Department of Computer Science
National Tsing Hua University

CS 5244: Introduction to Cyber Physical Systems

Unit 20: Hierarchical State Machines

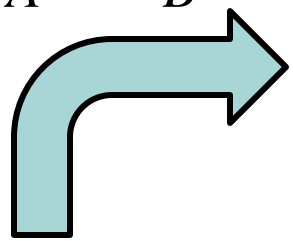
(Ch. 5)

Instructor: Cheng-Hsin Hsu

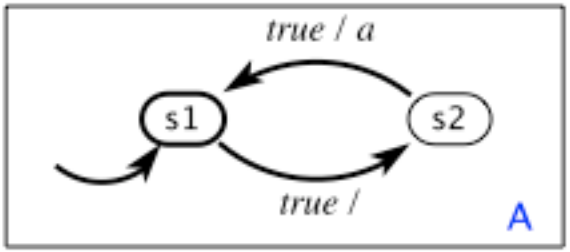
Acknowledgement: The instructor thanks Profs. Edward A. Lee & Sanjit A. Seshia at UC Berkeley for sharing their course materials

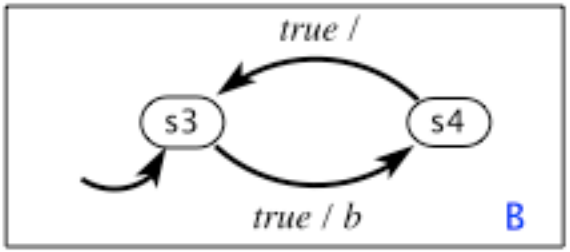
Recall Synchronous Composition:

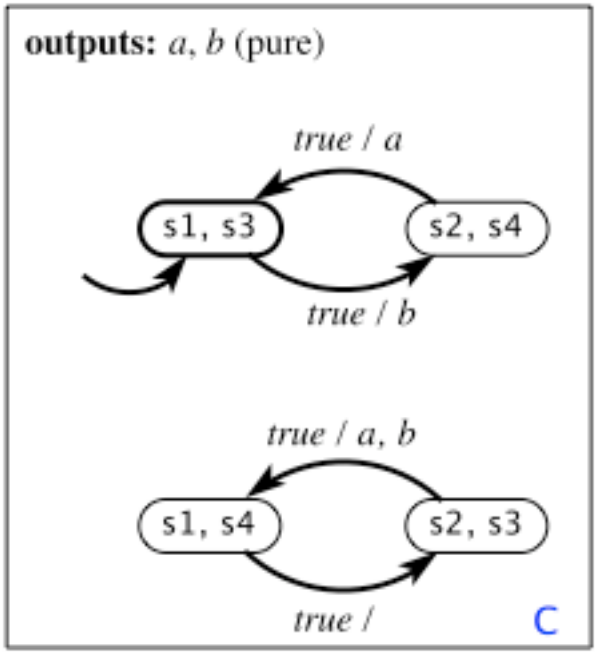
$$S_C = S_A \times S_B$$



outputs: *a*, *b* (pure)



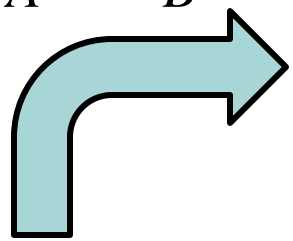




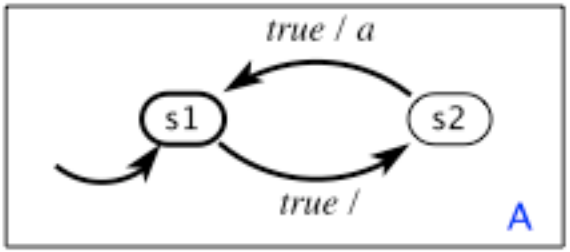
Synchronous composition

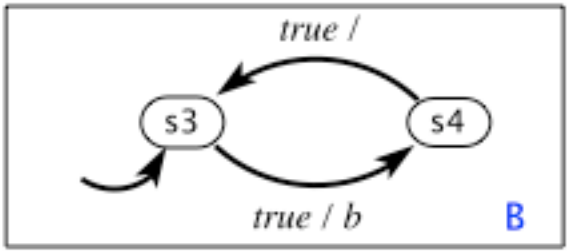
Recall Asynchronous Composition:

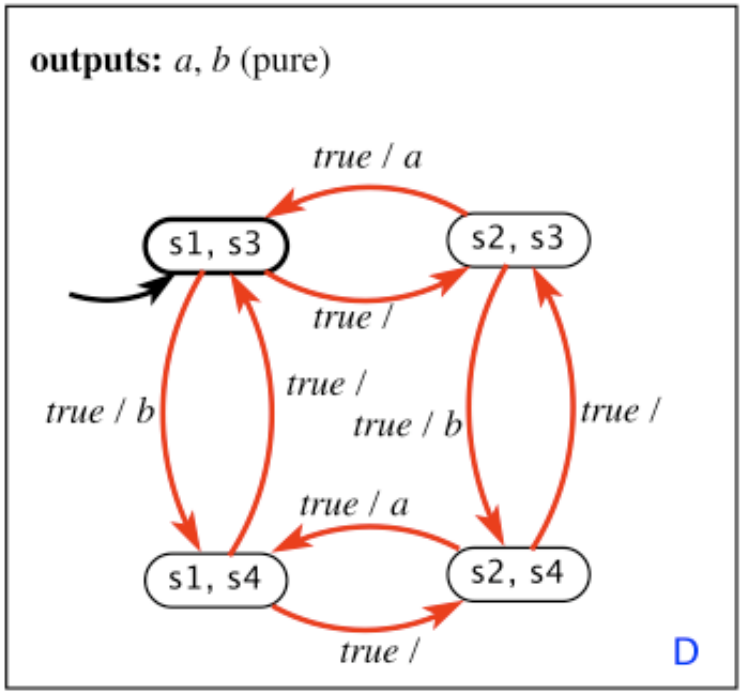
$$S_C = S_A \times S_B$$



outputs: *a*, *b* (pure)







Asynchronous composition with interleaving semantics

Recall program that does something for 2 seconds, then stops

```
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
  }
}
```

Is synchronous composition the right model for this?

Is asynchronous composition (with interleaving semantics) the right model for this?

Answer: no to both.

Position in the program is part of the state

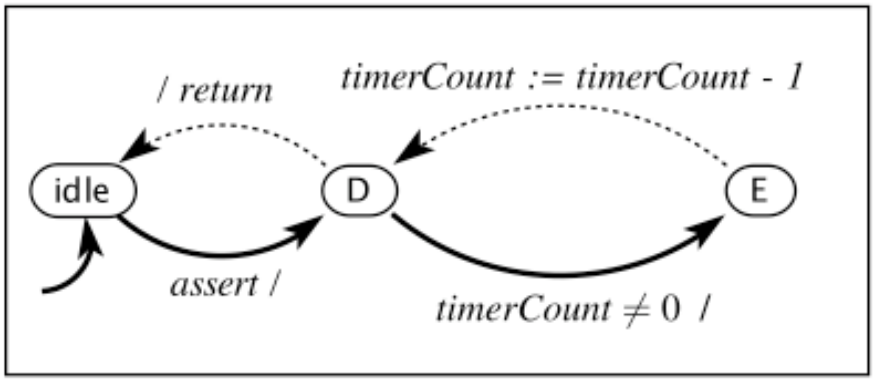
```
volatile uint timerCount = 0;
void ISR(void) {
    … disable interrupts
D → if(timerCount != 0) {
E →     timerCount--;
    }
    … enable interrupts
}
int main(void) {
    // initialization code
    SysTickIntRegister(&ISR);
    … // other init
    timerCount = 2000;
A → while(timerCount != 0) {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

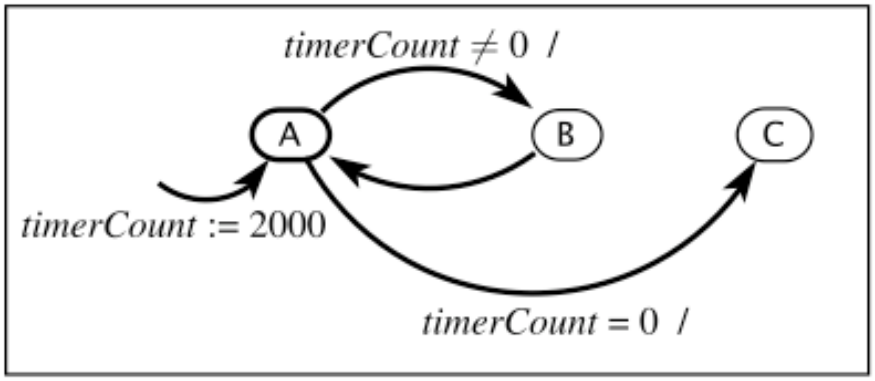
A key question: Assuming interrupt occurs infinitely often, is position C always reached?

State machine model

