COMPUTER ENGINEERING

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

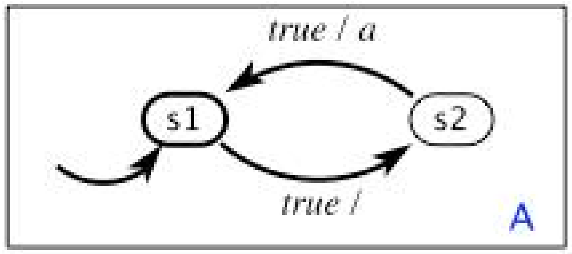
EMBEDDED SYSTEM DESIGN
Hierarchical State Machines

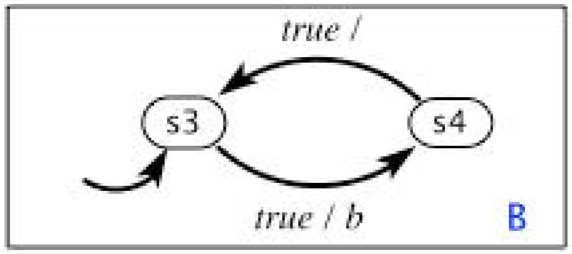
Doan Duy, Ph. D.
Email: duyd@uit.edu.vn

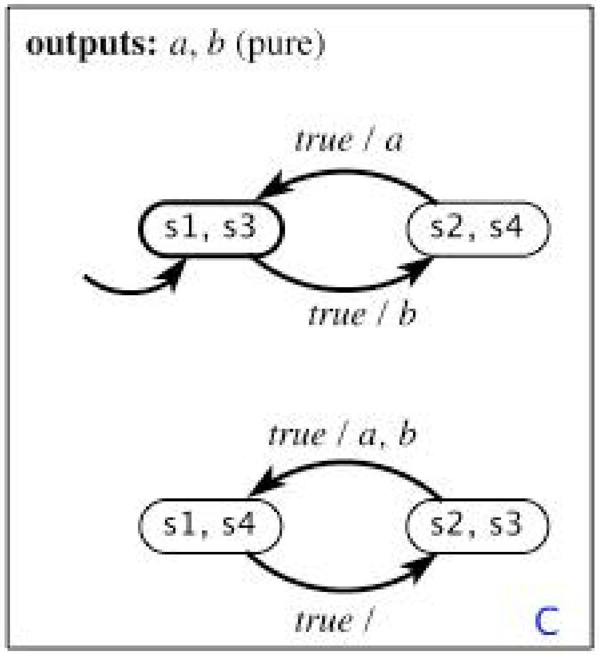
Recall Synchronous Composition:

$$S_C = S_A \times S_B$$

outputs: a, b (pure)



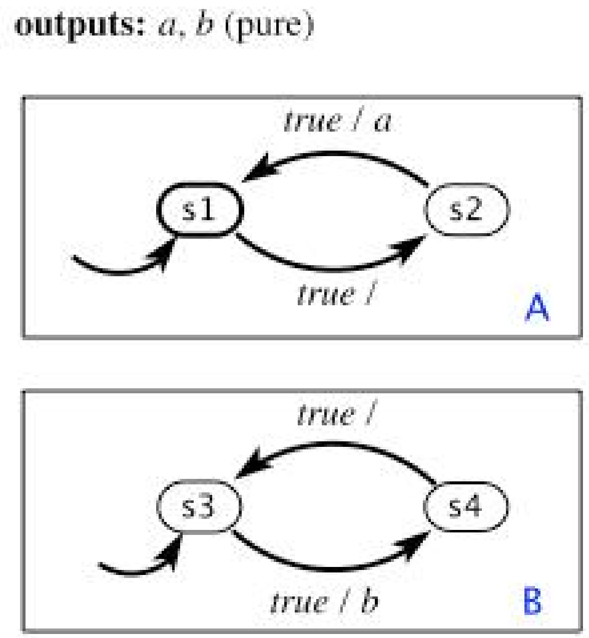


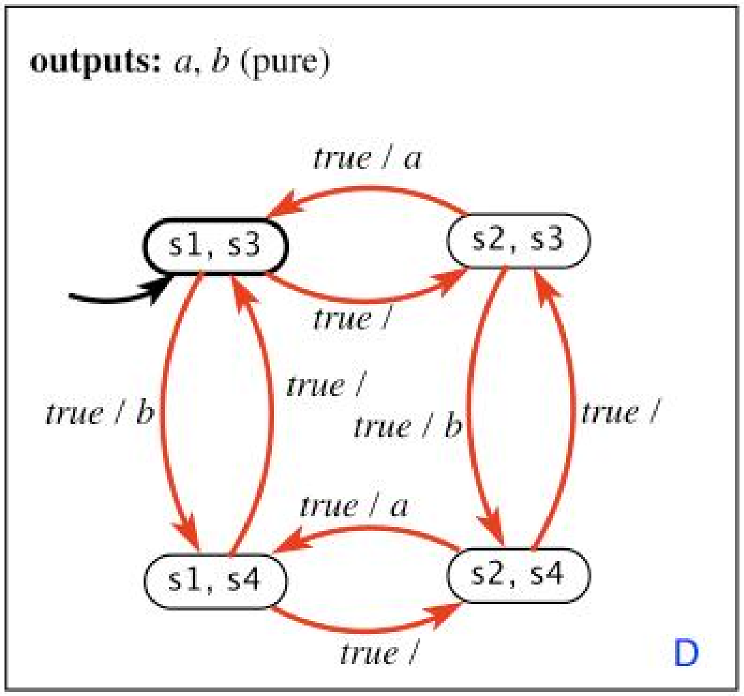


Synchronous composition

Recall Asynchronous Composition:

$$S_C = S_A \times S_B$$





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);
  SysTickEnable();
  ... // other init
  timerCount = 2000;
  while(timerCount != 0) {
    ... code to run for 2 seconds
  }
}
```

Position in the program is part of the state

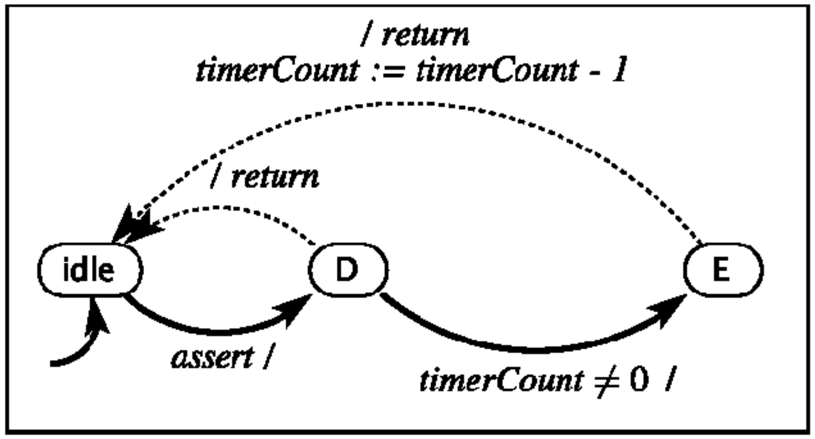
```
      volatile uint timerCount = 0;
      void ISR(void) {
         … disable interrupts
D →      if(timerCount != 0) {
E →         timerCount--;
         }
         … enable interrupts
      }
      int main(void) {
         // initialization code
         SysTickIntRegister(&ISR);
         SysTickEnable();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

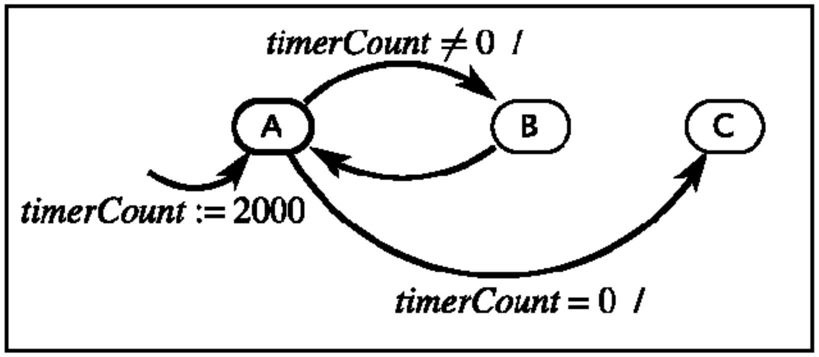
A key question: Assuming interrupt can occur infinitely often, is position C always reached?

