Embedded Systems Design and Modeling

Chapter 5
Composition of State Machines

Outline

- Introduction
- Finite state machines as actors
- Composition techniques:
  - Side-by-side synchronous
  - Side-by-side asynchronous
  - Cascade
  - Hierarchical models

Introduction

- Important engineering principle: larger, more complex systems can be built by putting together smaller, simpler ones
- Definition: putting systems together = composition of systems
- Our focus: composition of state machines
- In general, this can be quite complicated and confusing:
  - Compositions with the same syntax can have different semantics

Important Questions

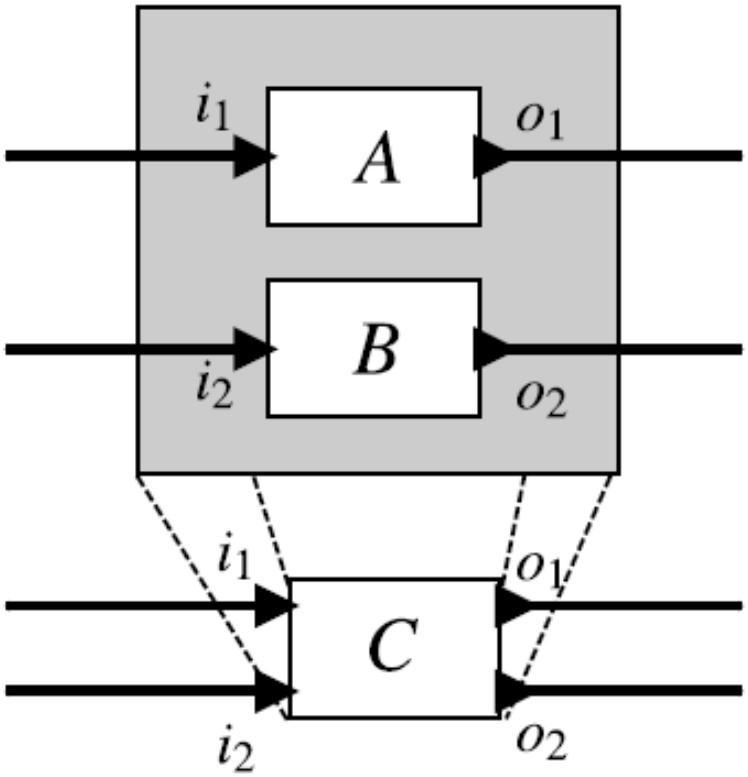
- How to compose a new system considering:
  - systematic design?
  - systematic analysis?
  - effective computer program manipulation?
- Which subsystems are required to make bigger systems?
- How to check whether the system satisfies its specification in its operating environment?

Composition Techniques

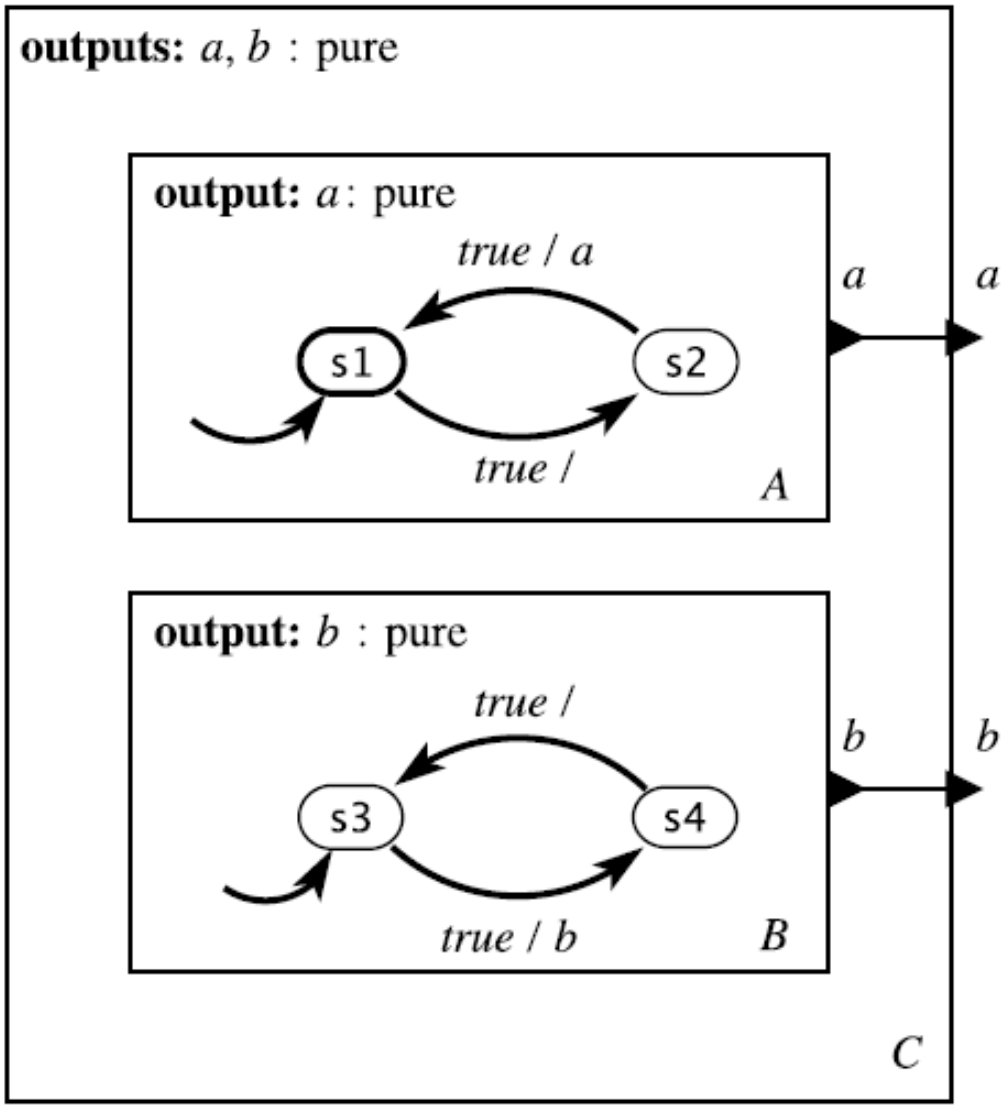
- When the machines react?
  1. Concurrent composition
     - Synchronous composition (simultaneous)
     - Asynchronous composition (independent)
  2. Hierarchical: a state is itself another FSM
- Relative position of the machines?
  1. Side-by-side
  2. Cascade
  3. Feedback

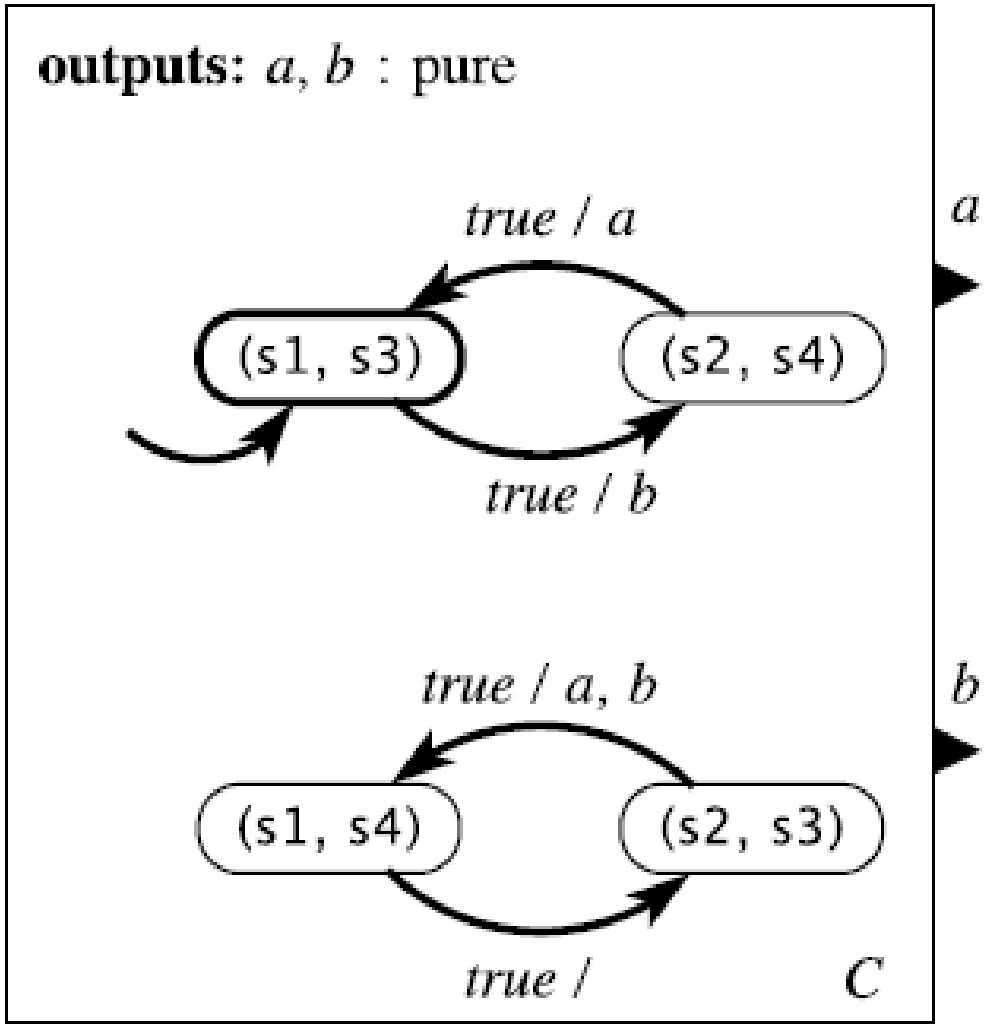
Side-by-side Synchronous

- Assumptions:
  - Inputs and outputs of the actors are disjoint
  - State variables can be disjoint or shared
  - The actors A and B react simultaneously
  - The overall actor (C) reacts simultaneously



Example


outputs: a, b : pure
output: a : pure
true / a
s1
s2
true /
A
a
a
output: b : pure
true /
s3
s4
true / b
B
b
b
C


outputs: a, b : pure
true / a
(s1, s3)
(s2, s4)
true / b
a
true / a, b
(s1, s4)
(s2, s3)
true /
b
C

Side-by-side Synchronous Features

- It is simple to design and analyze
- Composition of two FSM's is an FSM
- If the two state machines are determinate, the composition FSM is also determinate
- In general, it is compositional: a property of components is applicable to the composition
- There are often unreachable states