```
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
}
```

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

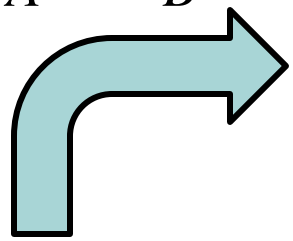




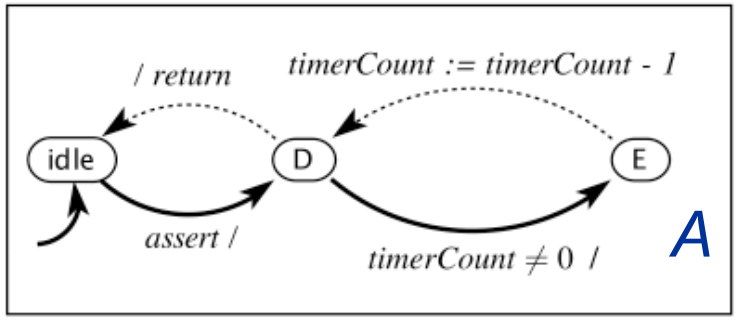
Is asynchronous composition the right thing to do here?

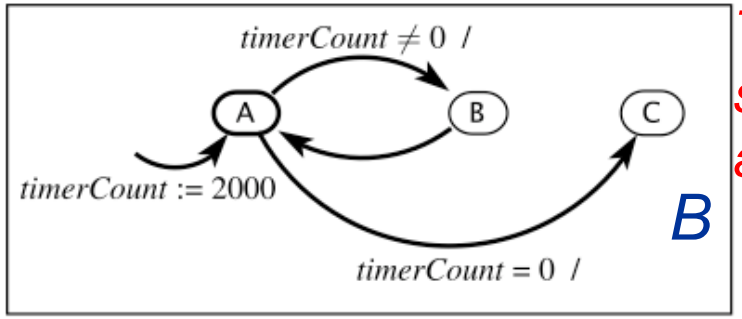
Asynchronous composition

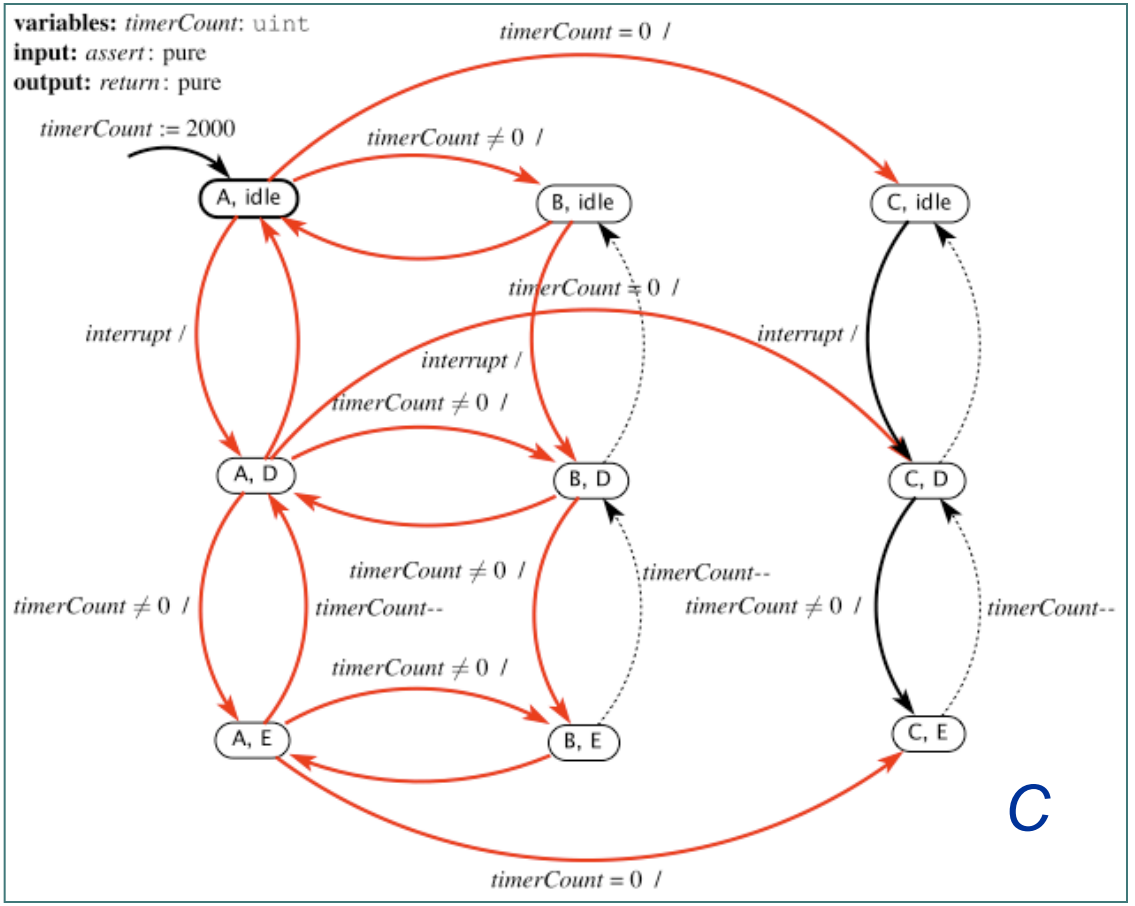
$$S_C = S_A \times S_B$$



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



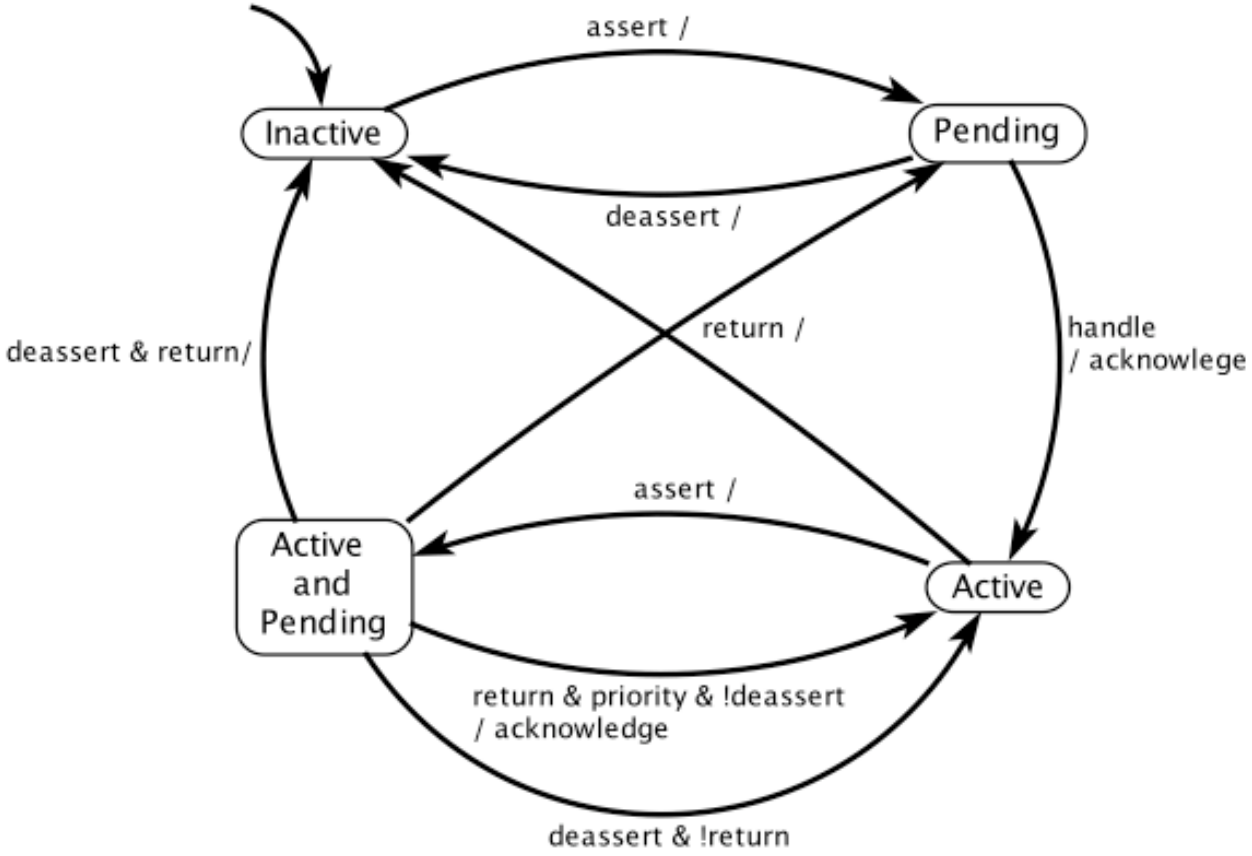




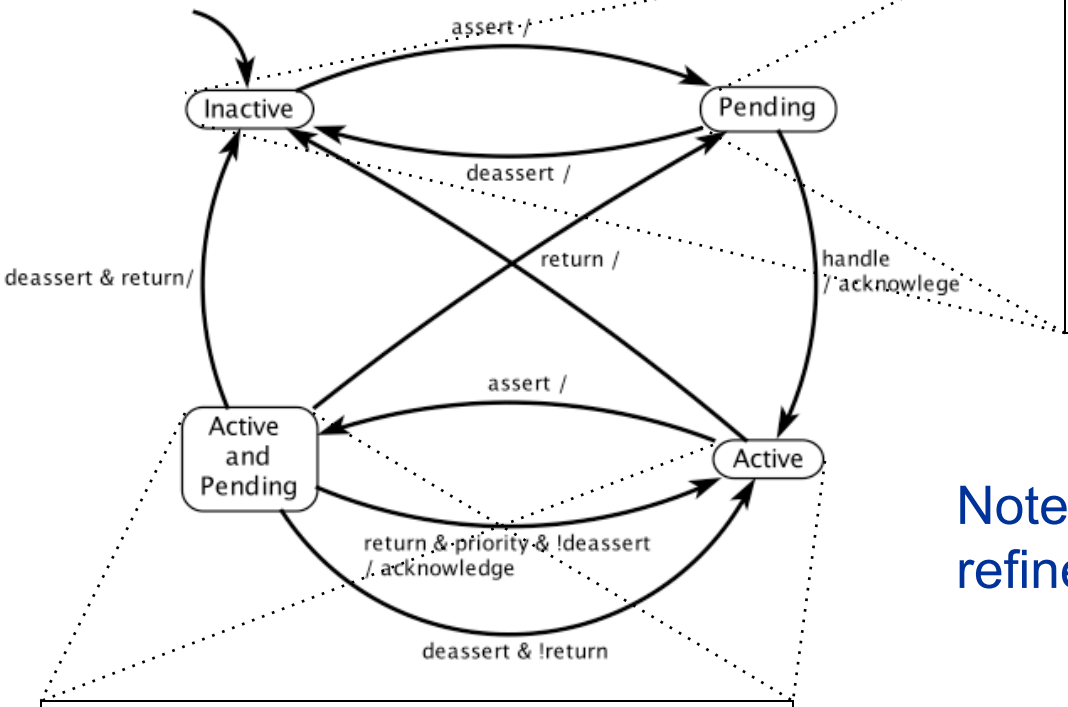
This has transitions that will not occur in practice, such as A,D to B,D. Interrupts have priority over application code.

Modeling an interrupt controller

FSM model of a single interrupt handler in an interrupt controller:



Modeling an interrupt controller

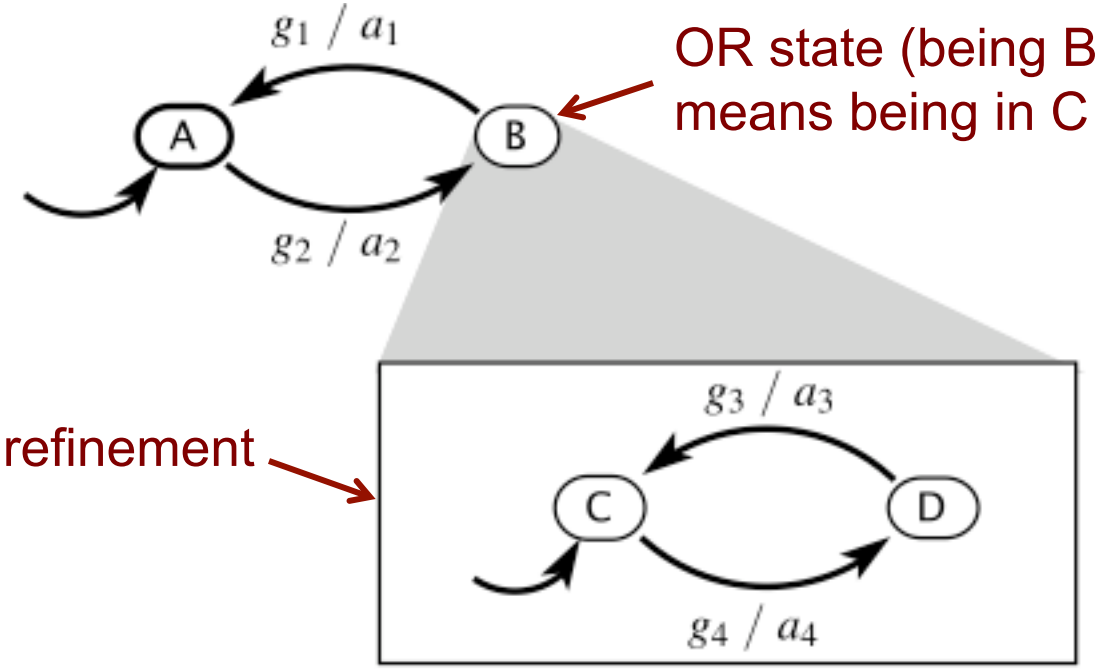