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
A →  timerCount = 2000;
B →  while(timerCount != 0) {
       … code to run for 2 seconds
     }
C  … whatever comes next
  }
```

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

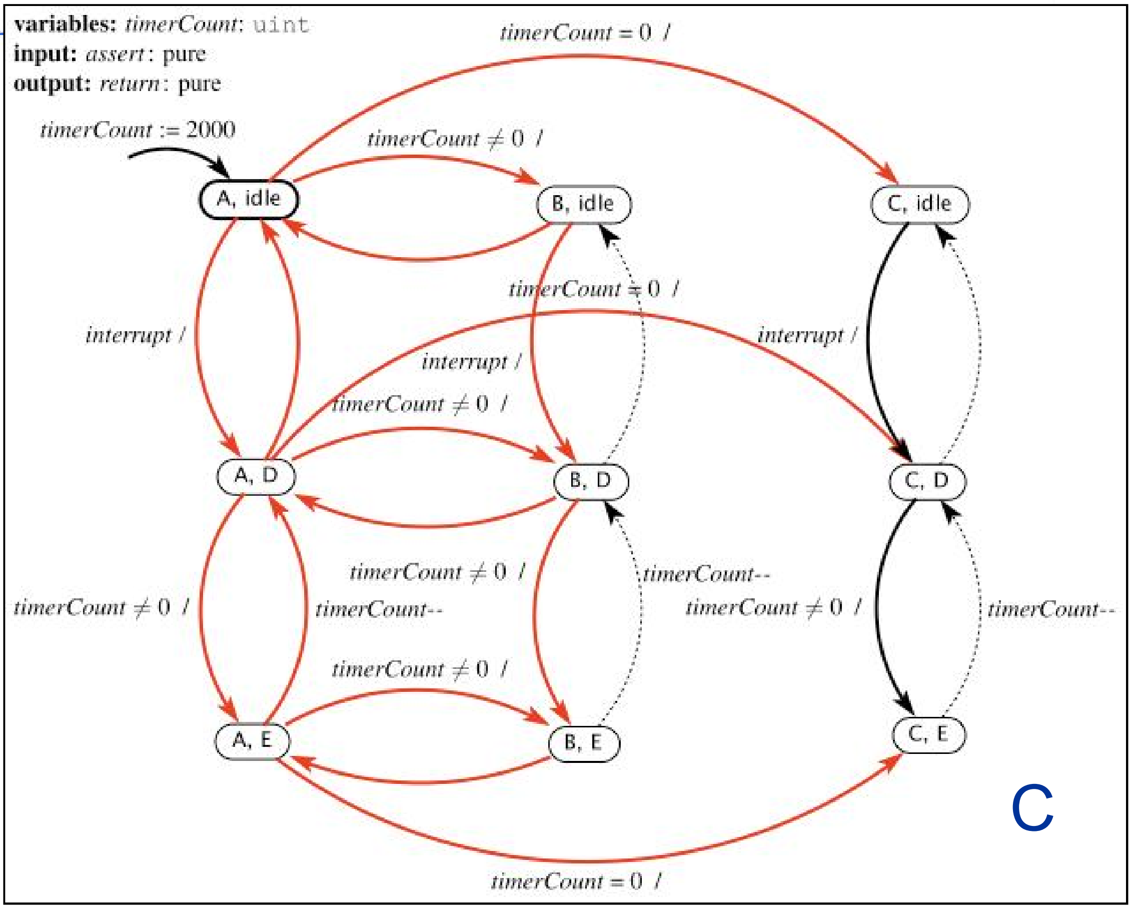




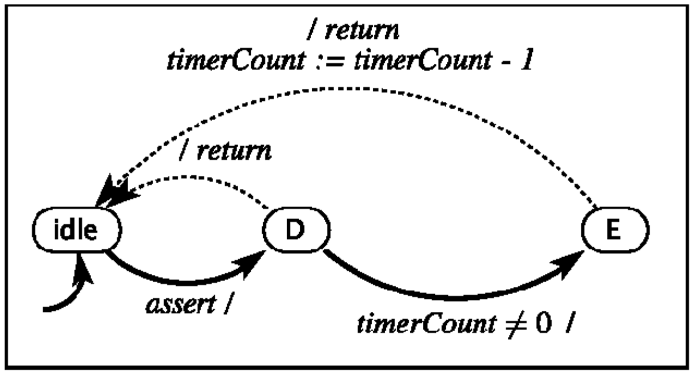
Is asynchronous composition the right thing to do here?

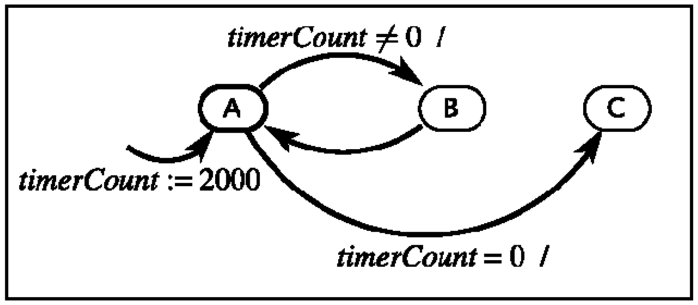
Asynchronous composition

$$S_C = S_A \times S_B$$



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.

Asynchronous vs Synchronous Composition

```
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?