Formal Definition

$$A = (States_A, Inputs_A, Outputs_A, update_A, initialState_A)$$

$$B = (States_B, Inputs_B, Outputs_B, update_B, initialState_B)$$

- Synchronous side-by-side:

$$States_C = States_A \times States_B$$

$$Inputs_C = Inputs_A \times Inputs_B$$

$$Outputs_C = Outputs_A \times Outputs_B$$

$$initialState_C = (initialState_A, initialState_B)$$

Formal Definition (Continued)

$$update_C((s_A, s_B), (i_A, i_B)) = ((s'_A, s'_B), (o_A, o_B)),$$

where

$$(s'_A, o_A) = update_A(s_A, i_A),$$

and

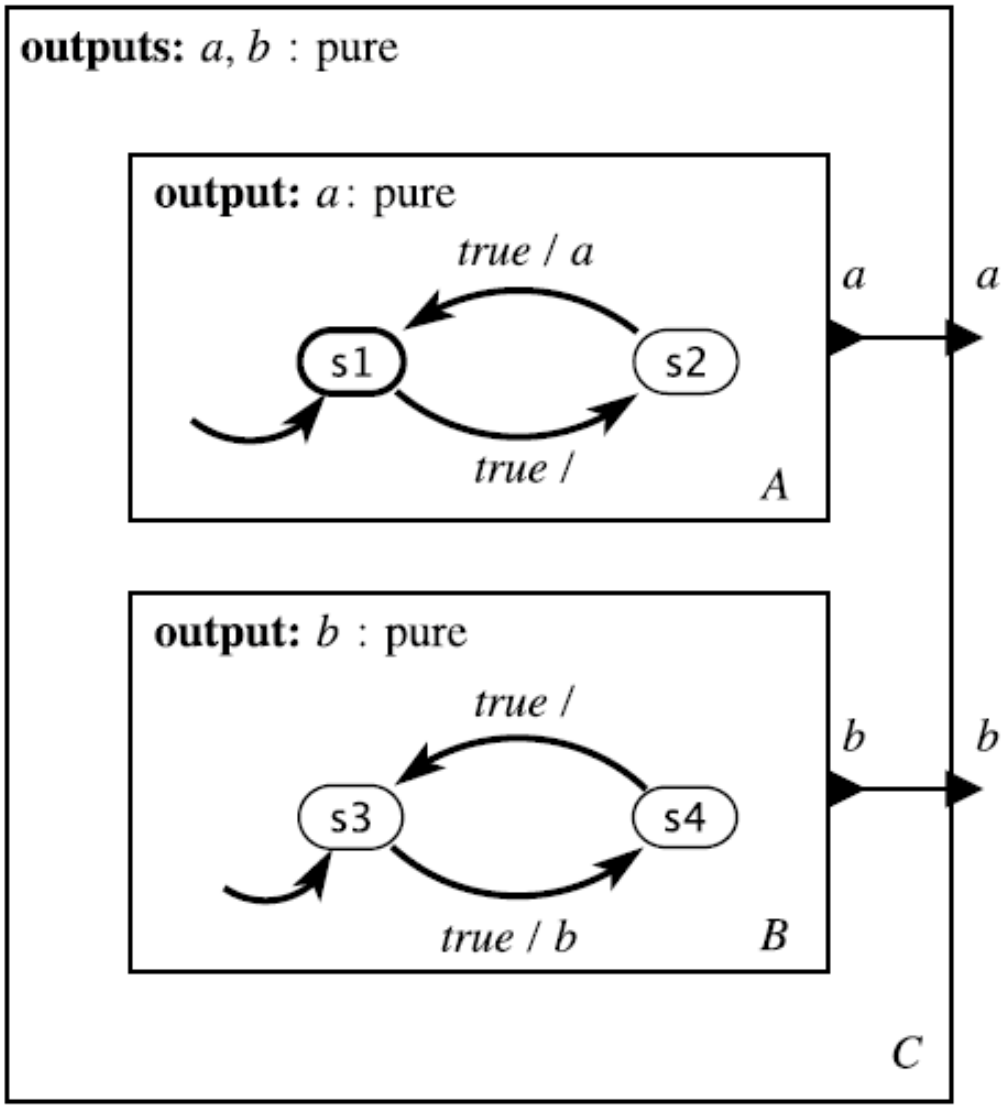
$$(s'_B, o_B) = update_B(s_B, i_B),$$

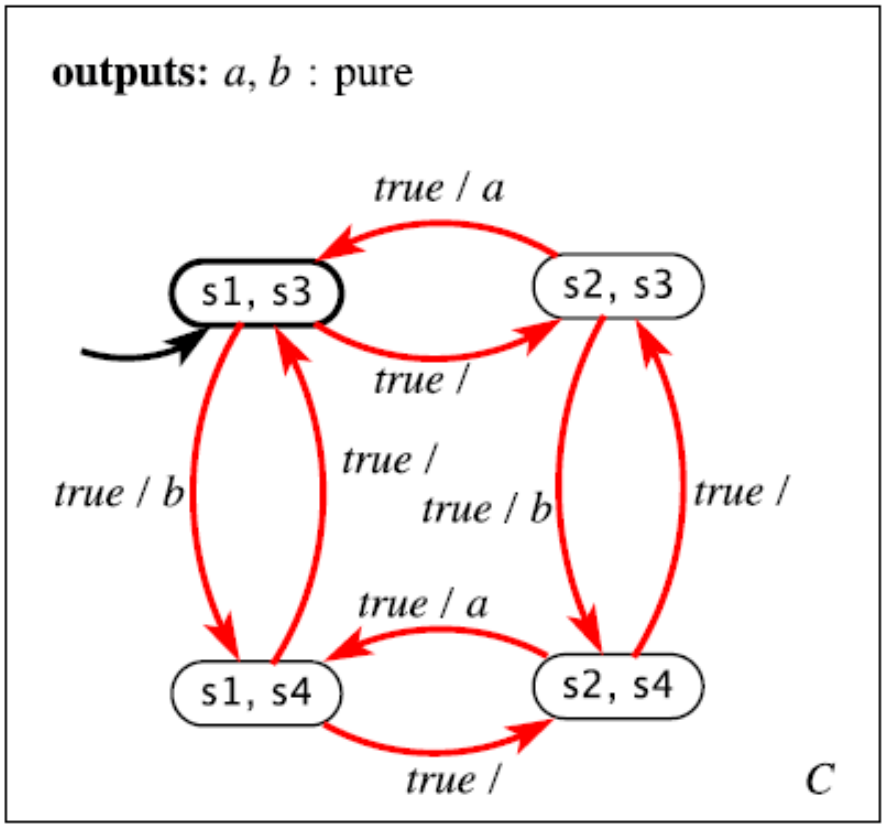
for all $s_A \in States_A$, $s_B \in States_B$, $i_A \in Inputs_A$, and $i_B \in Inputs_B$.

Side-by-side Asynchronous

- The FSM's can react at any point and independent from each other
- Two possible semantics:
  - C reacts when either A or B react (semantics 1, also called interleaving)
  - C reacts when A, or B, or both react (semantics 2)
- In both semantics no prior knowledge exists regarding the order of A or B reaction (nondeterministic)

Example (Semantics 1)





Note that the machine is nondeterministic!

Formal Definition (Semantics 1)

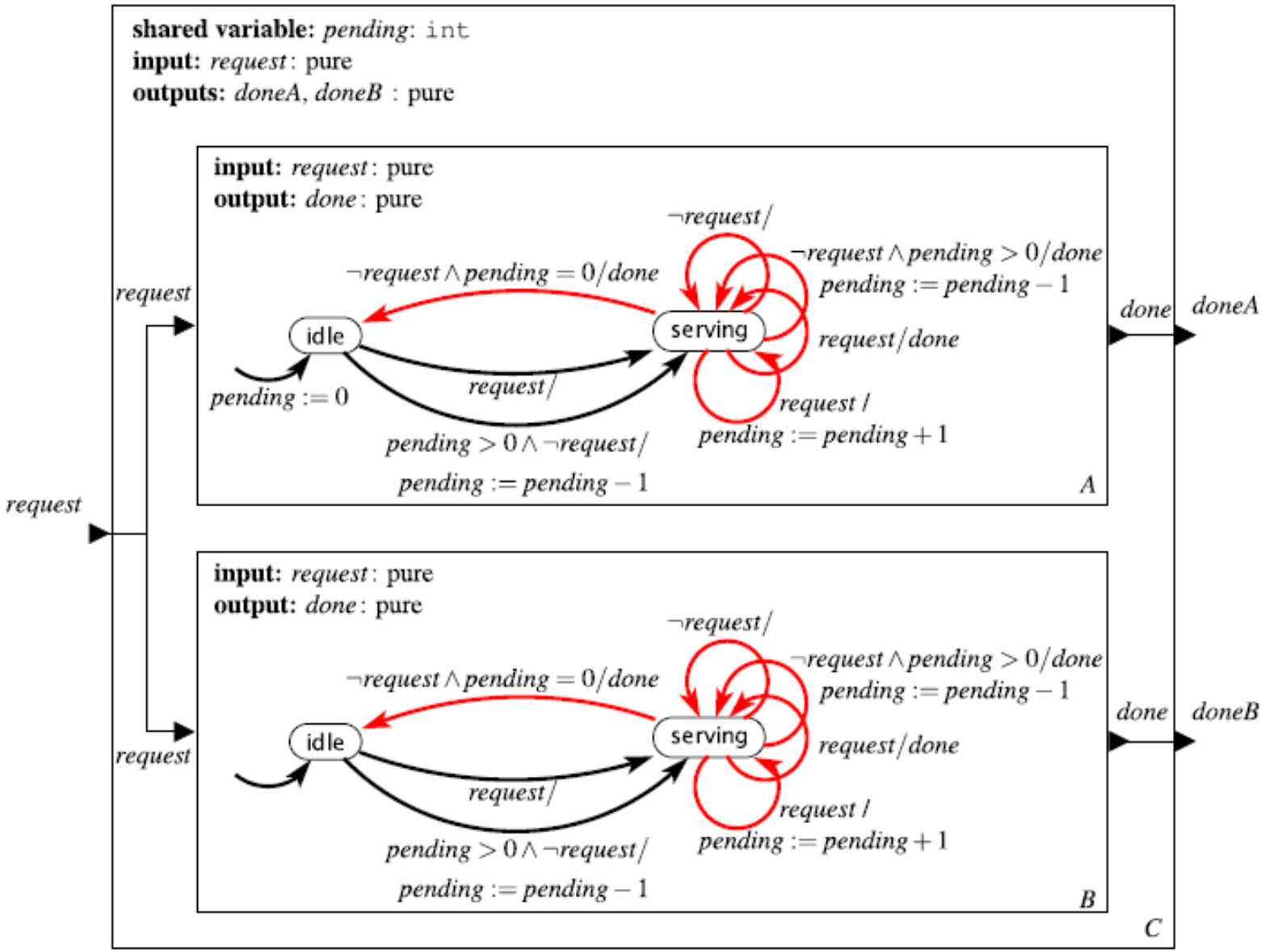
$$update_C((s_A, s_B), (i_A, i_B)) = ((s'_A, s'_B), (o'_A, o'_B)),$$

$$(s'_A, o'_A) = update_A(s_A, i_A) \text{ and } s'_B = s_B \text{ and } o'_B = absent$$

$$(s'_B, o'_B) = update_B(s_B, i_B) \text{ and } s'_A = s_A \text{ and } o'_A = absent$$

for all $s_A \in States_A$, $s_B \in States_B$, $i_A \in Inputs_A$, and $i_B \in Inputs_B$