```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

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

Note that states can share refinements.

Hierarchical State Machines



OR state (being B means being in C or D)

refinement

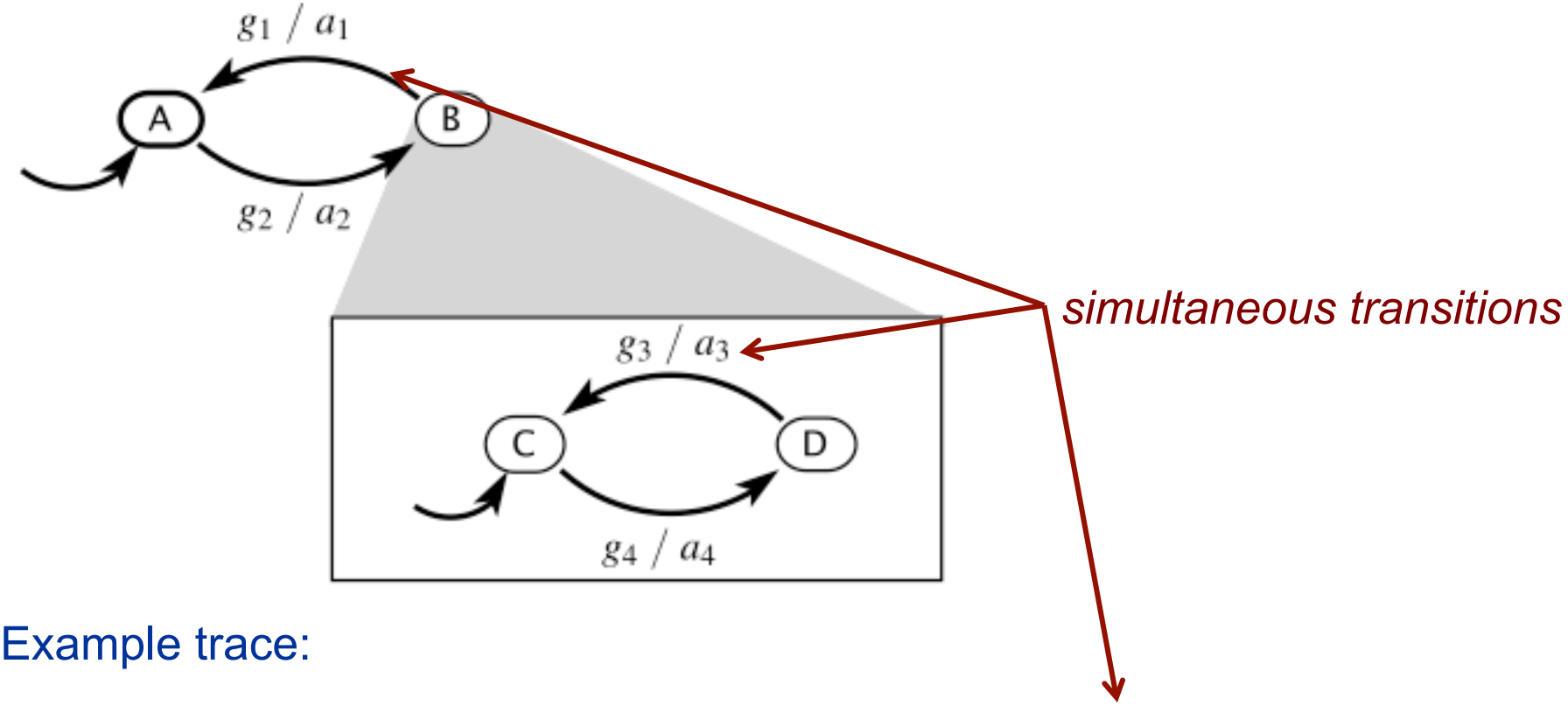
Reaction:

1. First, the refinement of the current state (if any) reacts.
2. Then the top-level machine reacts.

If both produce outputs, they are required to not conflict.