COMPUTER ENGINEERING

Asynchronous vs Synchronous Composition

```
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?

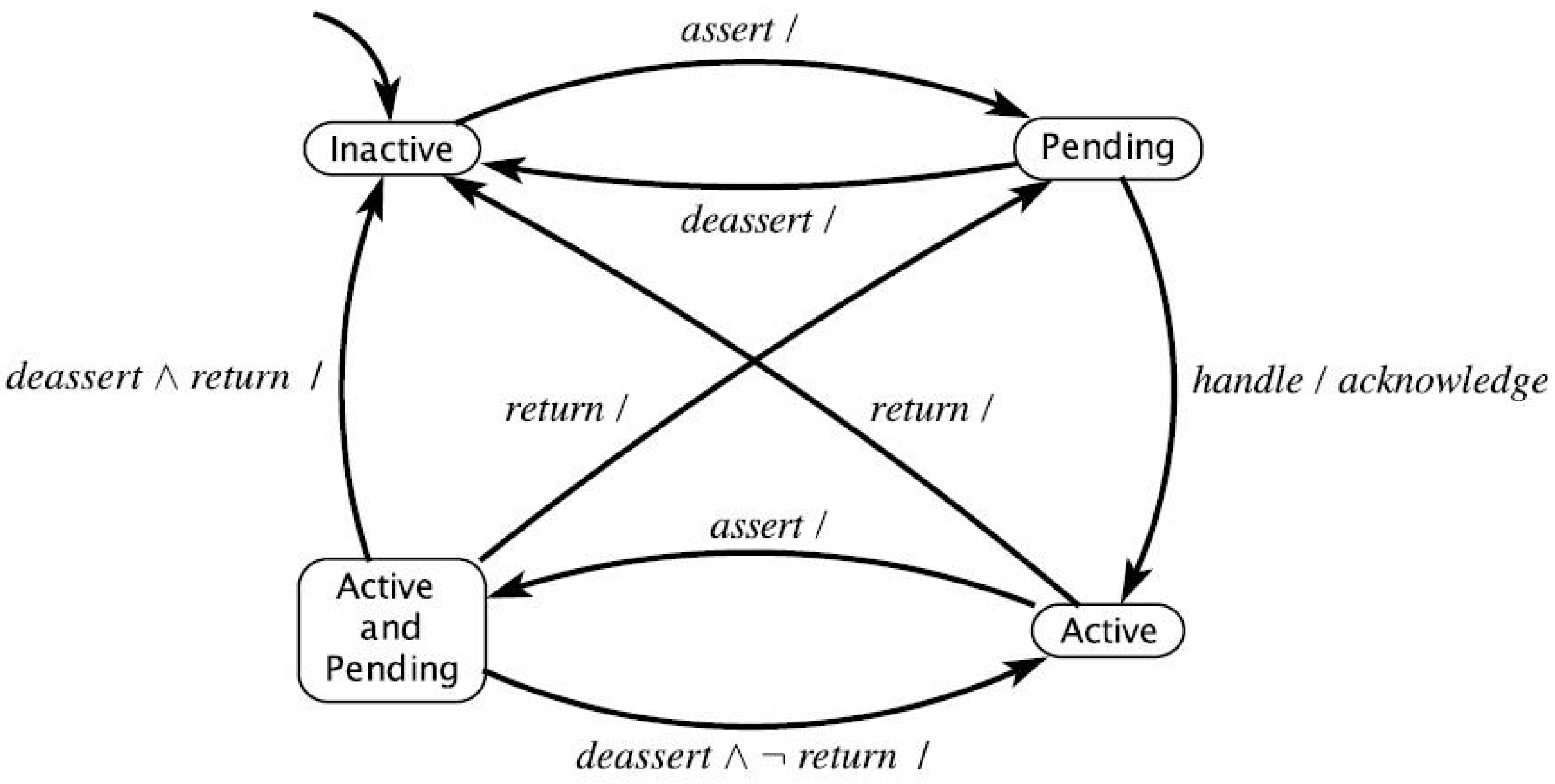
Is asynchronous composition (with interleaving semantics) the right model for this?

Answer: no to both.

Modeling an interrupt controller

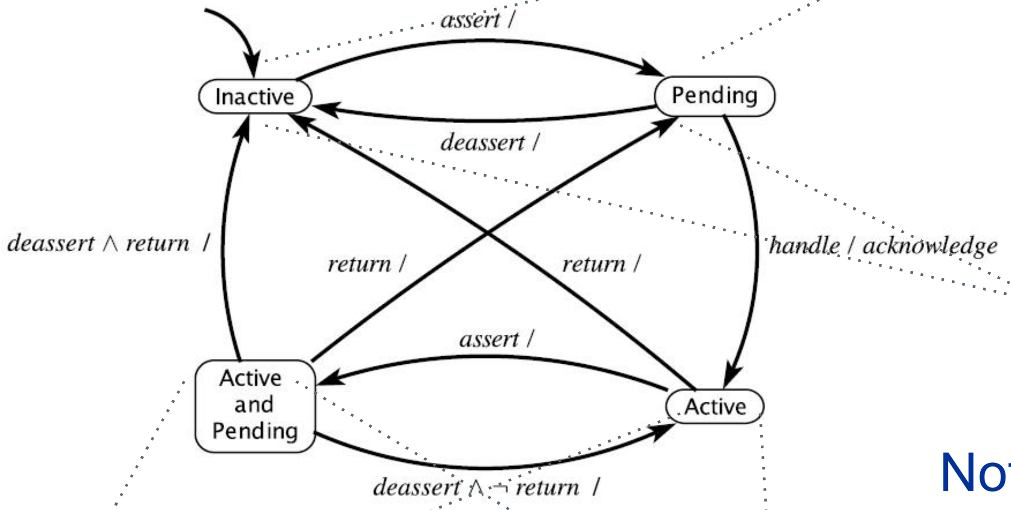
- FSM model of a single interrupt handler in an interrupt controller:

input: *assert*, *deassert*, *handle*, *return* : pure
output: *acknowledge*



Modeling an interrupt controller

input: *assert*, *deassert*, *handle*, *return*: pure
output: *acknowledge*

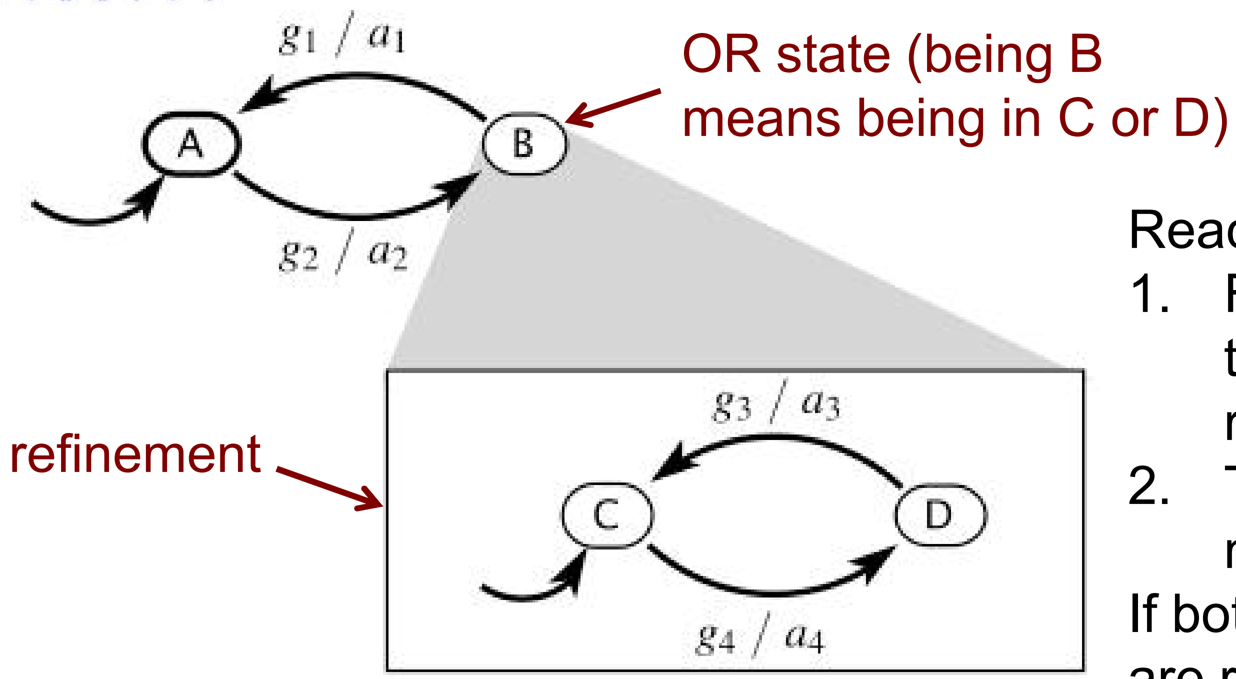


```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

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

Note that states can share refinements.

Hierarchical State Machines



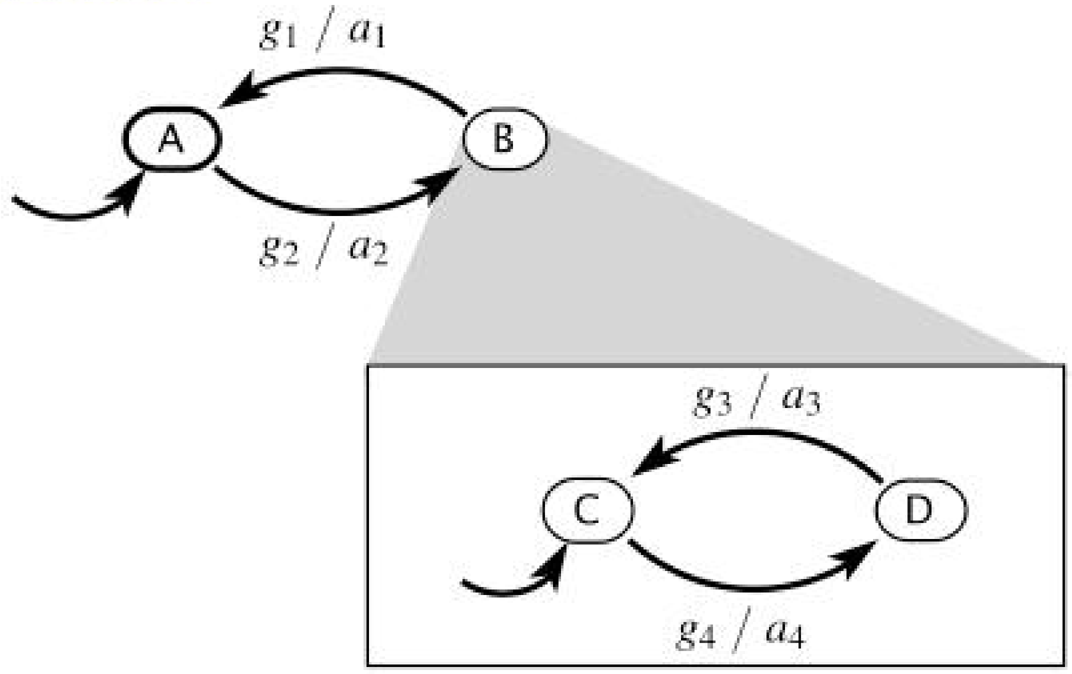
Reaction:

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.

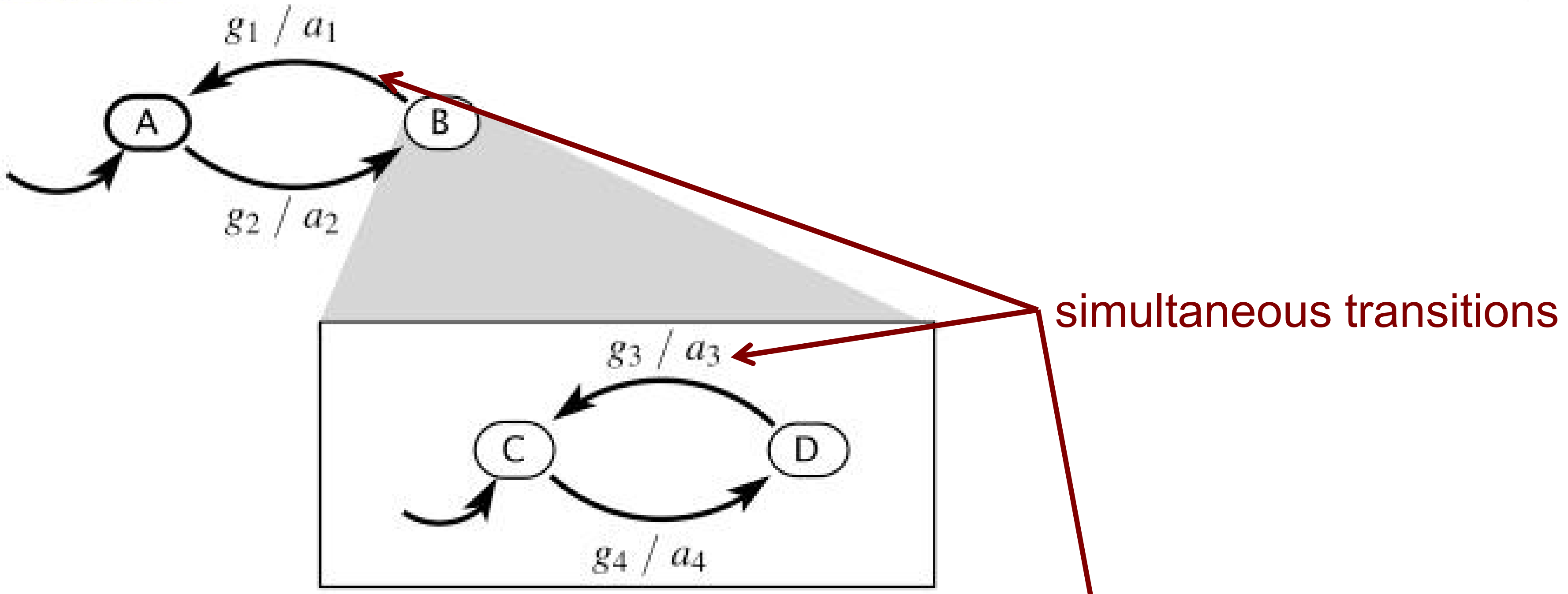
[Statecharts, David Harel, 1987]