Example With Shared Variables



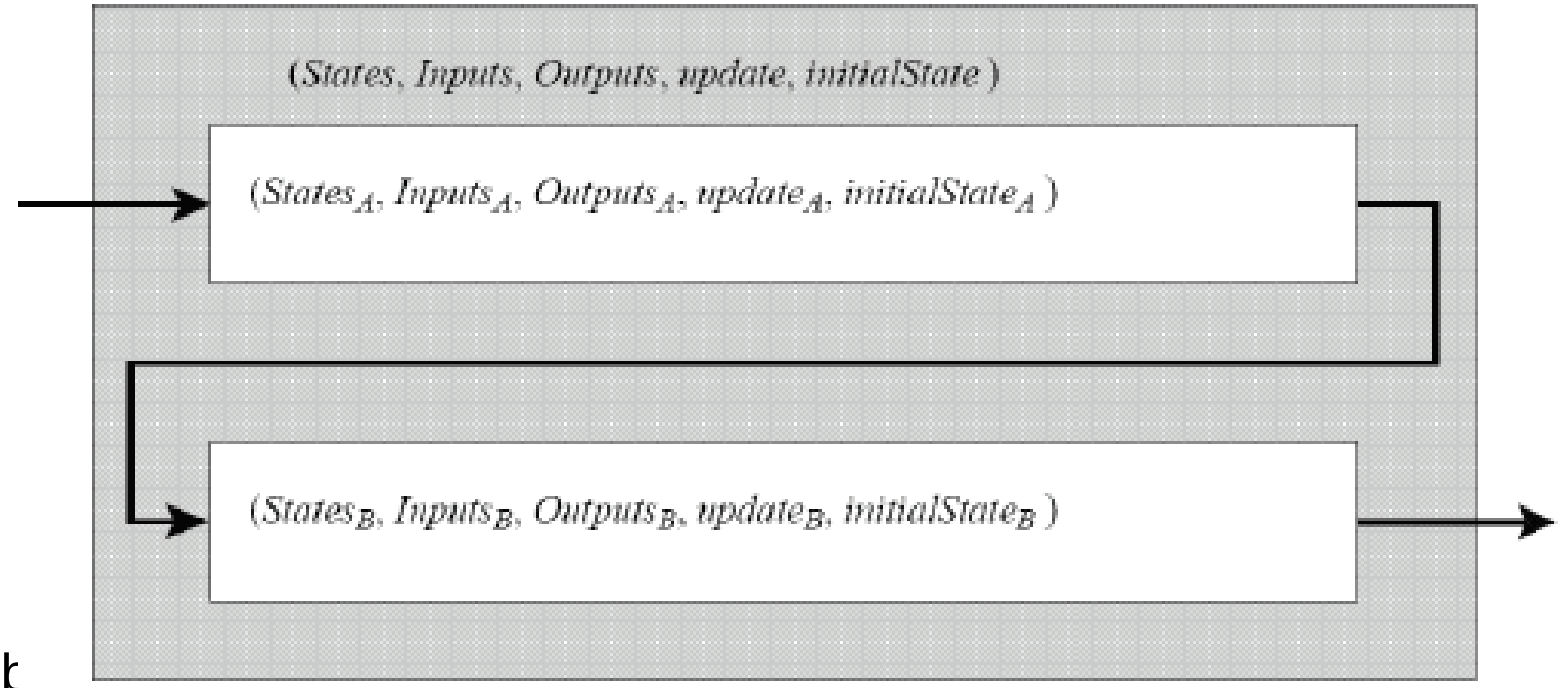
Em

Server Example Points

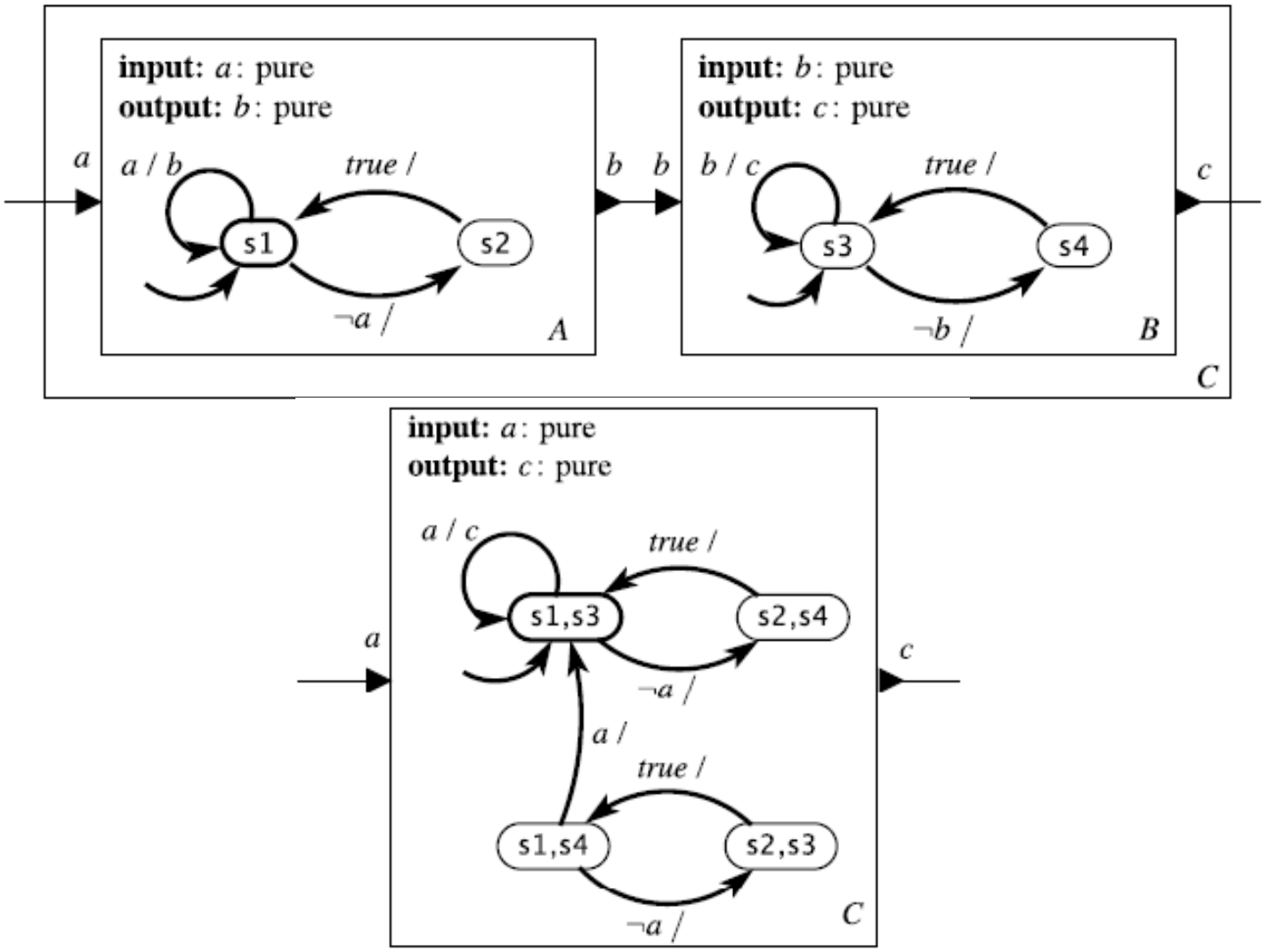
- Shared variables create difficulties:
  - Accesses must be atomic
  - Otherwise system won't work properly
  - May not be always possible
  - Write before read/read before write?
- Semantics 1 is better because the input goes to both machines
  - No input will be missed
- If inputs were independent, semantics 2 would be better

Cascade Composition

- The output of one FSM feeds the input of another FSM
  - Also called serial composition
- It has to type check input/output