The two steps are part of the same reaction.

Hierarchical State Machines

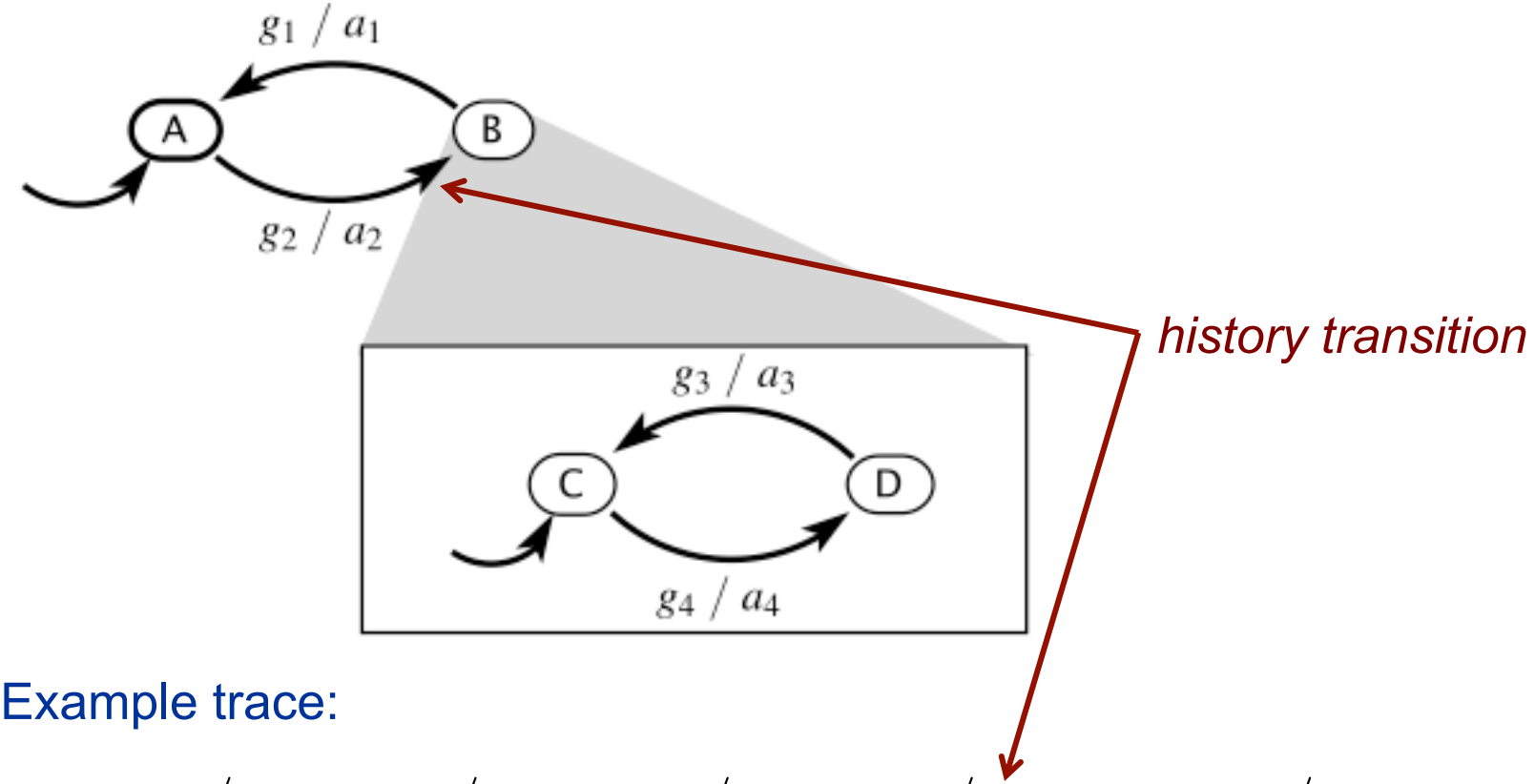


Example trace:

$$A \xrightarrow{g_2/a_2} C \xrightarrow{g_4/a_4} D \xrightarrow{g_1/a_1} A \xrightarrow{g_2/a_2} D \xrightarrow{g_3 \wedge g_1 / a_3, a_1} A \cdots$$

Simultaneous transitions can produce multiple outputs. These are required to not conflict.

Hierarchical State Machines

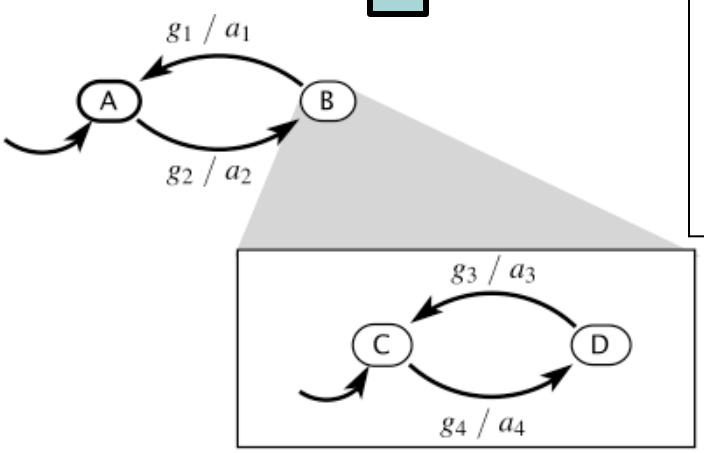


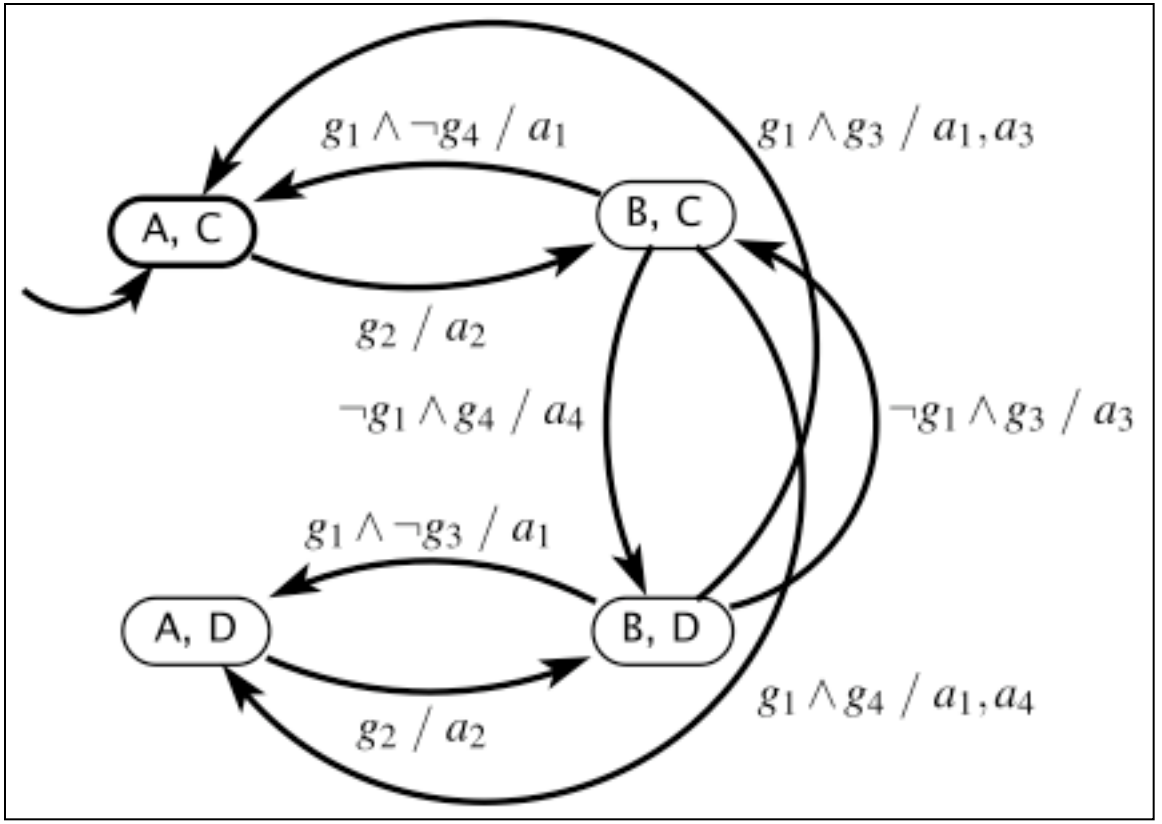
Example trace:

$$A \xrightarrow{g_2/a_2} C \xrightarrow{g_4/a_4} D \xrightarrow{g_1/a_1} A \xrightarrow{g_2/a_2} D \xrightarrow{g_3 \wedge g_1/a_3,a_1} A \cdots$$

A history transition implies that when a state with a refinement is left, it is nonetheless necessary to remember the state of the refinement.