Hierarchical State Machines



Example trace:

Hierarchical State Machines

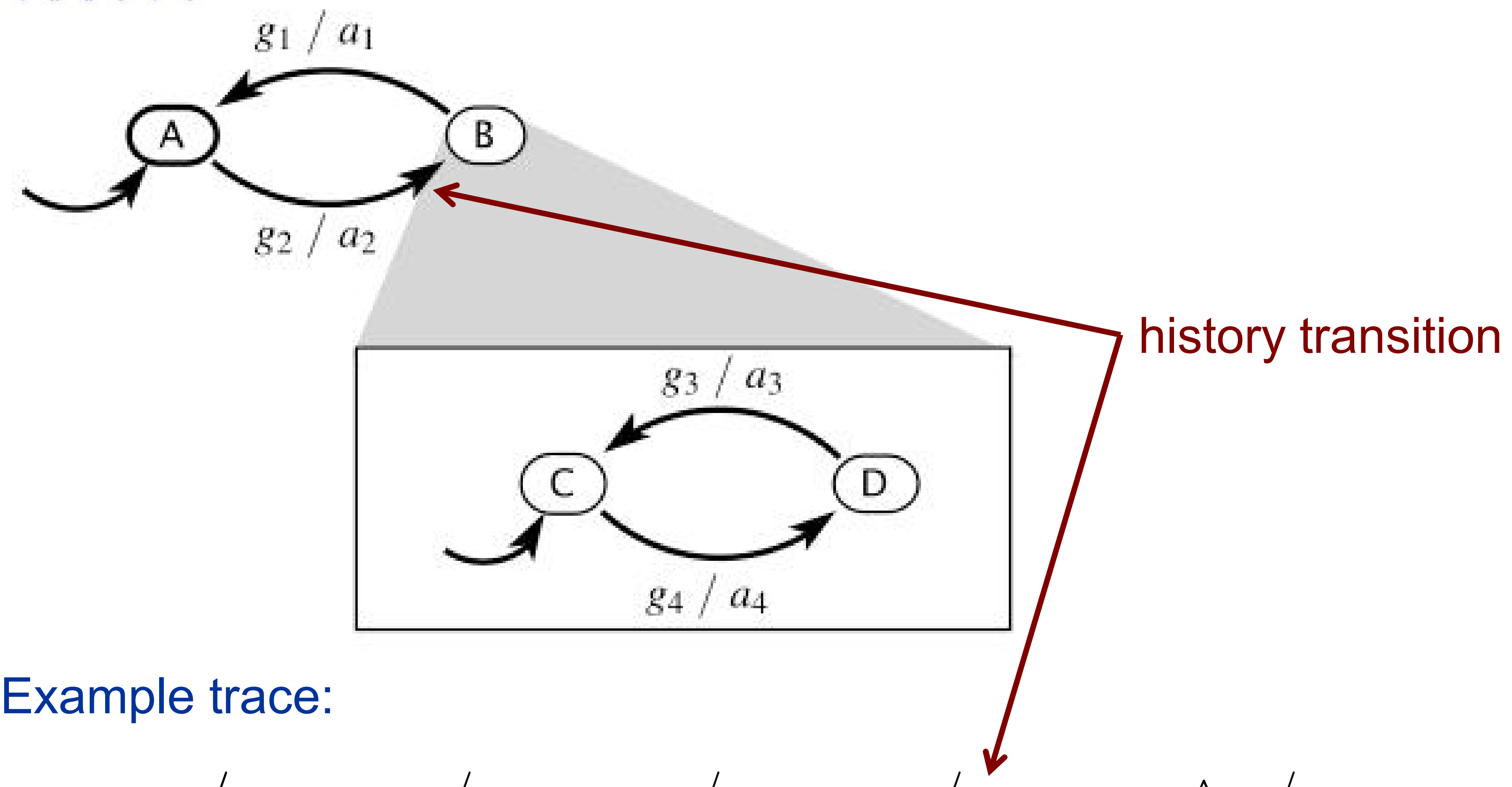


Example trace:

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

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

Hierarchical State Machines



Example trace:

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

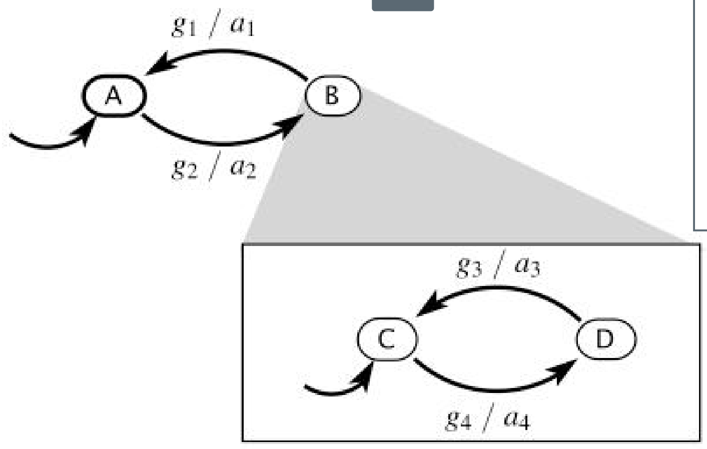
A history transition implies that when a state with a refinement is left, it is nonetheless necessary to remember the state of the refinement.

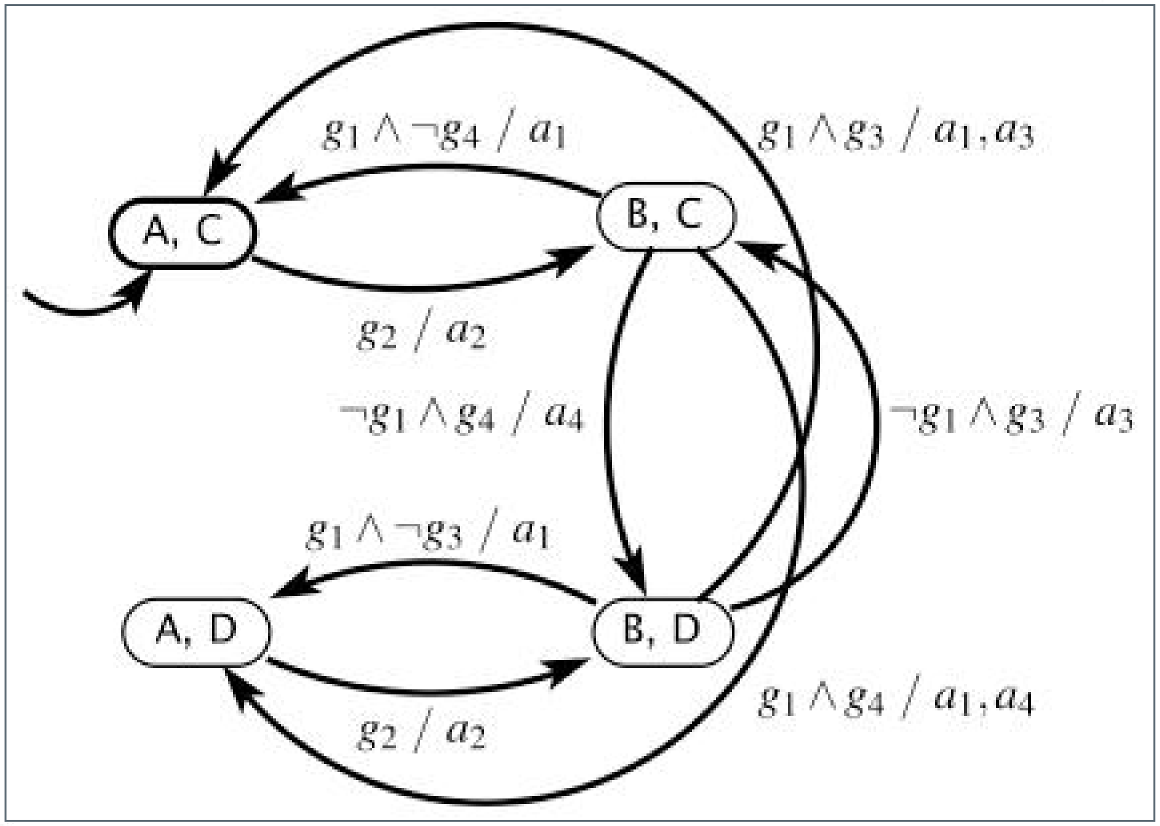
Equivalent Flattened State Machine

- Every hierarchical state machine can be transformed into an equivalent “flat” state machine.