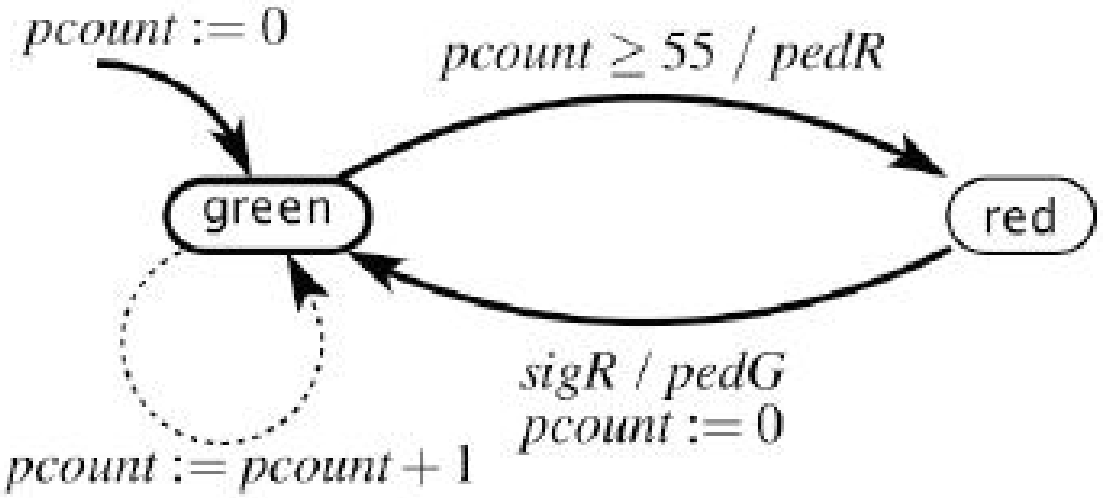


Cascaded Composition Example

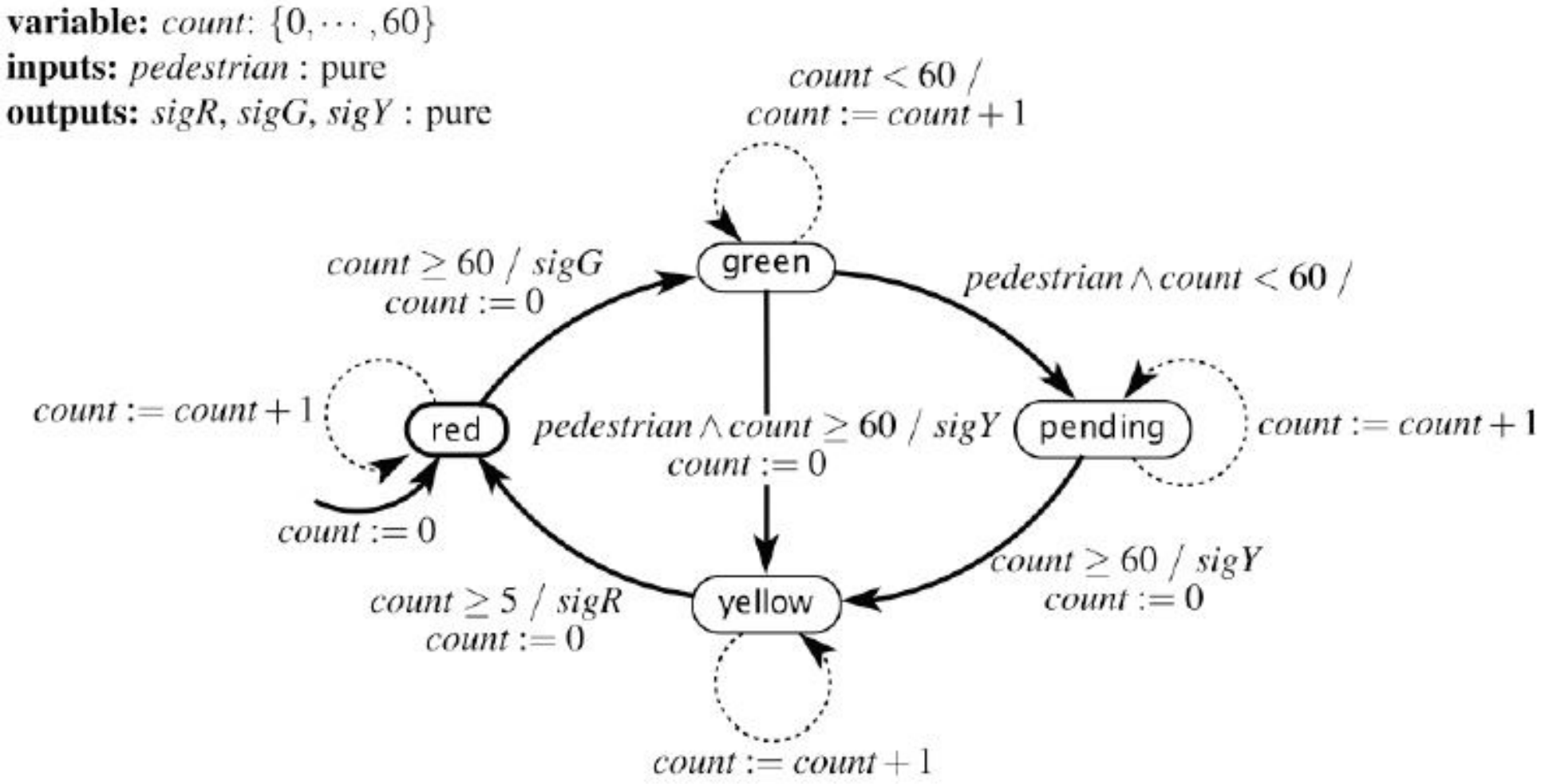


Example (Pedestrian Light FSM 1)

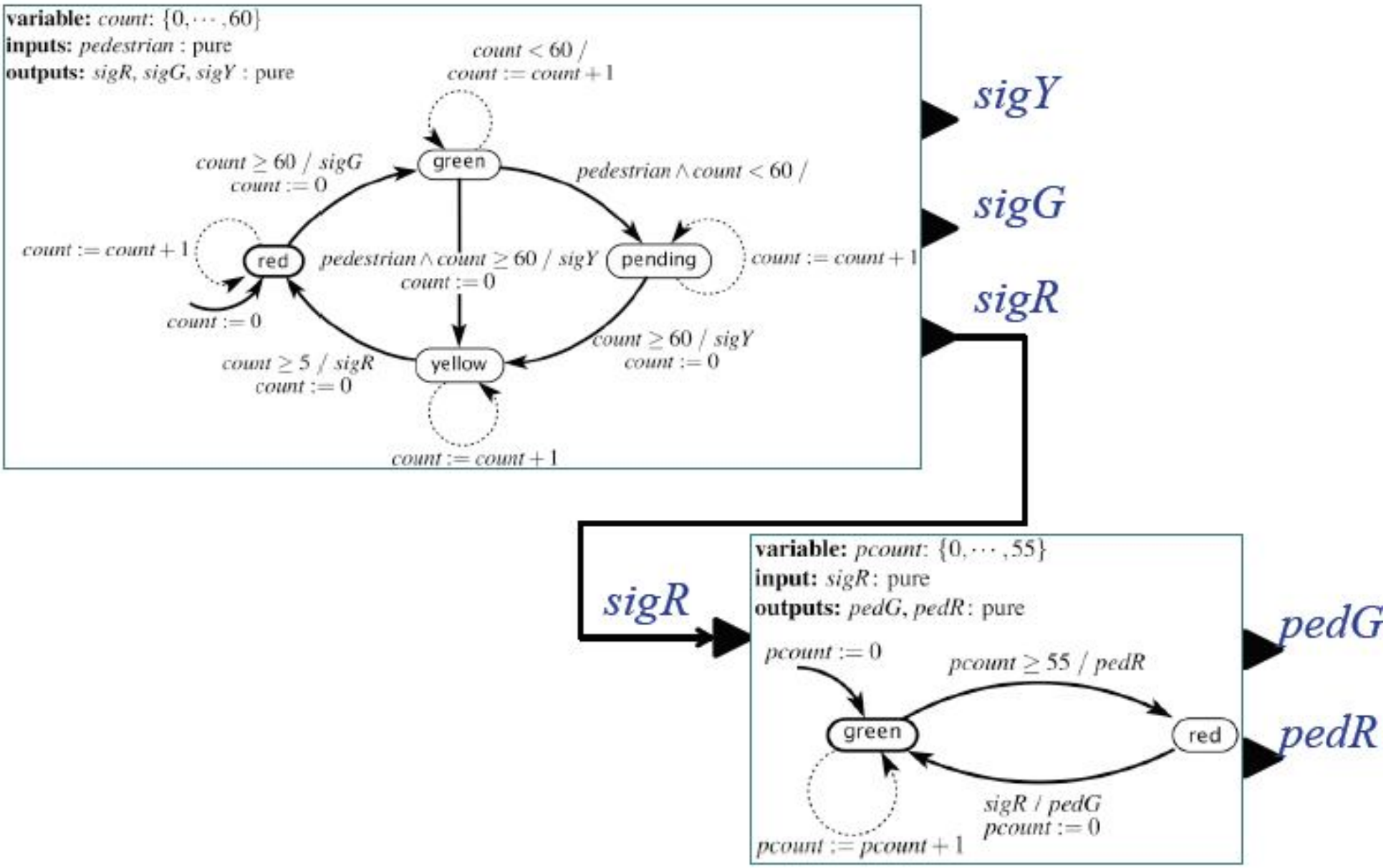
variable: $pcount$: $\{0, \cdots, 55\}$
input: $sigR$: pure
outputs: $pedG$, $pedR$: pure



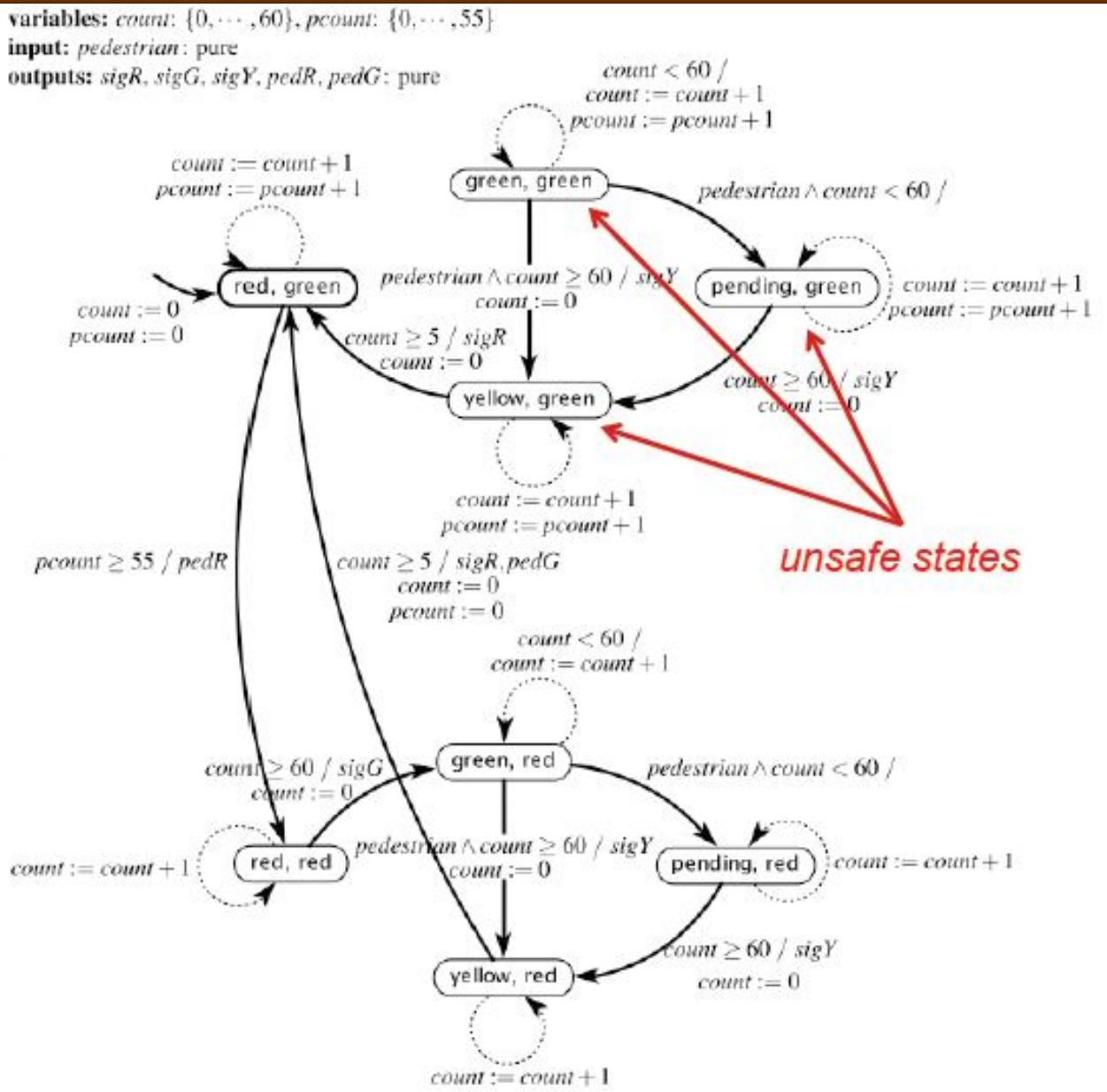
Example (Car Light FSM 2)



Example (Cascaded FSMs)