Flattening the state machine (assuming history transitions):

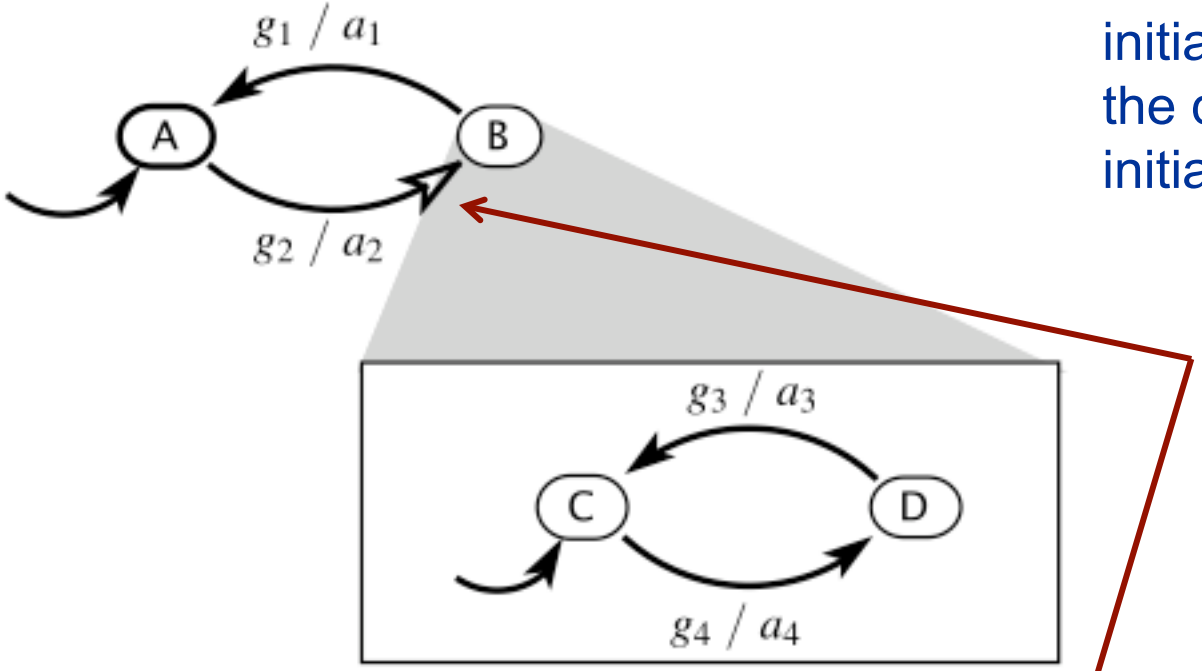




A history transition implies that when a state with a refinement is left, it is nonetheless necessary to remember the state of the refinement. Hence A,C and A,D.

Hierarchical State Machines with Reset Transitions

A reset transition always initializes the refinement of the destination state to its initial state.



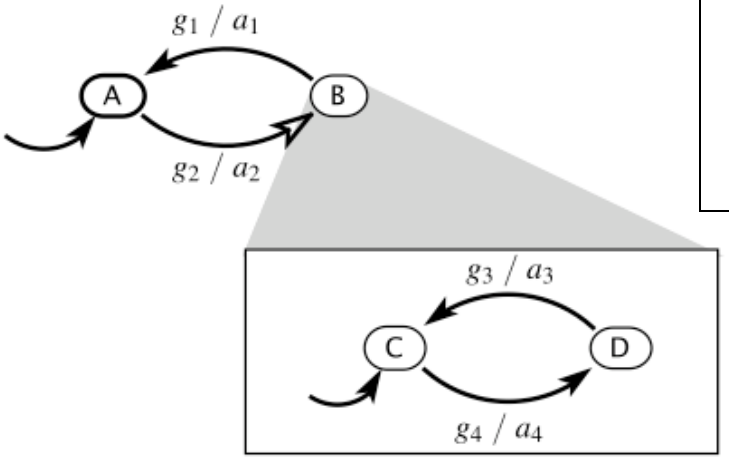
reset transition

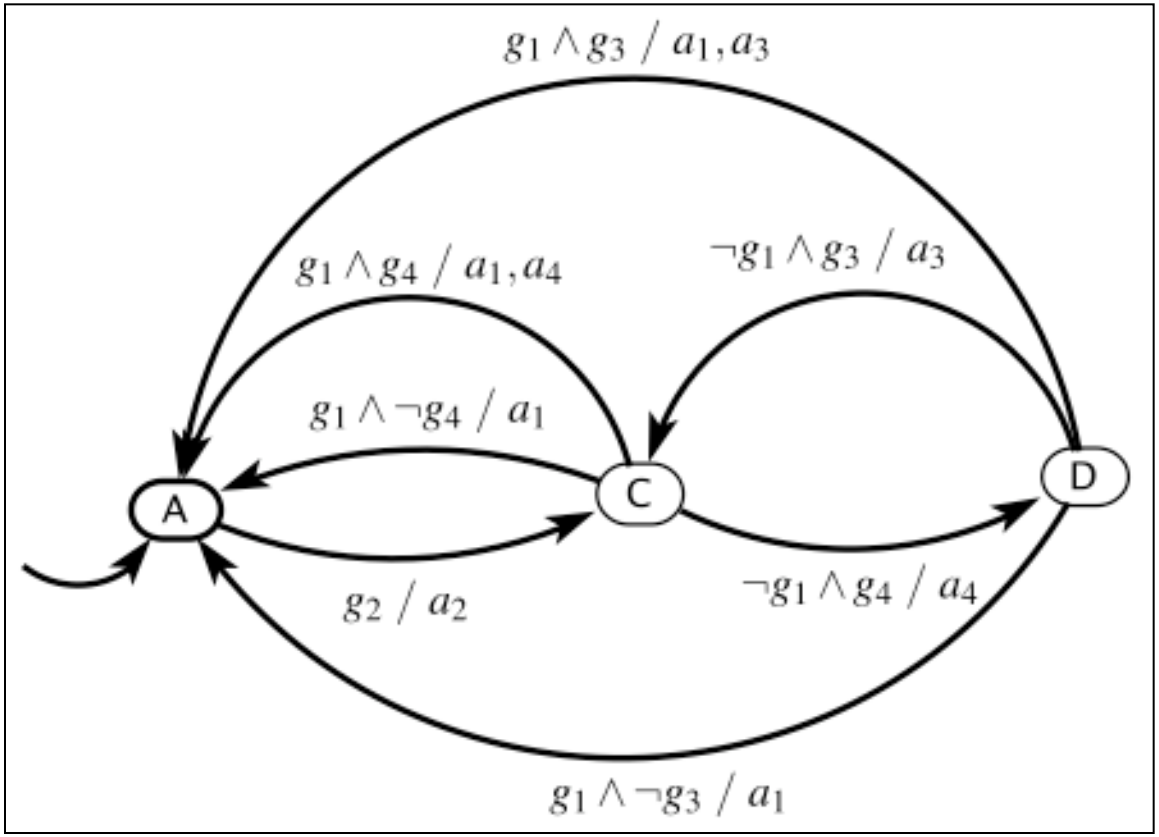
Example trace:

$$A \xrightarrow{g_2/a_2} C \xrightarrow{g_4/a_4} D \xrightarrow{g_1/a_1} A \xrightarrow{g_2/a_2} C \xrightarrow{g_4 \wedge g_1/a_4,a_1} A \cdots$$

A reset transition implies that when a state with a refinement is left, you can forget the state of the refinement.