- This transformation can cause the state space to blow up substantially.

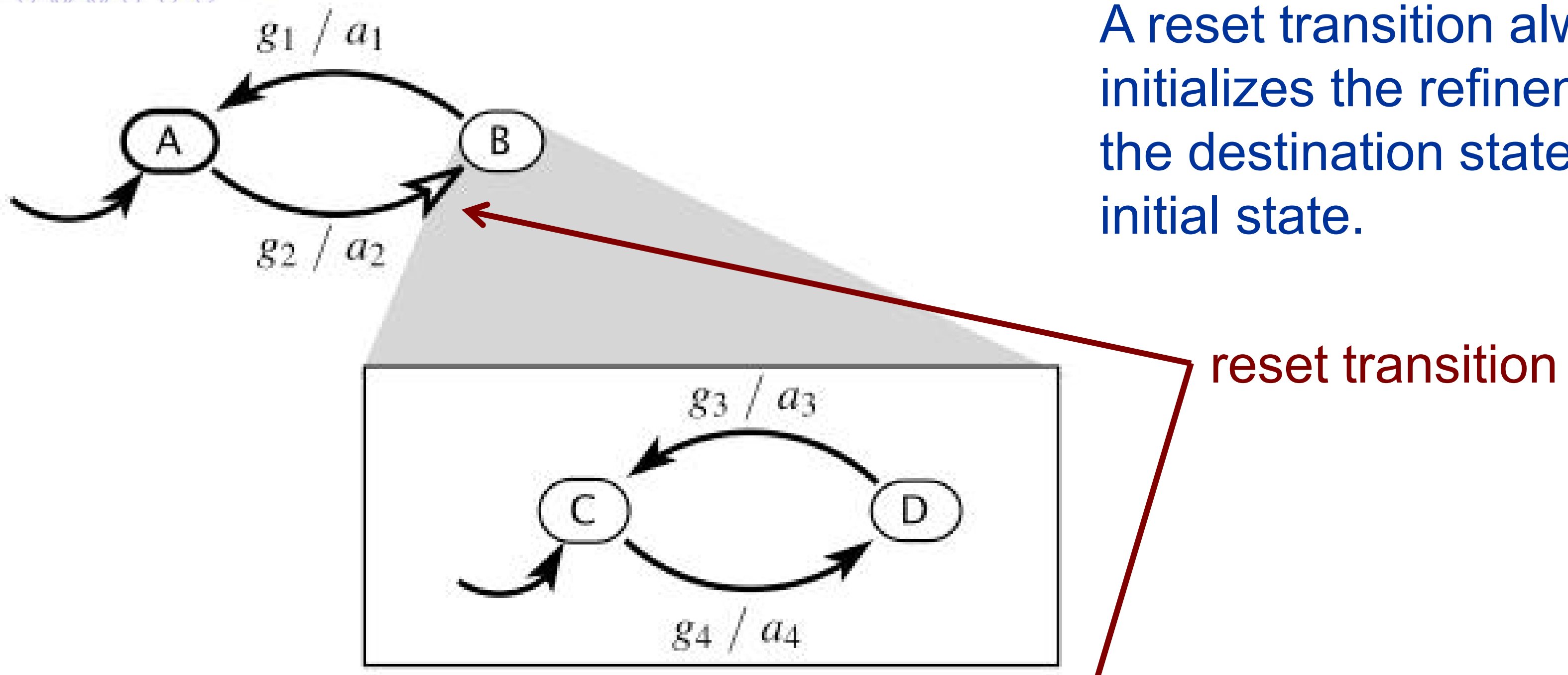
Flattening the state machine (assuming history transitions):




$g_1 \wedge \neg g_4 / a_1$
$g_1 \wedge g_3 / a_1, a_3$
A, C
B, C
g_2 / a_2
$\neg g_1 \wedge g_4 / a_4$
$\neg g_1 \wedge g_3 / a_3$
$g_1 \wedge \neg g_3 / a_1$
A, D
B, D
$g_1 \wedge g_4 / a_1, a_4$
g_2 / a_2

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



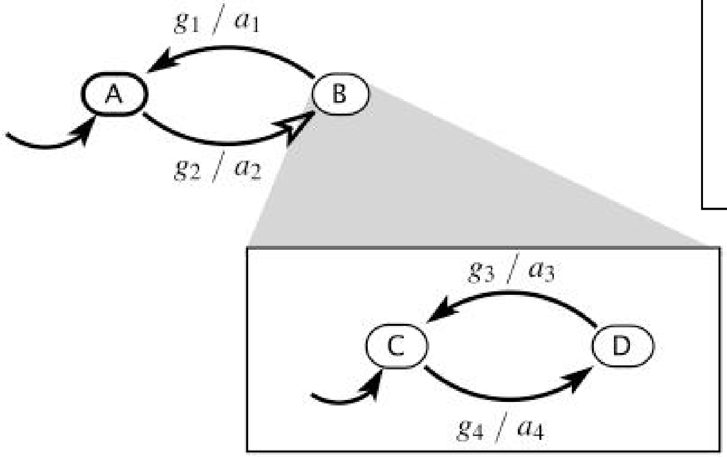
A reset transition always initializes the refinement of the destination state to its initial state.

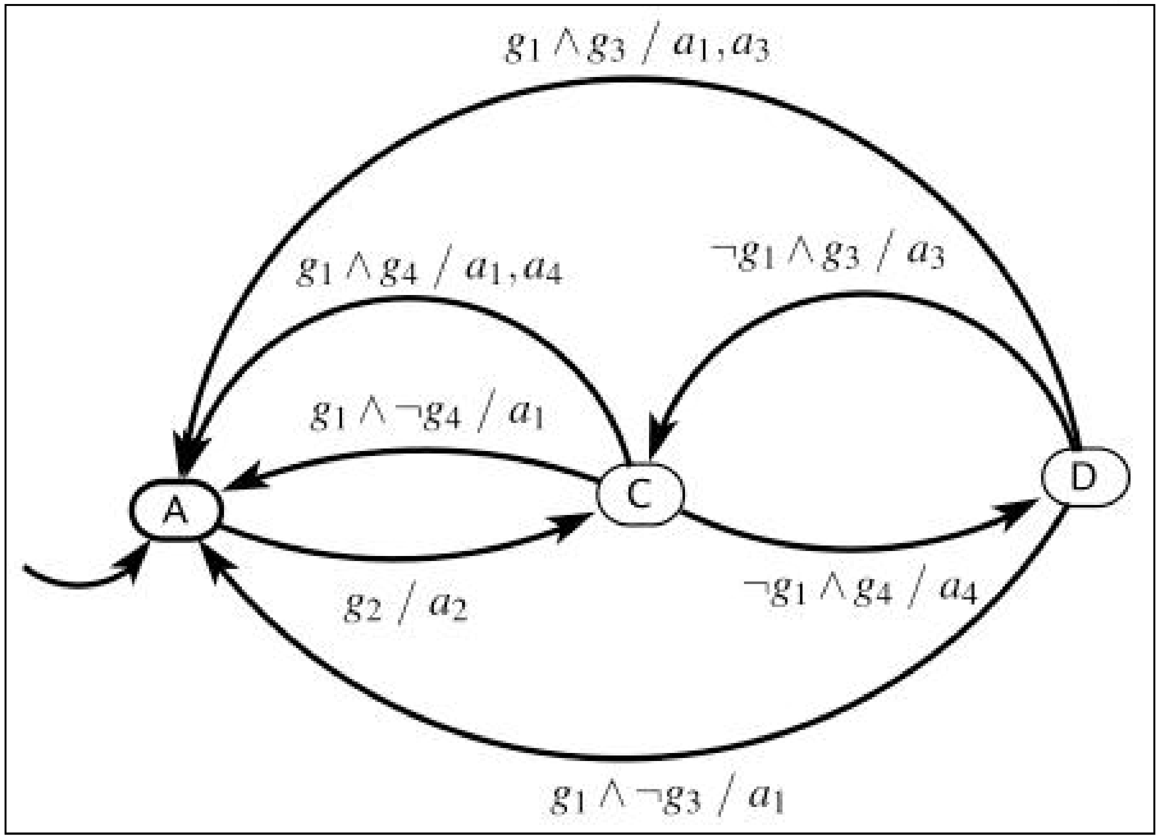
Example trace:

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

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

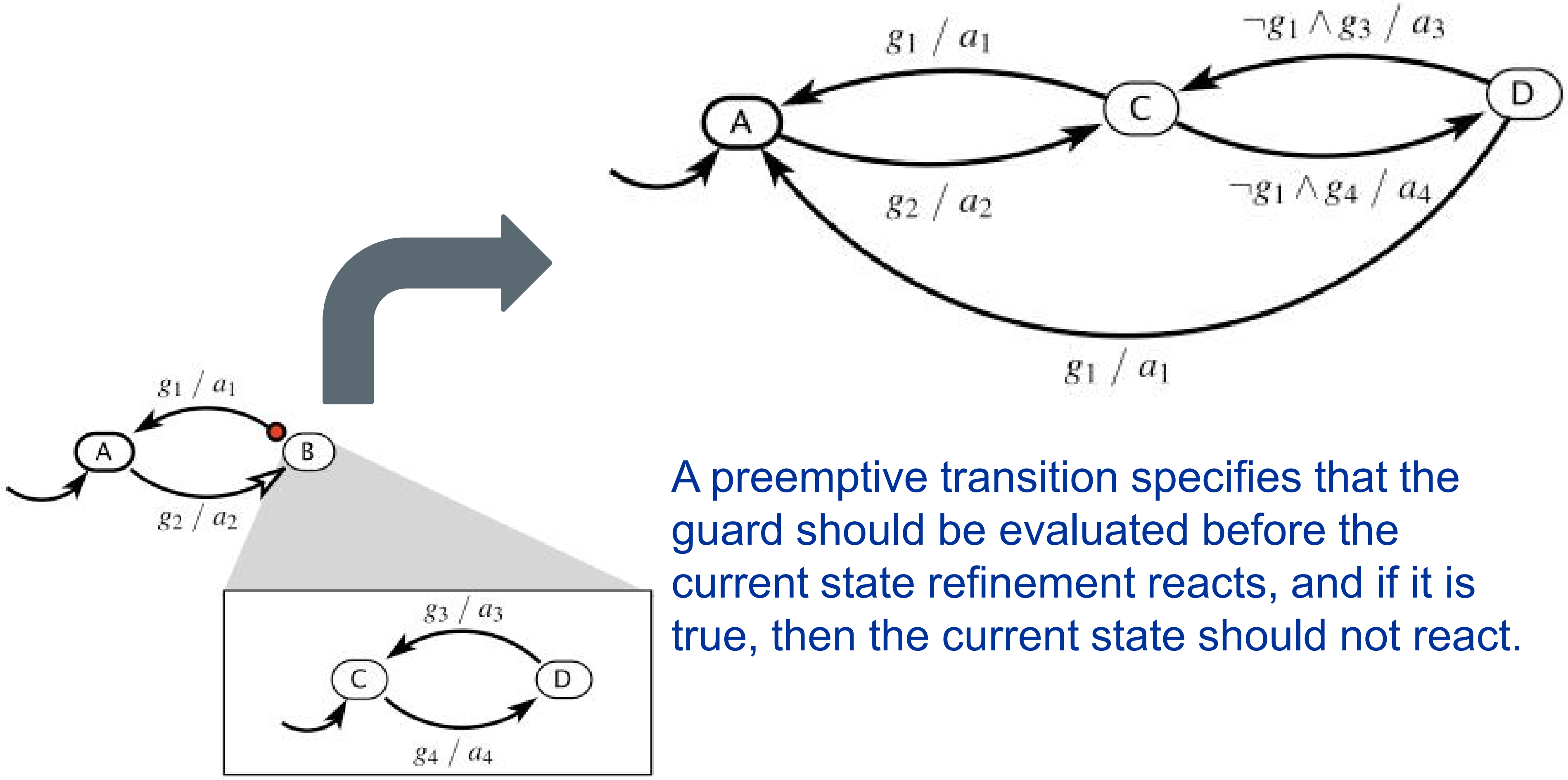
Flattening the state machine (assuming reset transitions):




$g_1 \wedge g_3 / a_1, a_3$
$g_1 \wedge g_4 / a_1, a_4$
$\neg g_1 \wedge g_3 / a_3$
$g_1 \wedge \neg g_4 / a_1$
A
C
D
g_2 / a_2
$\neg g_1 \wedge g_4 / a_4$
$g_1 \wedge \neg g_3 / a_1$

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.

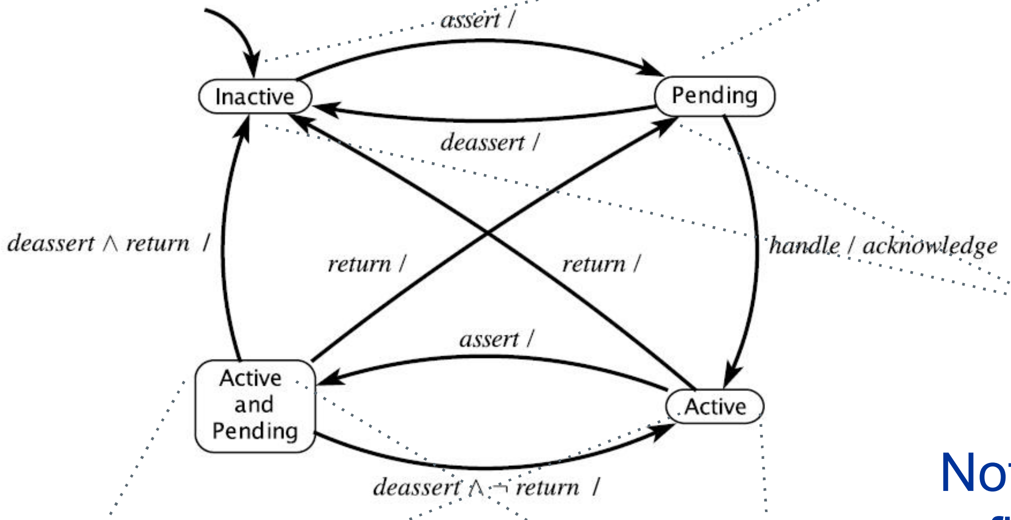
Summary of Key Concepts

- States can have refinements (other modal models)
  - OR states
  - AND states

- Different types of transitions:
  - History
  - Reset
  - Preemptive

Modeling an interrupt controller

input: *assert, deassert, handle, return*: pure
output: *acknowledge*



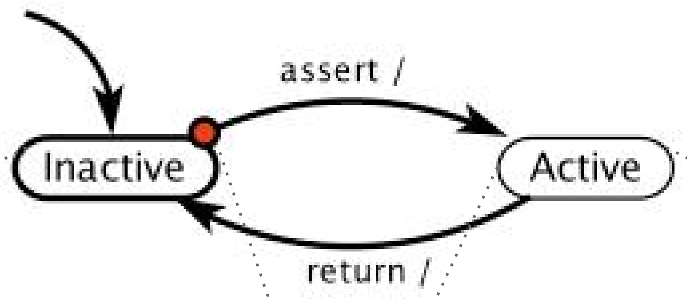
```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

```
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:



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

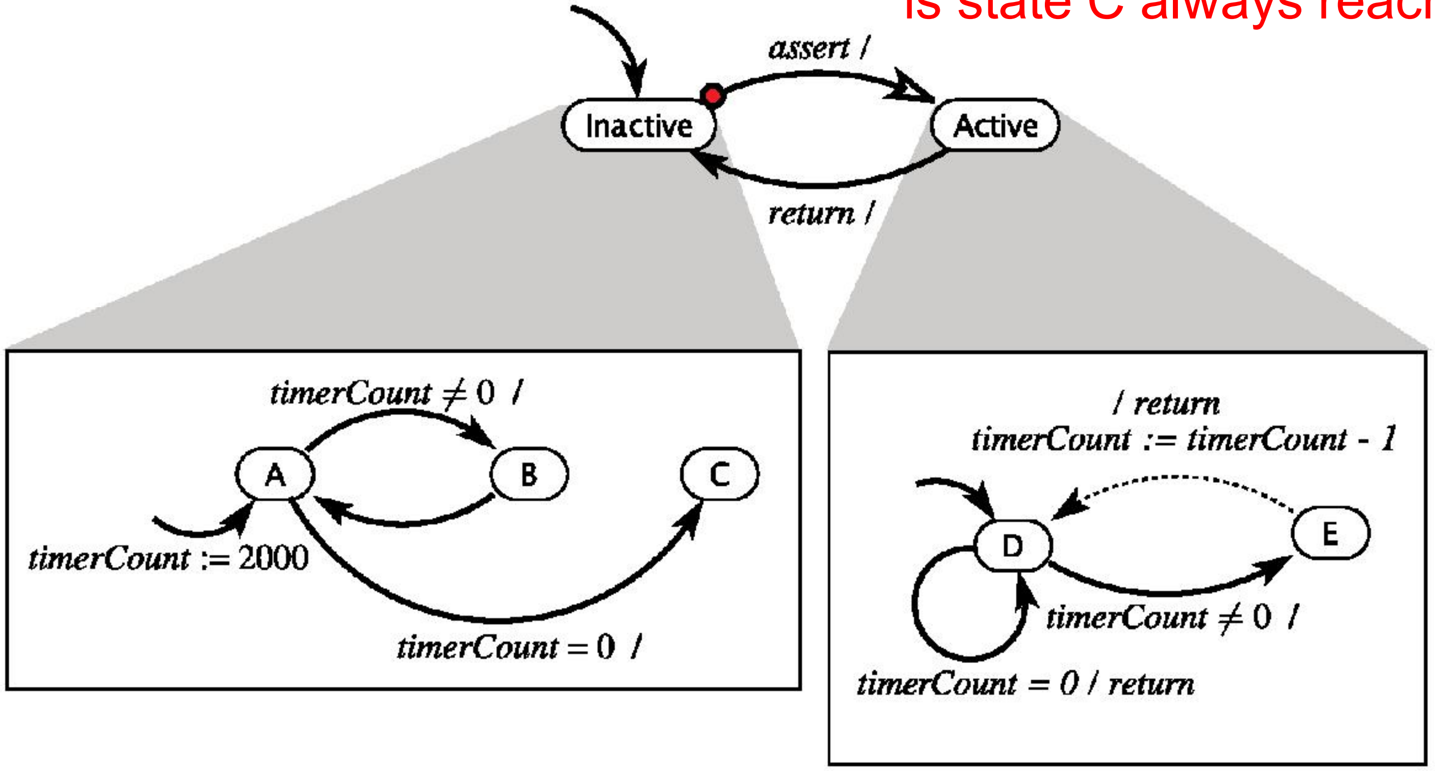
```
    volatile uint timerCount = 0;
    void ISR(void) {
D →    … disable interrupts
       if(timerCount != 0) {
E →       timerCount--;
       }
       … enable interrupts
    }
```