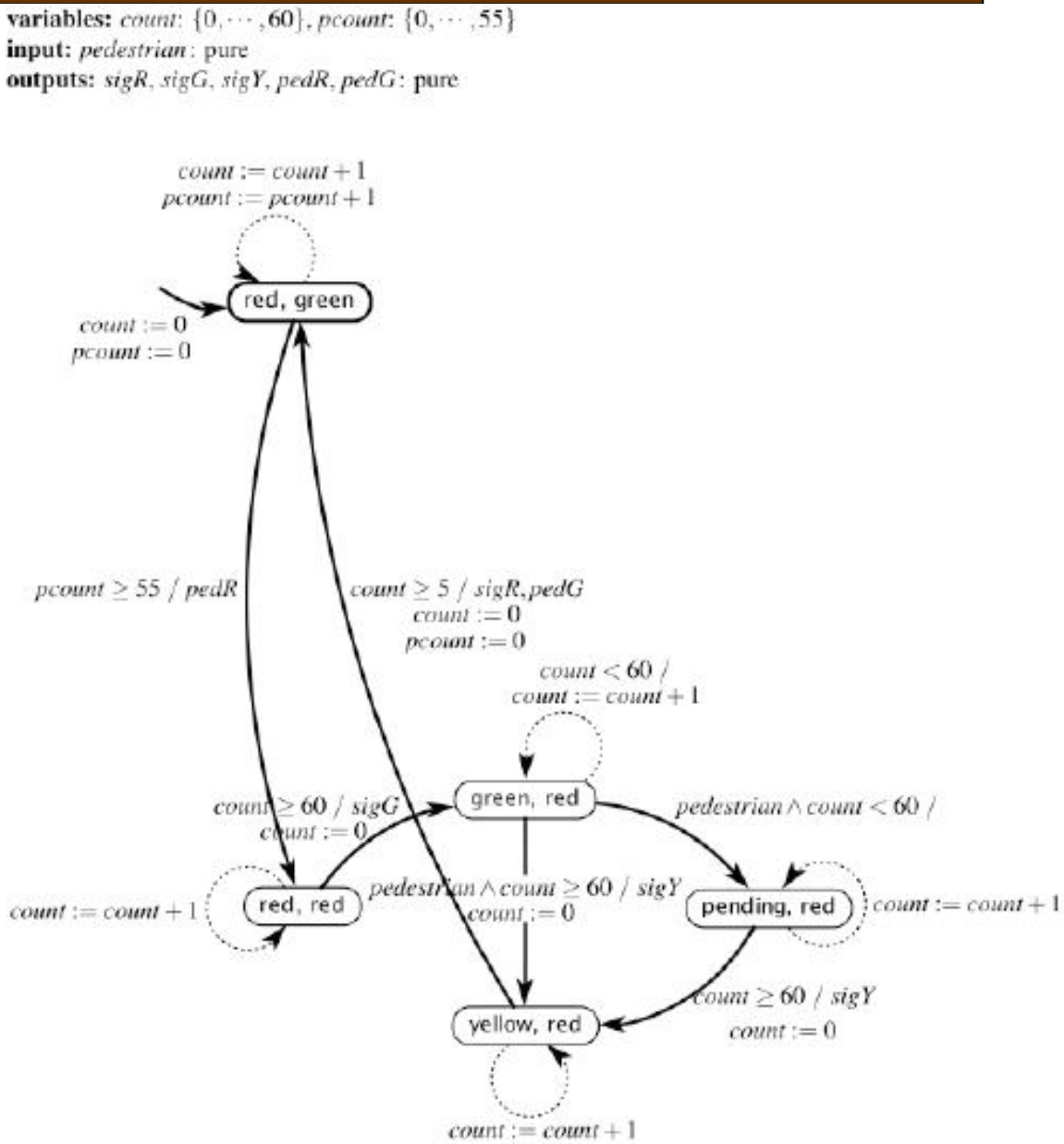


Example: Composition Machine (Assuming Synch. Comp.)



Example: Composition Machine (After Removing Unsafe States)

- Why can the unsafe states be removed?
- B/c they are unreachable
- The semantics allow us to modify the composition machine



Synchronous Cascade Consideration

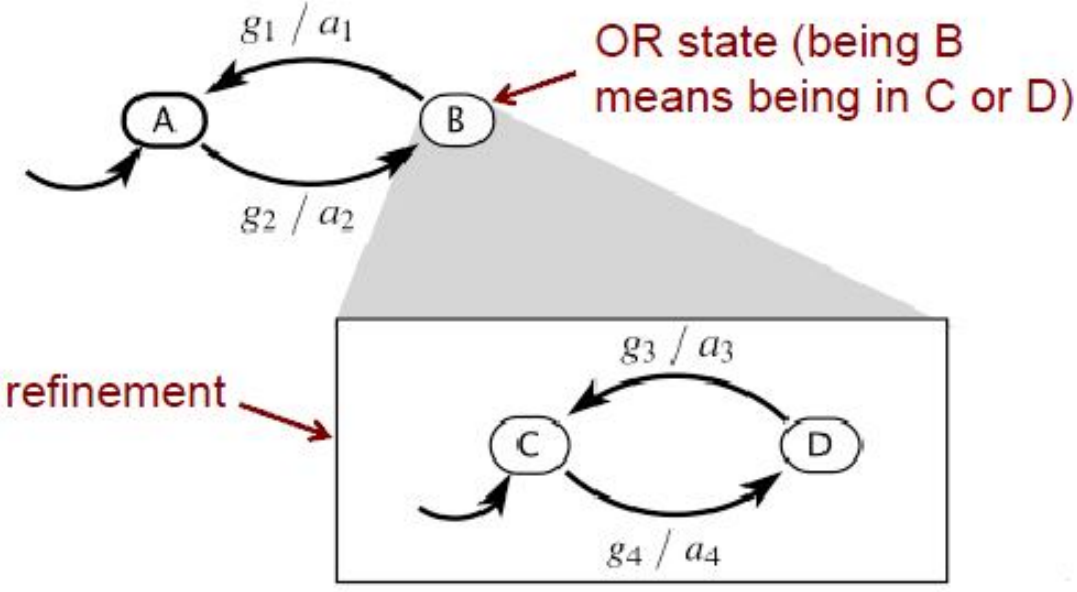
- Two possible setups for cascade composition:
  - Synchronous (like the previous examples)
  - Asynchronous (Chapter 6)
- Synchronous means:
  - Reaction of C means A reacts then B reacts
  - To avoid timing issues, we assume all reactions are instantaneous
  - This also means they are simultaneous
  - This doesn't violate causality, i.e., B still depends on A

Hierarchical State Machines

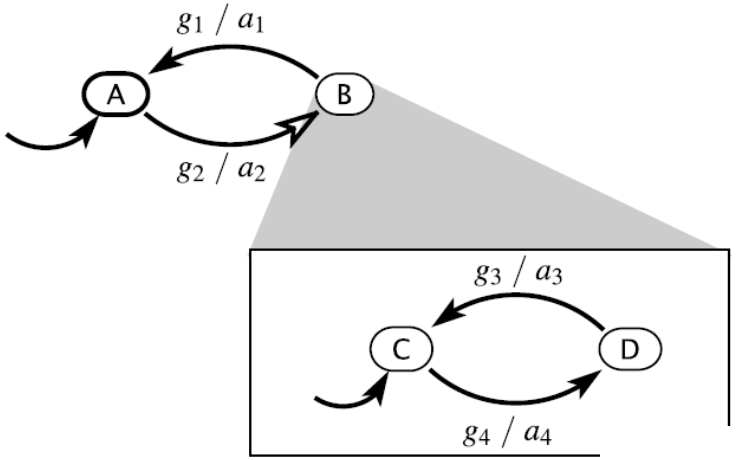
- When one (or more) state of an FSM is itself another FSM
  - The inner FSM is called a “refinement”
- There are different semantics for the composition FSM:
  - What happens if two reactions are possible in the refinement and the top level?
    - Priority given to the refinement (depth-first)
    - Priority given to the top level state (preemptive)
  - The outputs might be produced at both levels

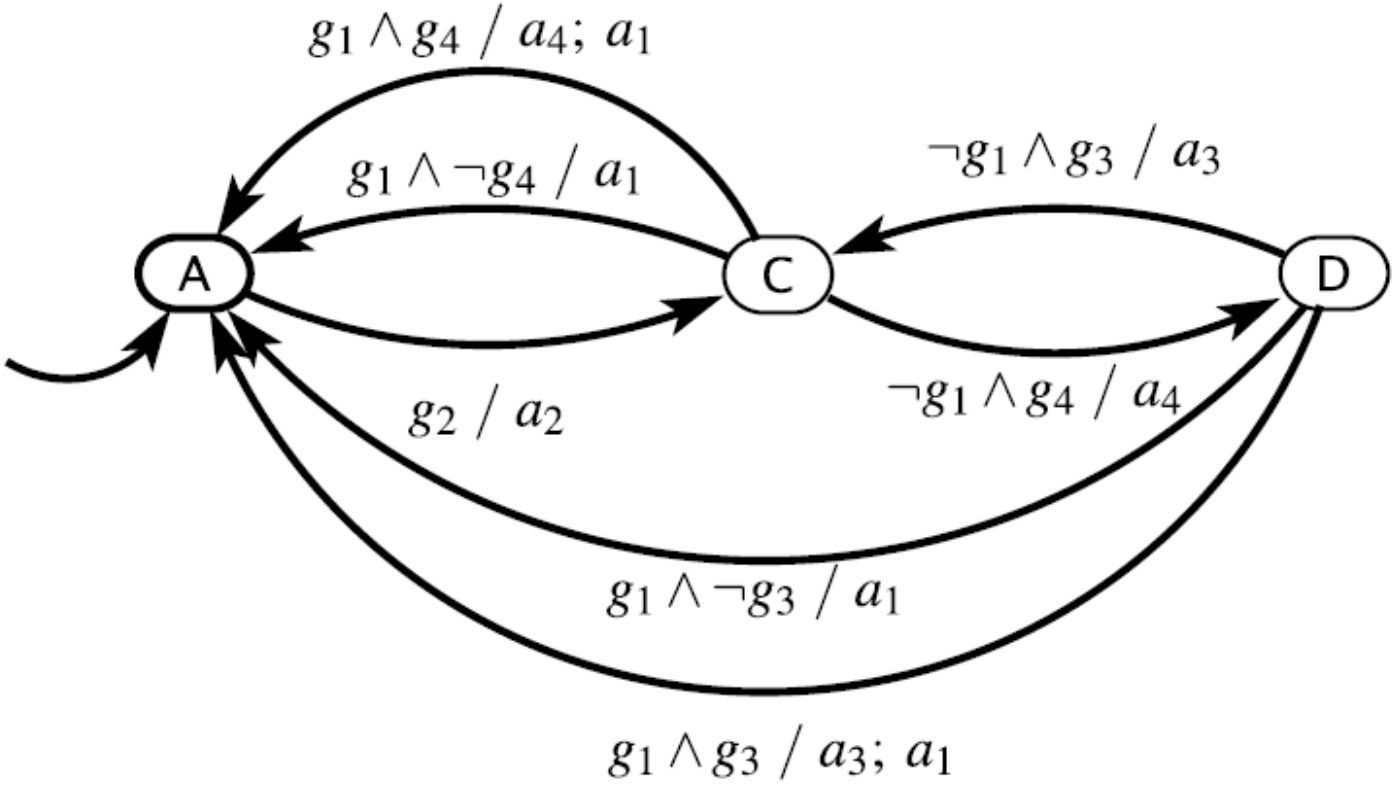
Hierarchical State Machines Reaction

- The reactions are still simultaneous and instantaneous
- Note that this composition allows for the OR operation
- If two outputs are produced, they are required not to have a conflict:
- Define sequence (shown by ;)