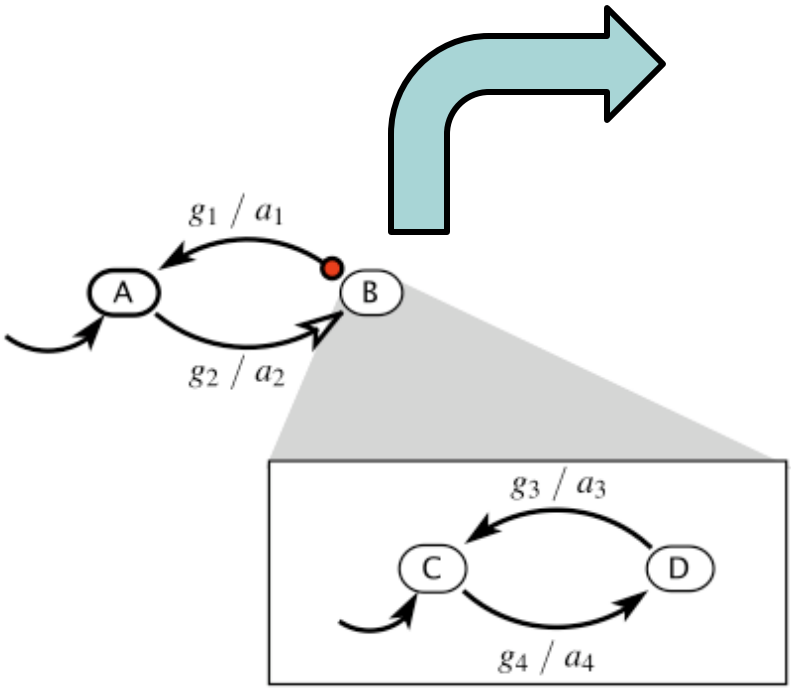
Flattening the state machine (assuming reset transitions):

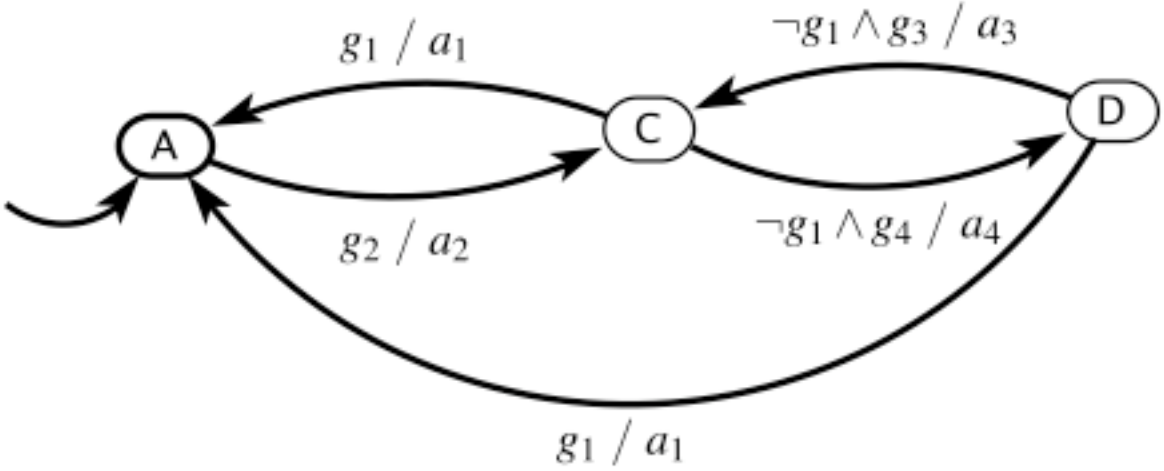




A reset transition implies that when a state with a refinement is left, it is not necessary to remember the state of the refinement. Hence there are fewer states.

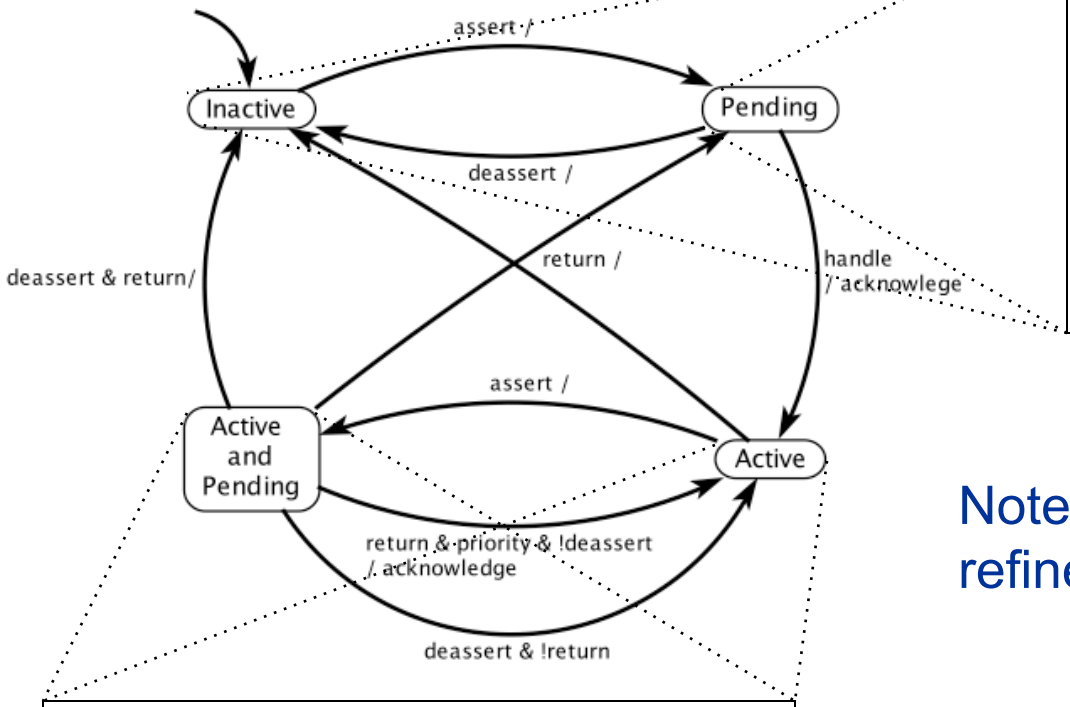
Preemptive Transitions





A preemptive transition specifies that the guard should be evaluated before the current state refinement reacts, and if it is true, then the current state should not react.