Hierarchical interrupt controller

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

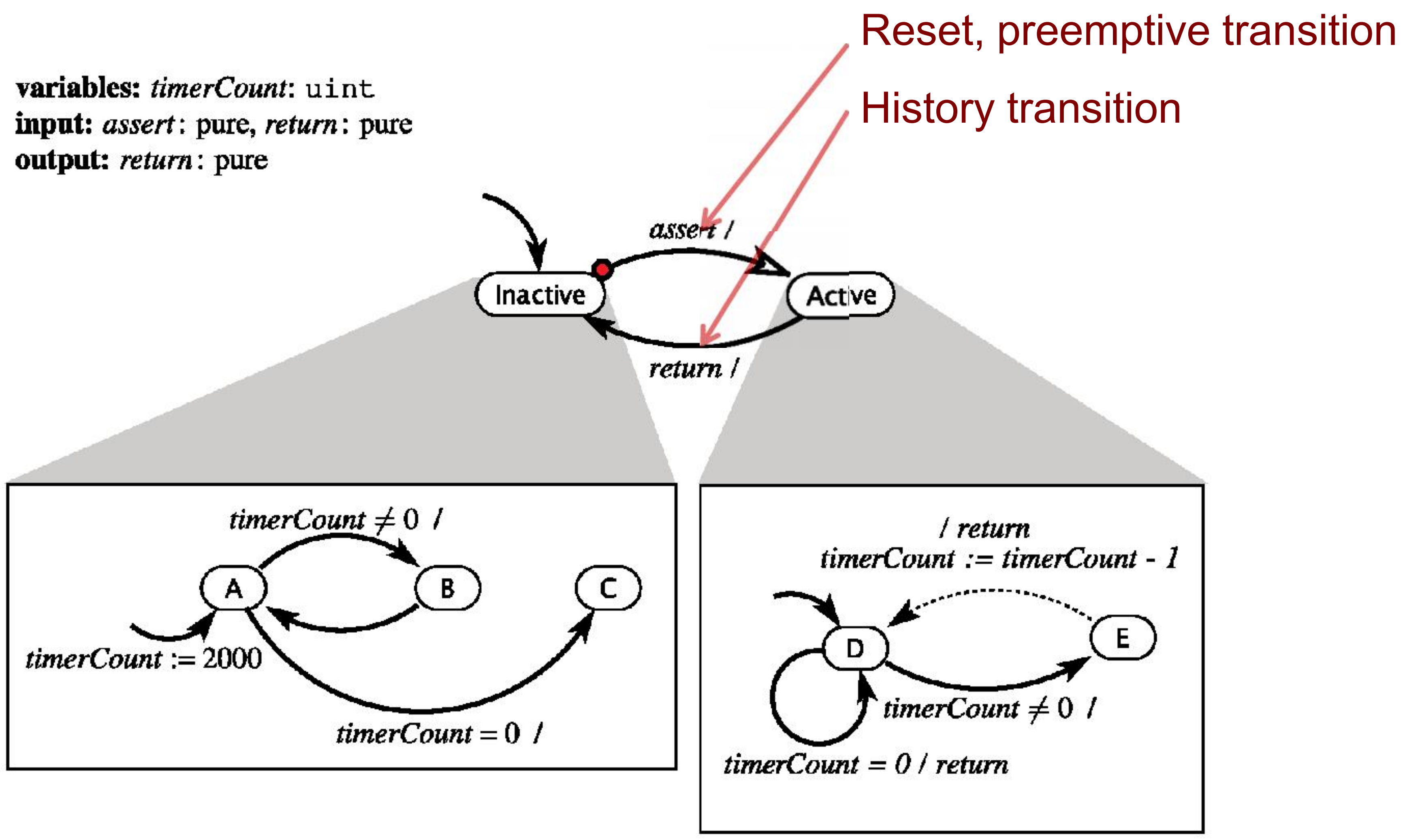
variables: *timerCount*: `uint`
input: *assert*: pure, *return*: pure
output: *return*: pure

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



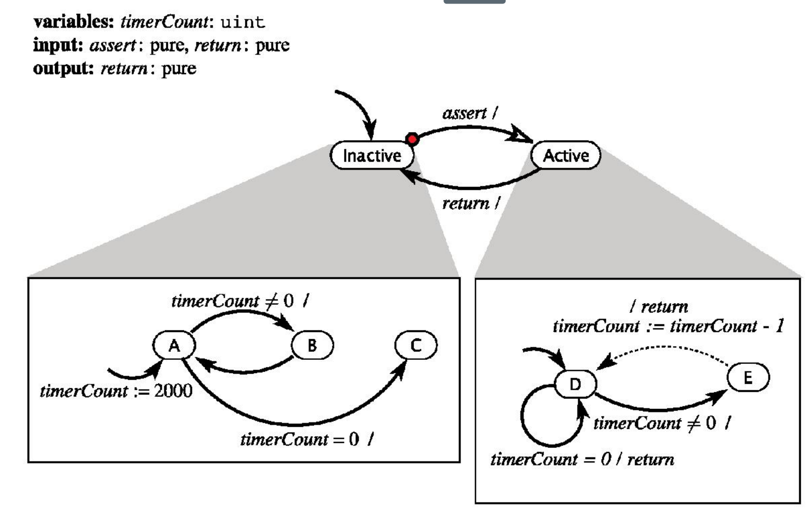
Hierarchical interrupt controller

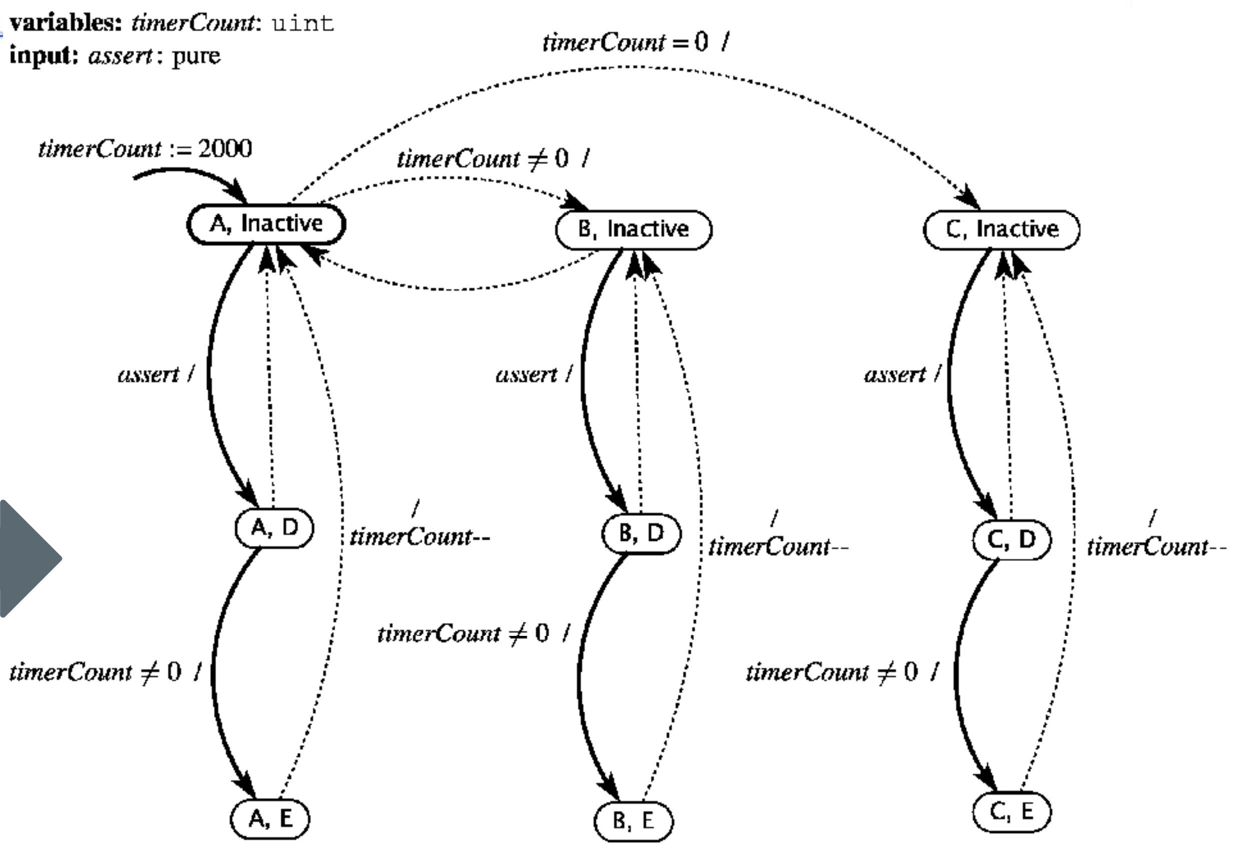
- This model assumes interrupts are disabled in the ISR:



Hierarchical composition to model interrupts

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

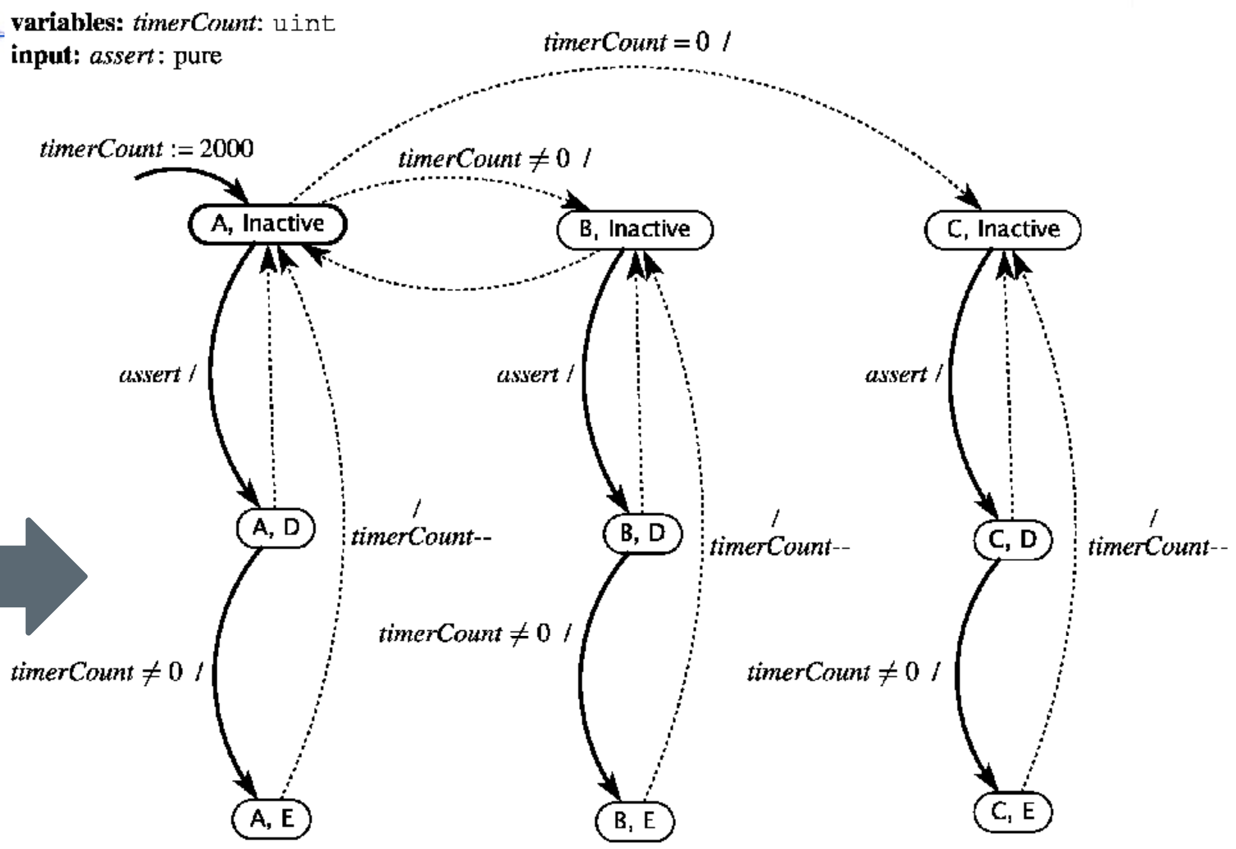


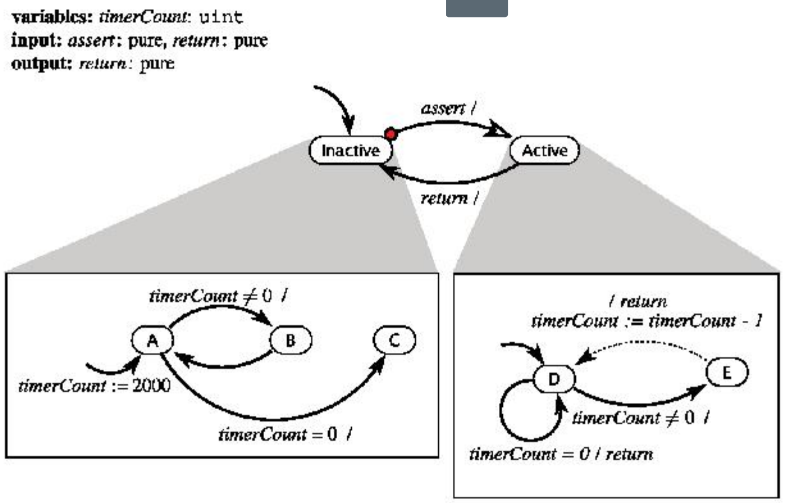


Examining this composition machine, it is clear that C is not necessarily reached if the interrupt occurs infinitely often. If assert is present on every reaction, C is never reached.

Hierarchical composition to model interrupts

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





Under what assumptions/model of “assert” would C be reached?

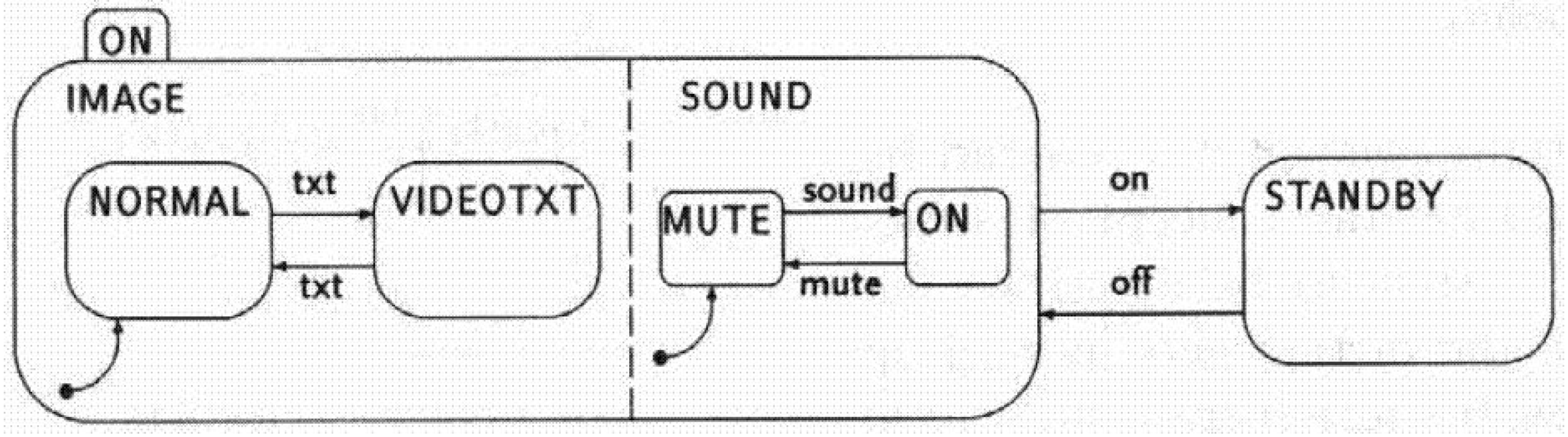
Communicating FSMs

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

- There are other alternatives for concurrent composition (see Chapter 6 of Lee & Seshia).

Hierarchical FSMs + Synchronous Composition: Statecharts

- 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 large state machines.
- These can be converted to single flat FSMs, but the resulting FSMs are quite complex and difficult to analyze by hand.
- Algorithmic techniques are needed to analyze large state spaces (e.g., *reachability analysis* and *model checking*, see Chapter 13 of Lee & Seshia).

COMPUTER ENGINEERING

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

Q&A