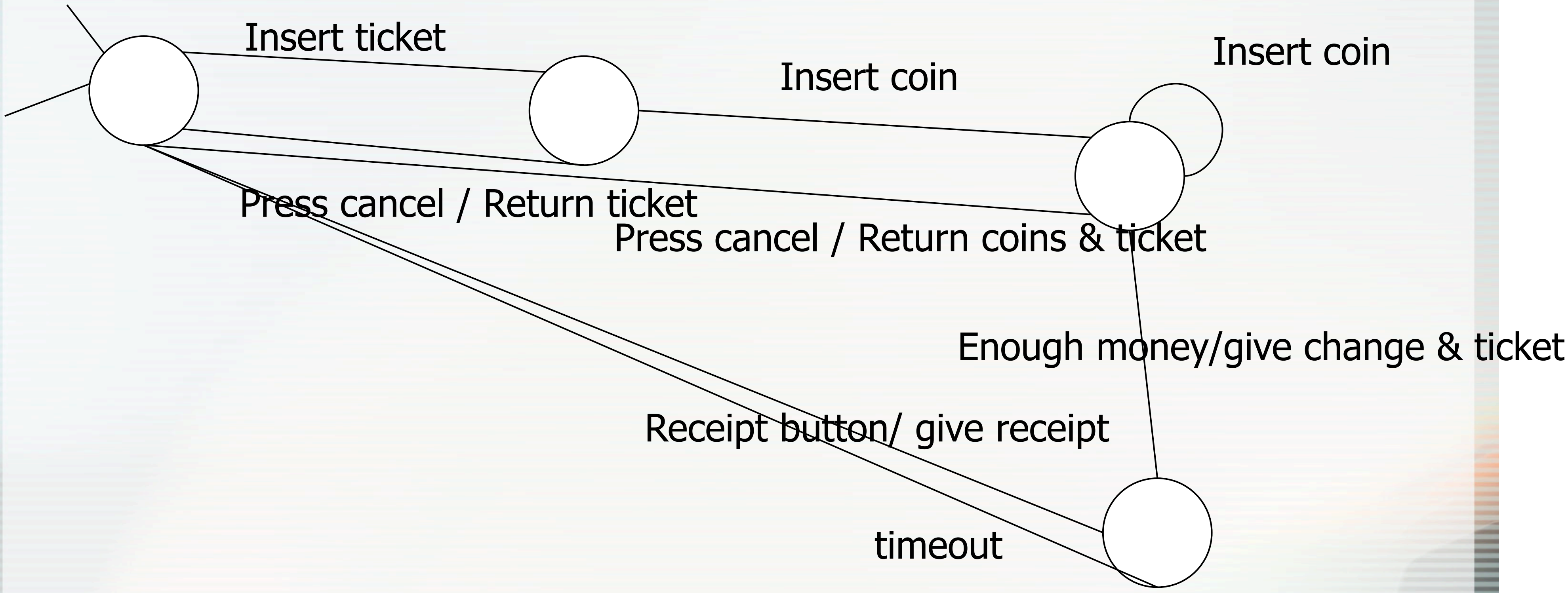
Now suppose we add another independent action (shift key?)

- Same pattern
  - But, gets really ugly
  - Won't attempt it here
- Quickly get combinatoric explosion
  - Big drawback of FSM

State machines very useful, but do have limits

- State machines don't handle independent actions very well
- Mostly useful for smaller things
  - Great for individual components
  - Not so great for whole dialogs
- Path of least resistance is rigid sequencing
  - Ask: is this good for what I am doing?

Questions?


Insert ticket
Insert coin
Insert coin
Press cancel / Return ticket
Press cancel / Return coins & ticket
Enough money/give change & ticket
Receipt button/ give receipt
timeout