Modeling an interrupt controller



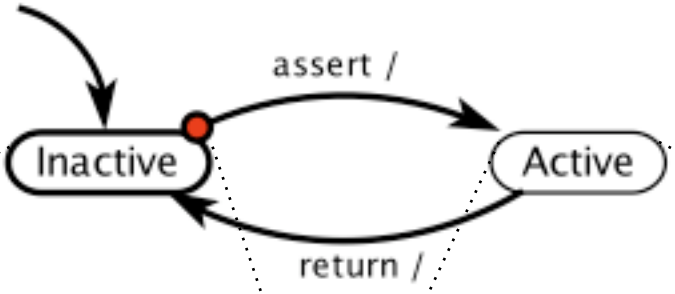

```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

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

Note that states can share refinements.

Simplified interrupt controller

This abstraction assumes that an interrupt is always handled immediately upon being asserted:



A
B
C

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

D
E

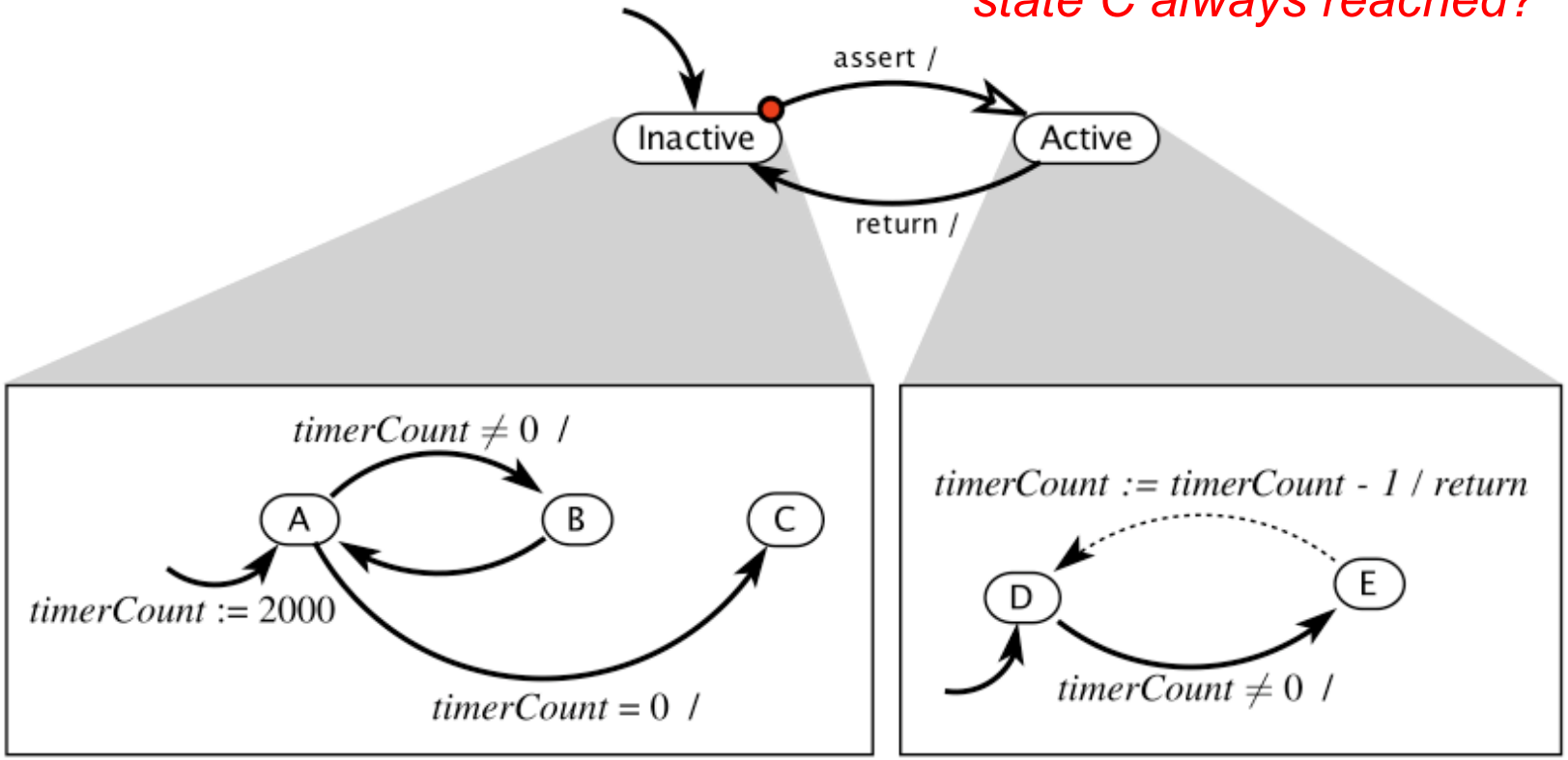
```
volatile uint timerCount = 0;
void ISR(void) {
   … disable interrupts
   if(timerCount != 0) {
      timerCount--;
   }
   … enable interrupts
}
```

Hierarchical interrupt controller

This model assumes further that interrupts are disabled in the ISR:

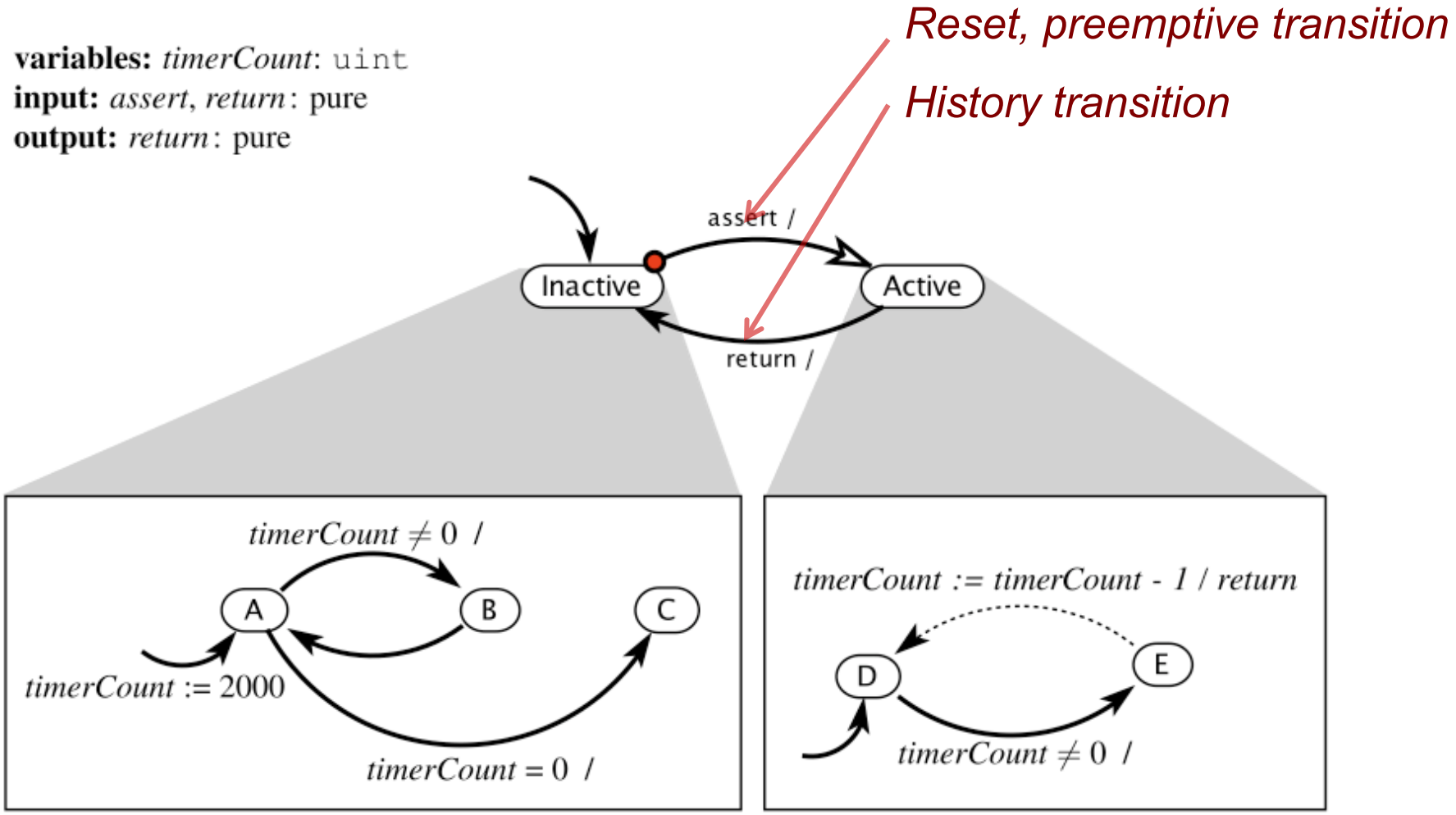
variables: *timerCount*: `uint`
input: *assert*, *return*: pure
output: *return*: pure

A key question: Assuming interrupt occurs infinitely often, is state C always reached?



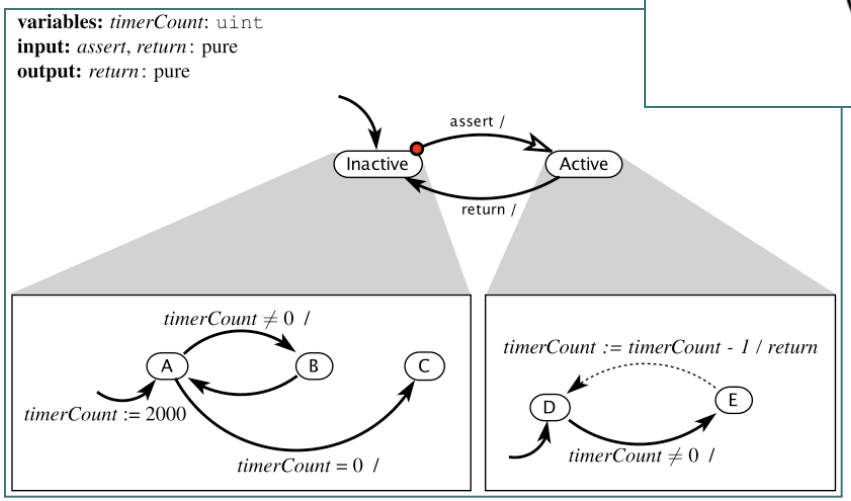
Hierarchical interrupt controller

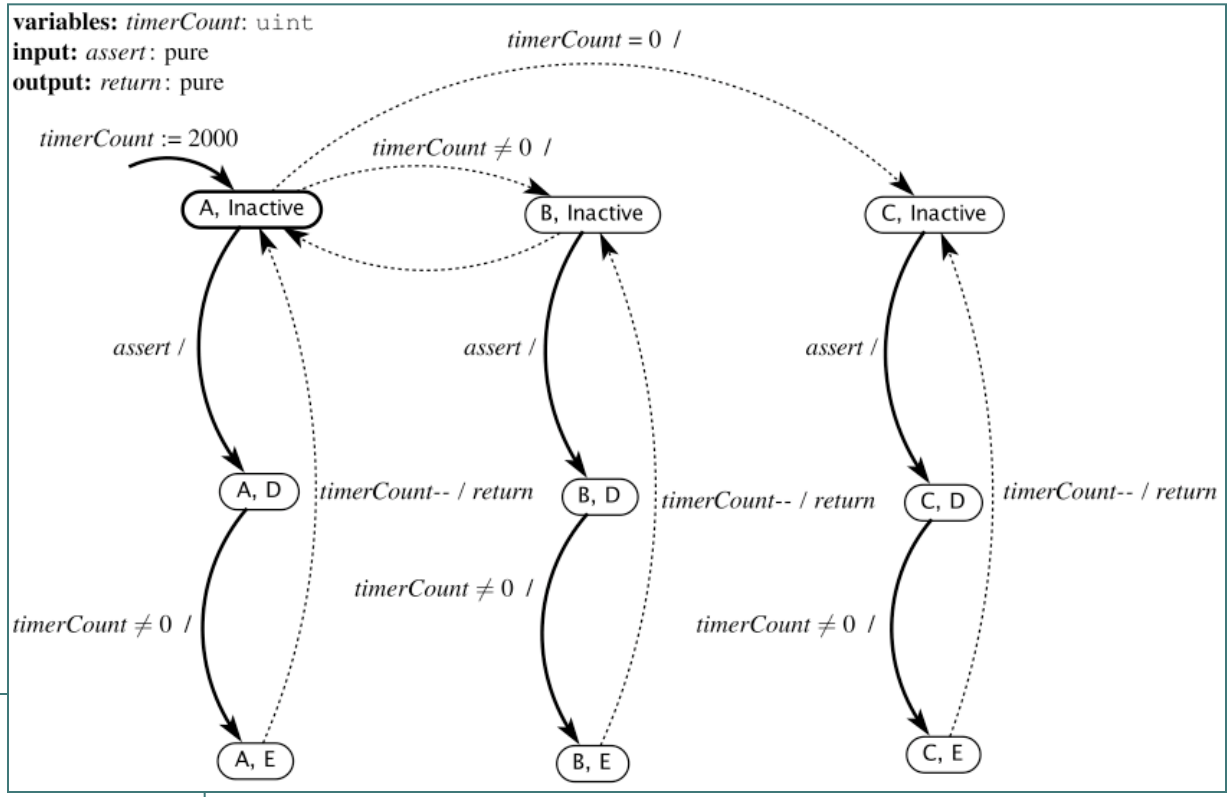
This model assumes interrupts are disabled in the ISR:



Hierarchical composition to model interrupts

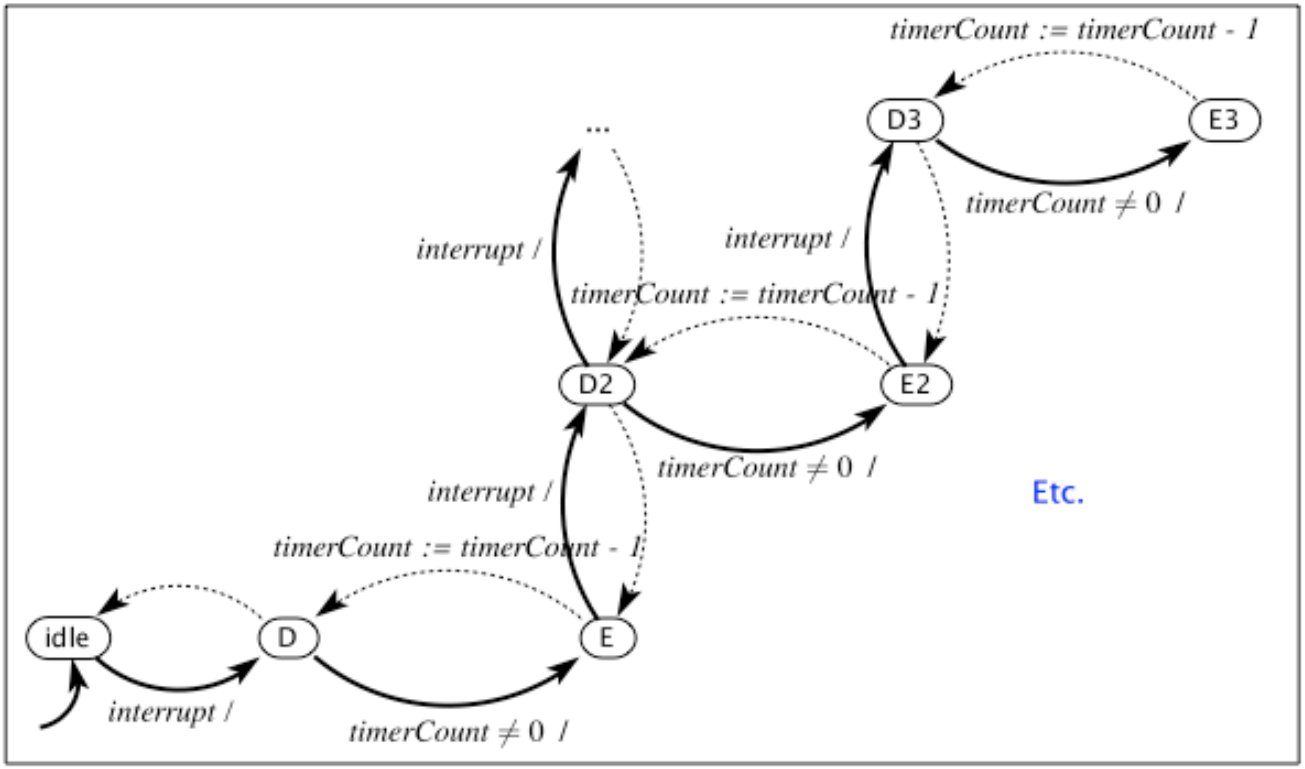
History transition results in product state space, but hierarchy reduces the number of transitions compared to asynchronous composition.

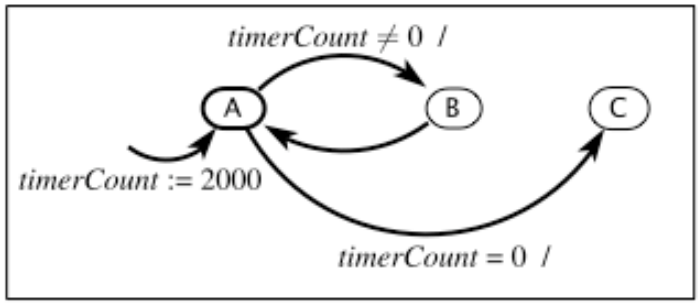