Depth-First Semantics

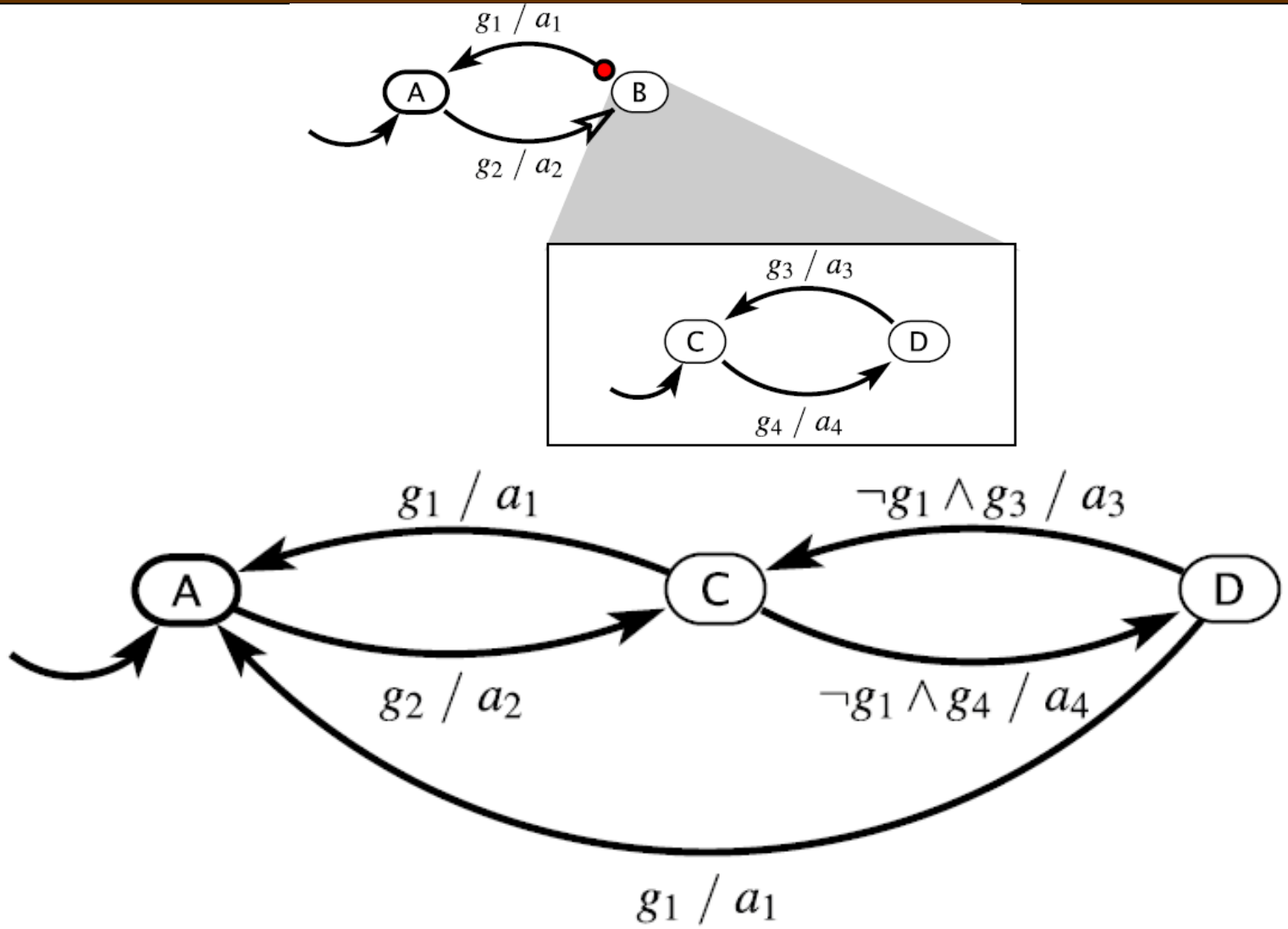




Preemptive Transitions

- Another way to ensure the outputs won't conflict: preemptive transitions
- The guards of the preemptive reaction (at the top level) are evaluated before the refinement reacts
- If true, the refinement will NOT react
- Syntax? Use a red circle to specify a preemptive transition