Examining this composition machine, it is clear that C is not necessarily reached if the interrupt occurs infinitely often. If assert is present on every reaction, C is never reached.

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





What if interrupts are not disabled?

A key question: Assuming interrupt occurs infinitely often, is state C always reached?

Answer: NO! Counterexample: each time timerCount = 1, get more than one nested interrupt. Trace in upper machine: idle, D, E, D2, E2, D2, E, D, …

Communicating FSMs

In the ISR, example our FSM models of the main program and the ISR communicate via shared variables and the FSMs are composed asynchronously.

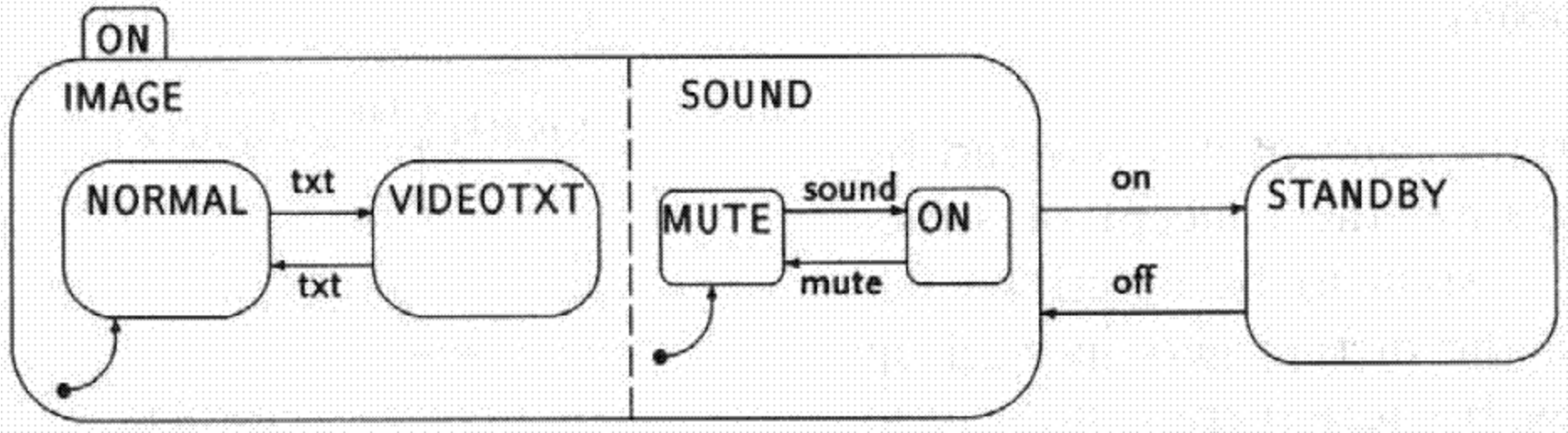
We call this model of computation *threads*.

There are better alternatives for concurrent composition.

Hierarchical FSMs + Synchronous Composition: Statecharts [Harel 87]

Modeling with

- Hierarchy (OR states)
- Synchronous composition (AND states)
- Broadcast (for communication)



Example due to Reinhard von Hanxleden

Summary

- Composition enables building complex systems from simpler ones.
- Hierarchical FSMs enable compact representations of complex behaviors.
- Both forms of composition can be converted to single flat FSMs, but the resulting FSMs are quite complex and difficult to analyze by hand.
- Algorithmic techniques are needed (e.g., model checking, the inventors of which won the 2009 Turing Award).