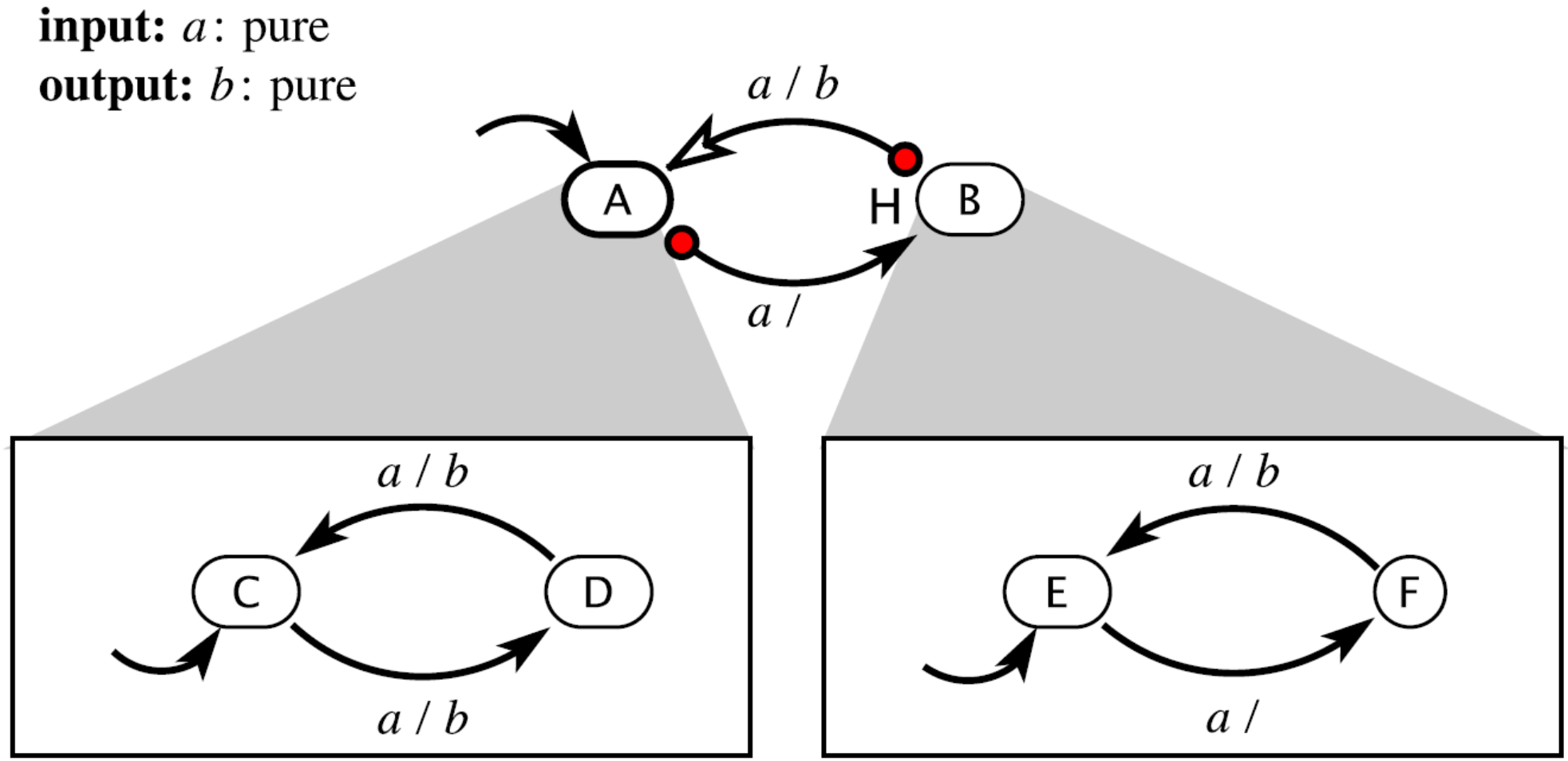
Preemptive Transitions Example



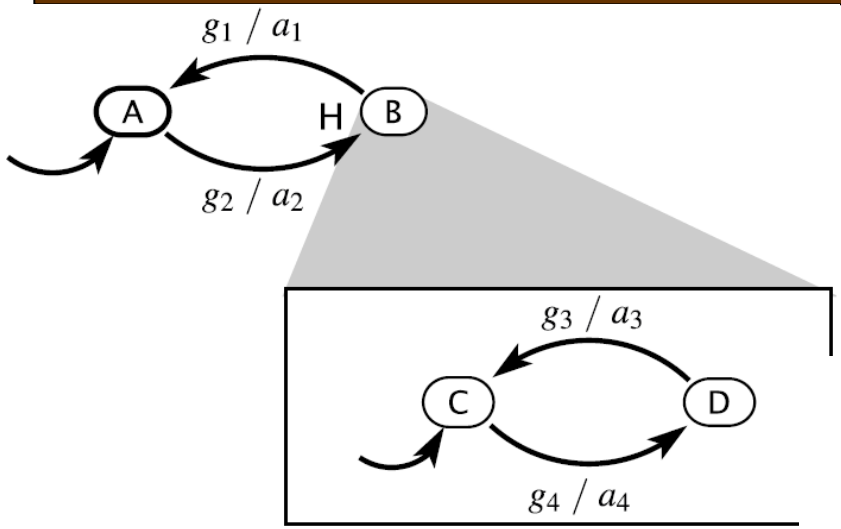
Reset Transition

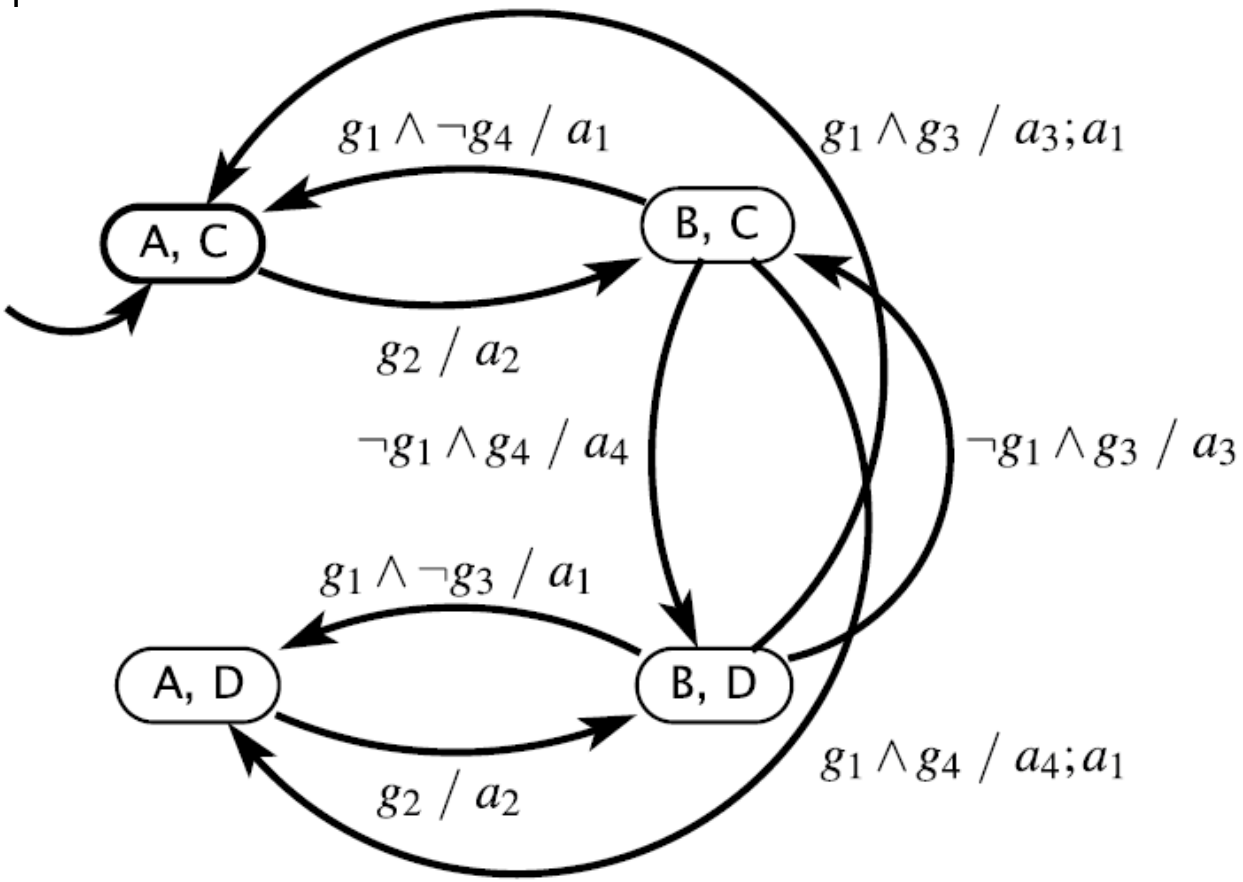
- When entering a refinement for the first time, always go to the initial state
- What to do when we reenter a refinement?
- Two answers:
  1. Always go to the initial state (reset transition)
     - Reset transition is represented by a hollow arrowhead (as shown in previous example)
  2. Refinement resumes in whatever state it was last in (history transition)
     - Represented by a solid arrowhead

Notation Example



History Transition Example





A Real Example

- Consider a program that does something for 2 seconds and then stops
- One possible implementation is shown on the right

```
volatile uint timerCount = 0;
void ISR(void) {
  … disable interrupts
  if(timerCount != 0) {
    timerCount--;
  }
  … enable interrupts
}
int main(void) {
  // initialization code
  SysTickIntRegister(&ISR);
  ... // other init
  timerCount = 2000;
  while(timerCount != 0) {
    ... code to run for 2 seconds
  }
}
```

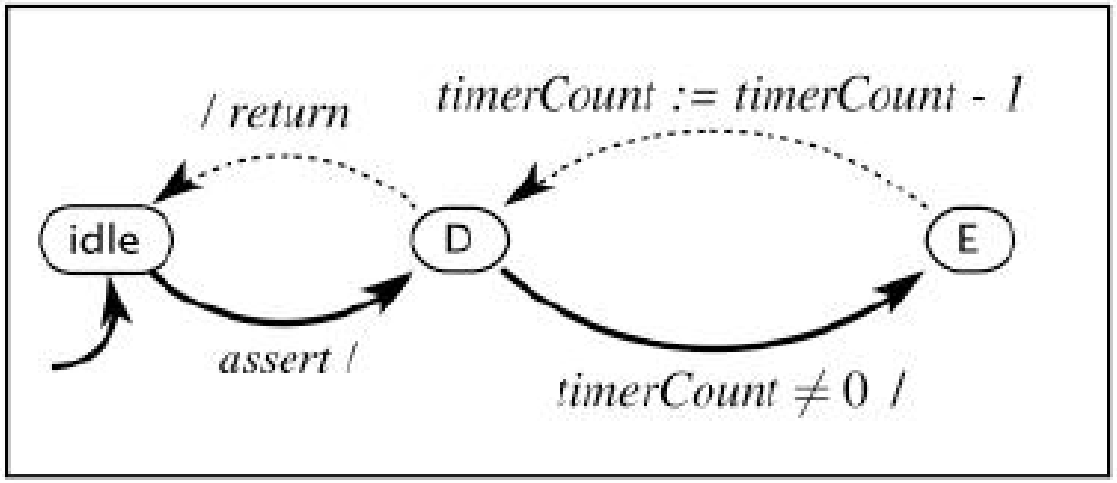
Example Implementation

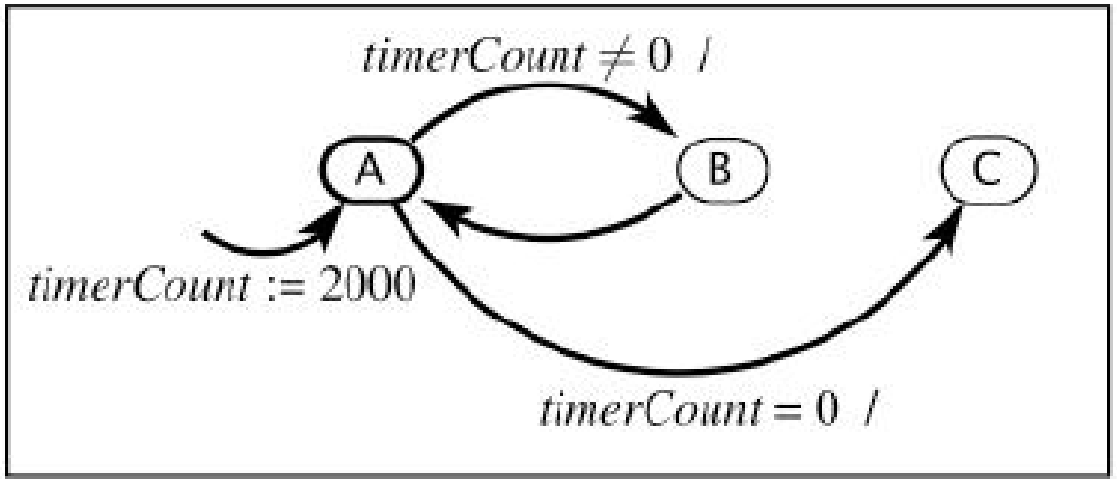
- What composition is most suitable?
- Let's name the different states, note that position in the program is part of the state
- Next: draw the state diagram of each component

```
volatile uint timerCount = 0;
void ISR(void) {
    … disable interrupts
D → if(timerCount != 0) {
E →     timerCount--;
    }
    … enable interrupts
}
int main(void) {
    // initialization code
    SysTickIntRegister(&ISR);
    … // other init
    timerCount = 2000;
A → while(timerCount != 0) {
B →  … code to run for 2 seconds
    }
C → … whatever comes next
}
```

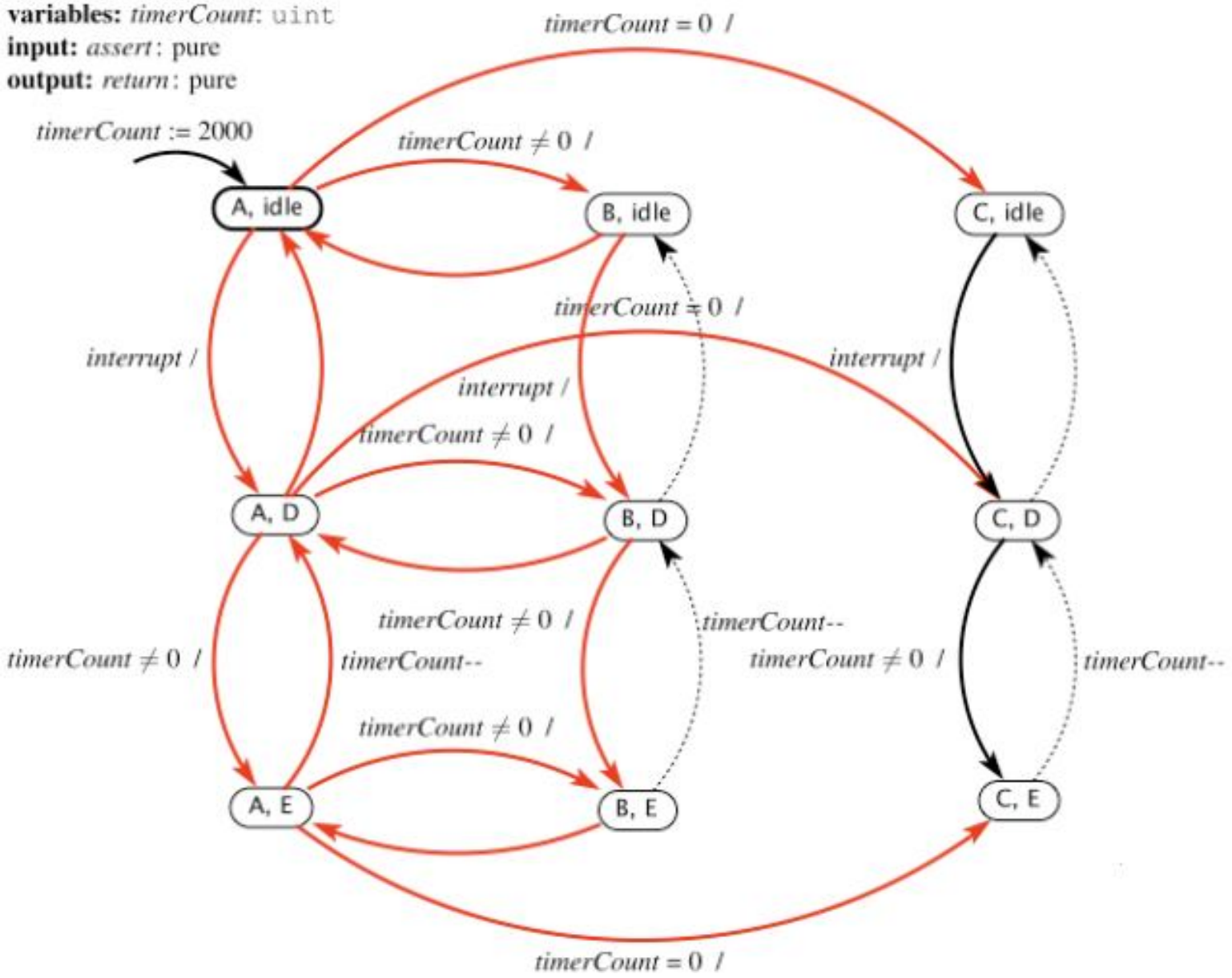
State Diagrams

variables: *timerCount*: `uint`
input: *assert* : pure
output: *return* : pure





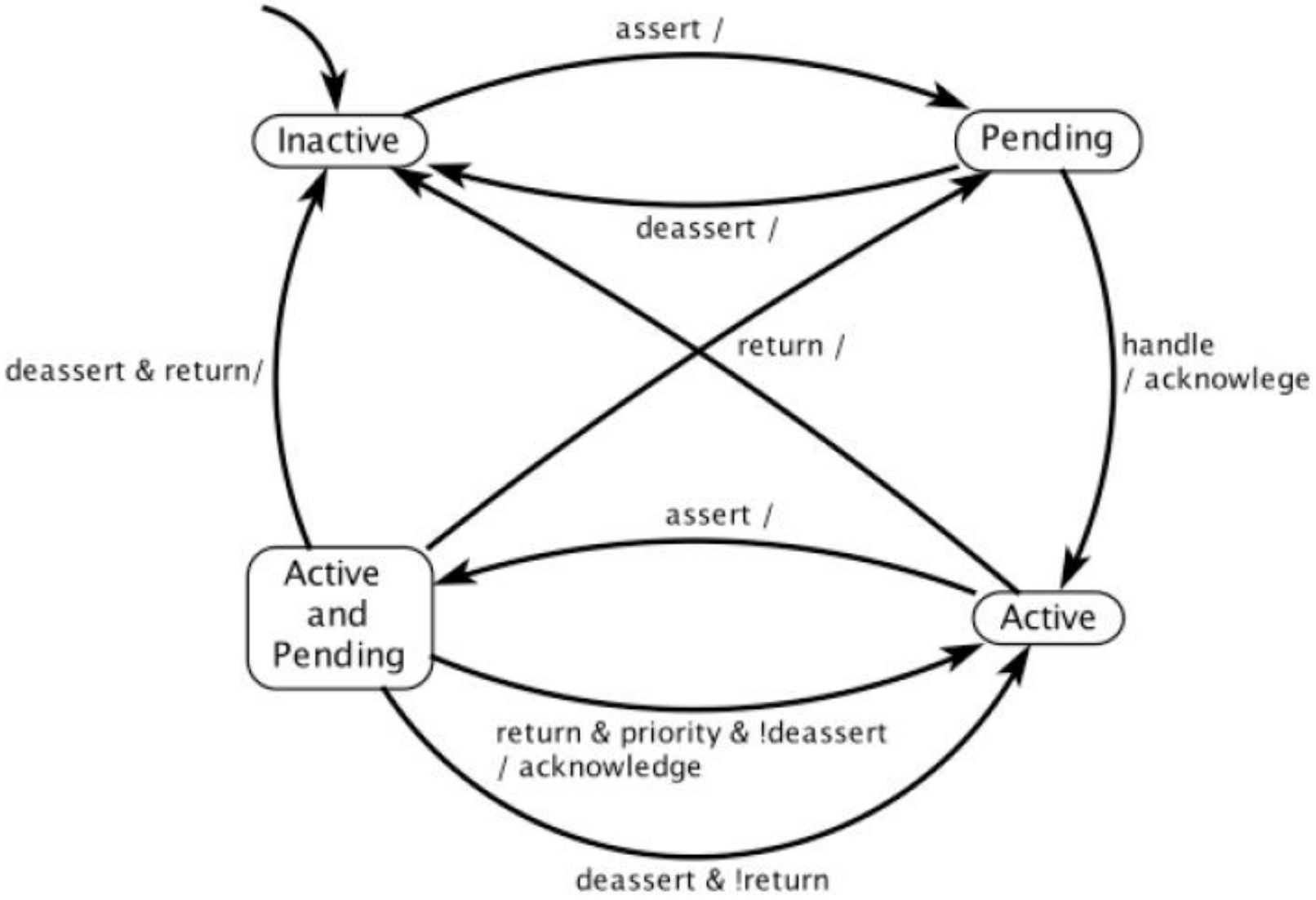
Concurrent Composition?



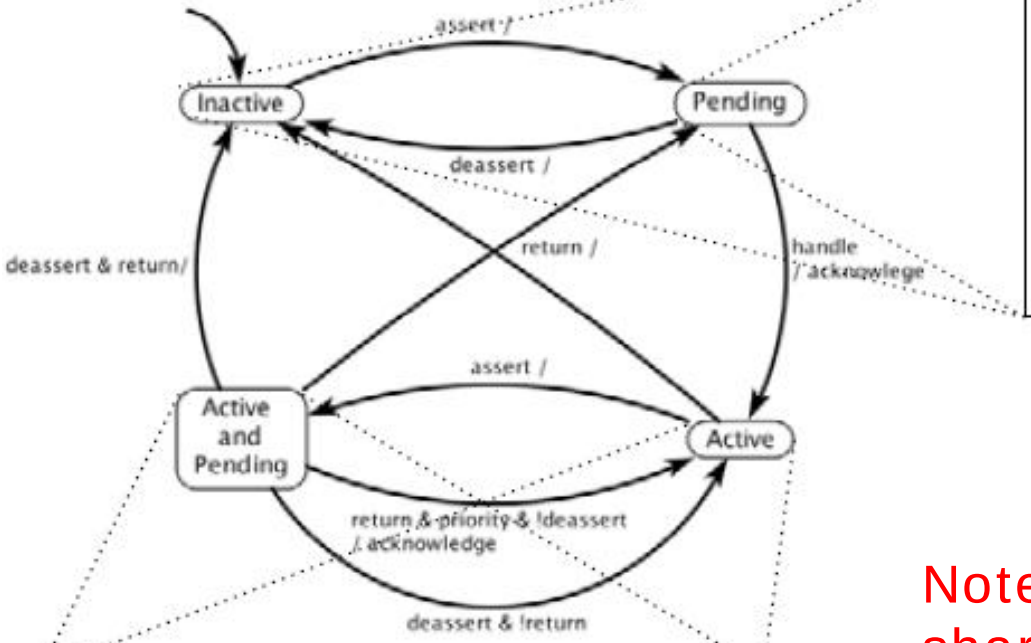
Observations

- Concurrent composition (synchronous or asynchronous) is NOT suitable here b/c:
- There are transitions that will not occur in practice (such as A,D to B,D)
- Since interrupts have priority over application code, concurrent compositions are not the right choice here
- Other compositions?

FSM of an Interrupt Controller



FSM of Our Example



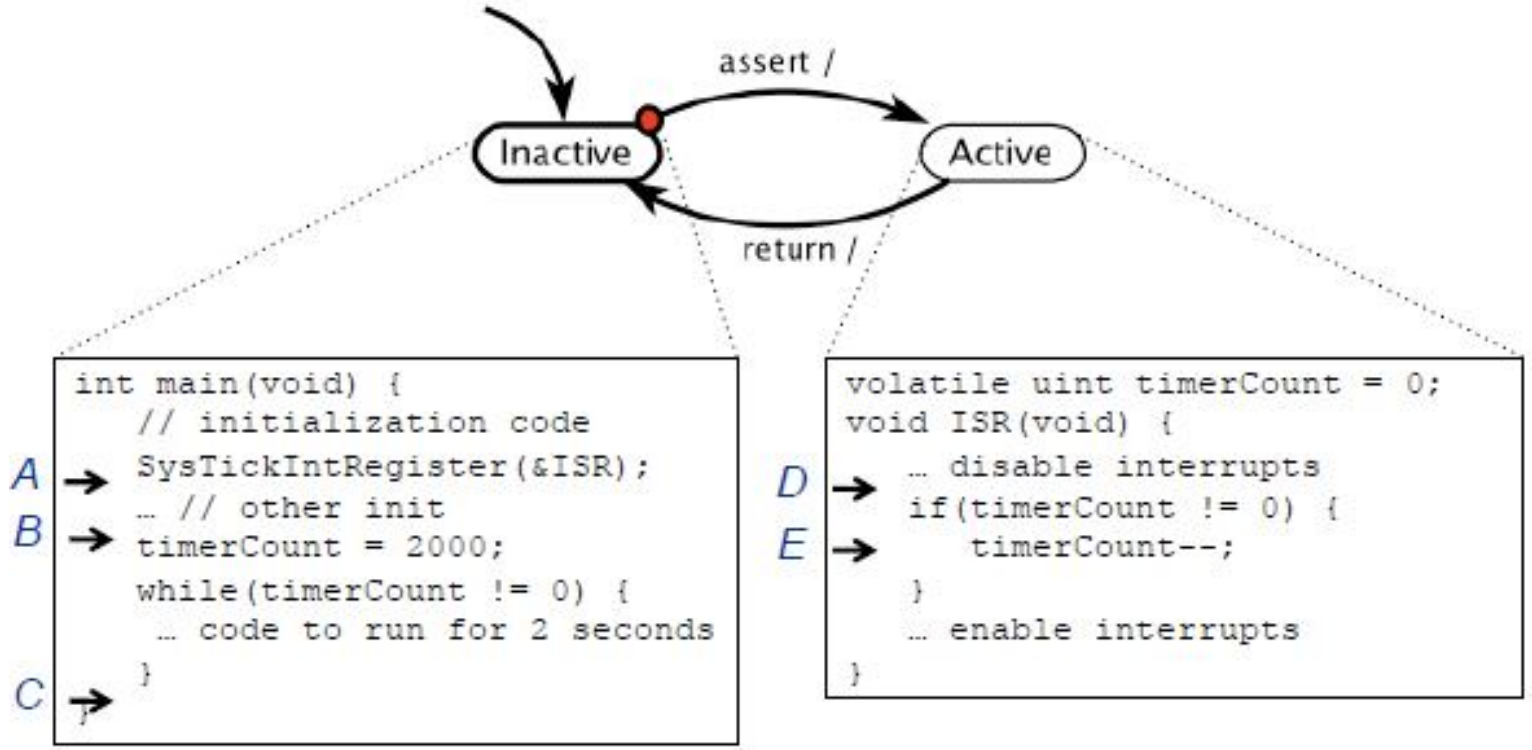

```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

```
volatile uint timerCount = 0;
void ISR(void) {
   … disable interrupts
   if(timerCount != 0) {
      timerCount--;
   }
   … enable interrupts
}
```

Note that the states are sharing their refinements

Composition Using Preemptive Transitions

- Note that this abstraction assumes that an interrupt is always handled immediately upon being asserted



Interrupt Handling Observations

- History transition results in product state space
- Hierarchy reduces the number of transitions compared to synchronous and asynchronous compositions

Homework Assignments

- Chapter 5 homeworks: 1, 2, 3, 5 (required)
- The rest: optional
- For Tuesday 1403/2/11