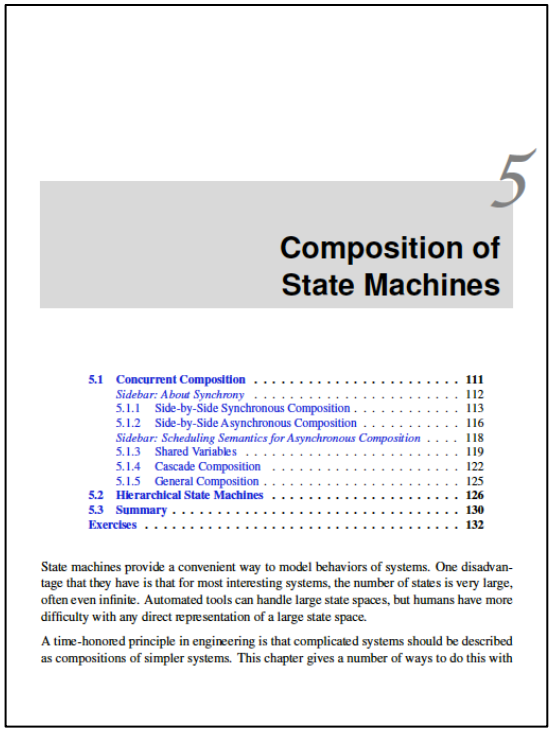
Lecture 12:
Hierarchical State Machines

Slides adapted from Edward A. Lee

Outline

- States can have refinements (other modal models)
  - OR states (hierarchy)
  - AND states (synchronous composition)
- Different types of transitions:
  - History
  - Reset
  - Preemptive

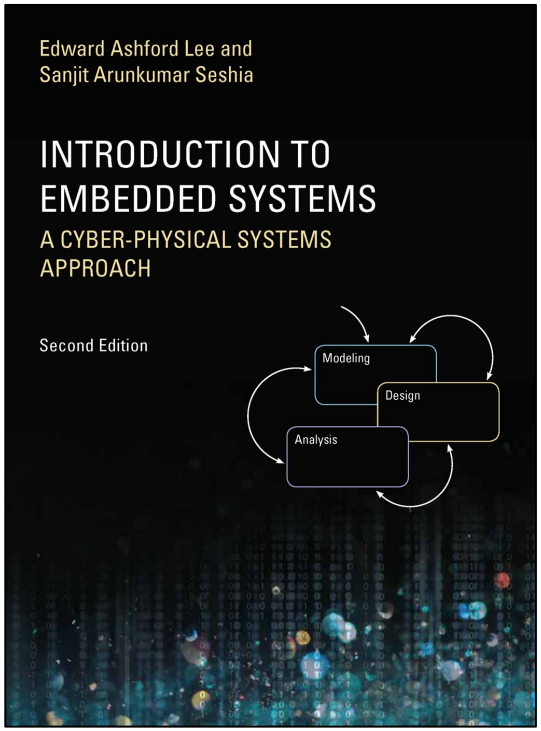


5

Composition of State Machines

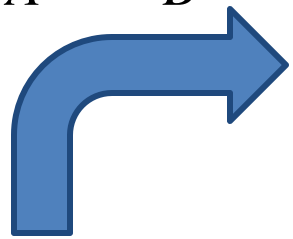
State machines provide a convenient way to model behaviors of systems. One disadvantage that they have is that for most interesting systems, the number of states is very large, often even infinite. Automated tools can handle large state spaces, but humans have more difficulty with any direct representation of a large state space.

A time-honored principle in engineering is that complicated systems should be described as compositions of simpler systems. This chapter gives a number of ways to do this with

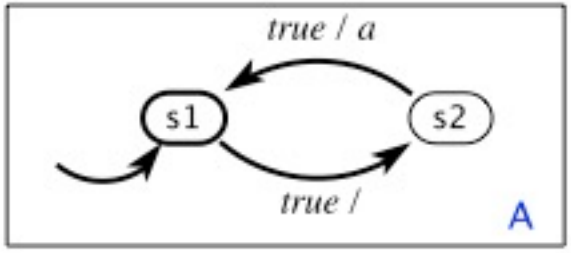


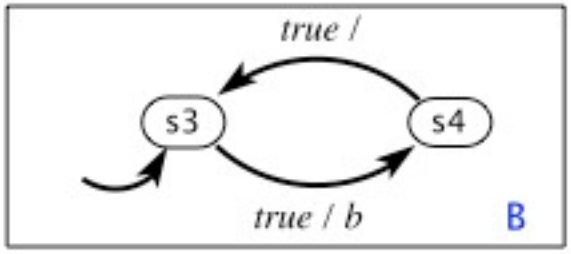
Recall Synchronous Composition:

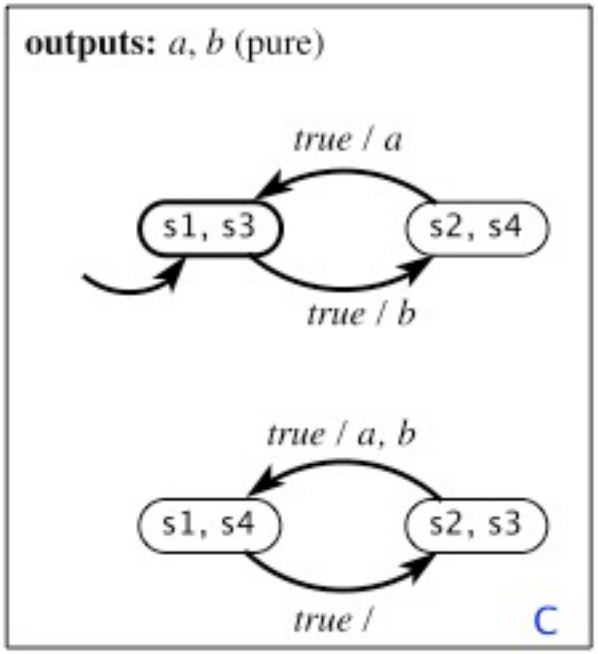
$$S_C = S_A \times S_B$$



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



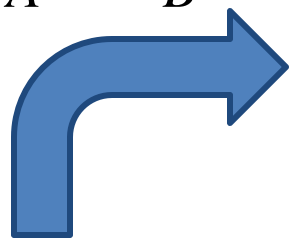




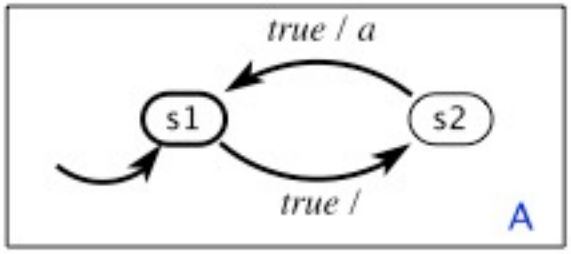
Synchronous composition

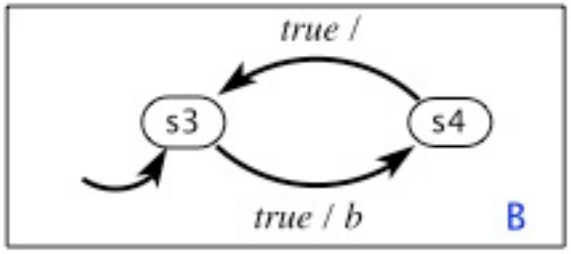
Recall Asynchronous Composition:

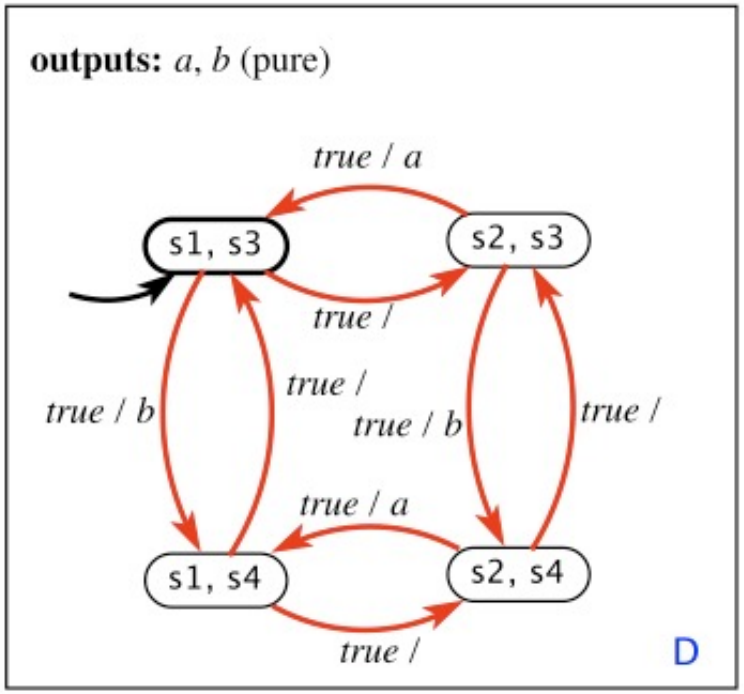
$$S_C = S_A \times S_B$$



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







Asynchronous composition with interleaving semantics

A 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
  }
}
```

Position in the program is part of the state

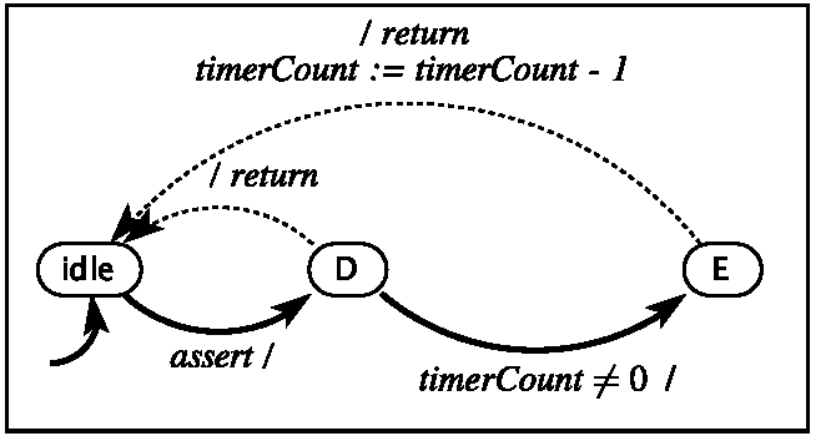
```
volatile uint timerCount = 0;
void ISR(void) {
   … disable interrupts
D → if(timerCount != 0) {
E →    timerCount--;
   }
   … enable interrupts
}
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
A → while(timerCount != 0) {
B →  …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

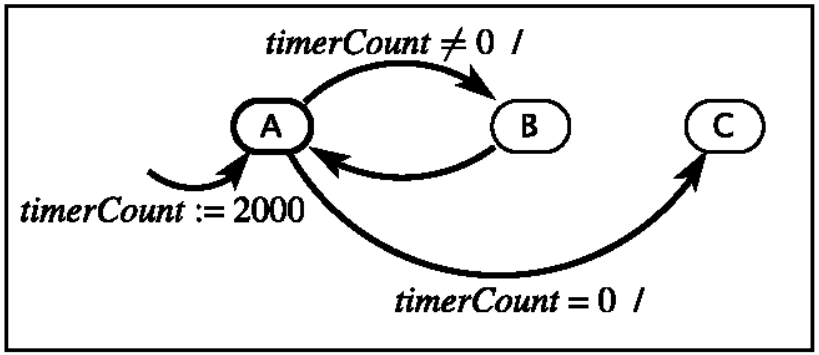
A key question: Assuming interrupt can occur infinitely often, is position C always reached?

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
    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 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?

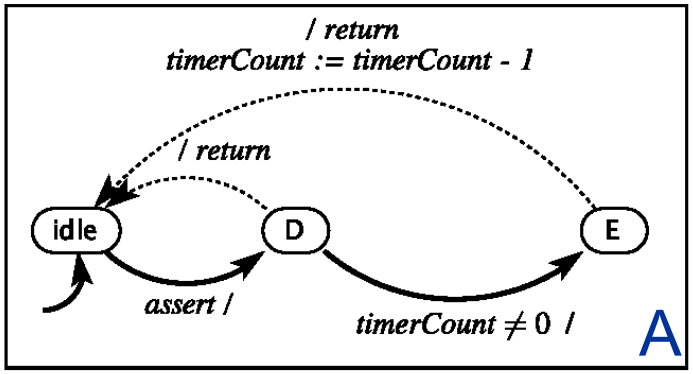
Is asynchronous composition (with interleaving semantics) the right model for this?

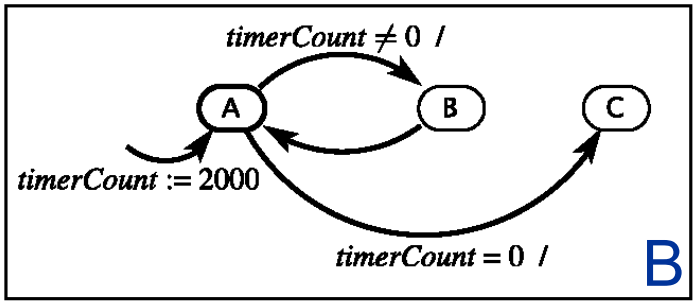
Answer: no to both.

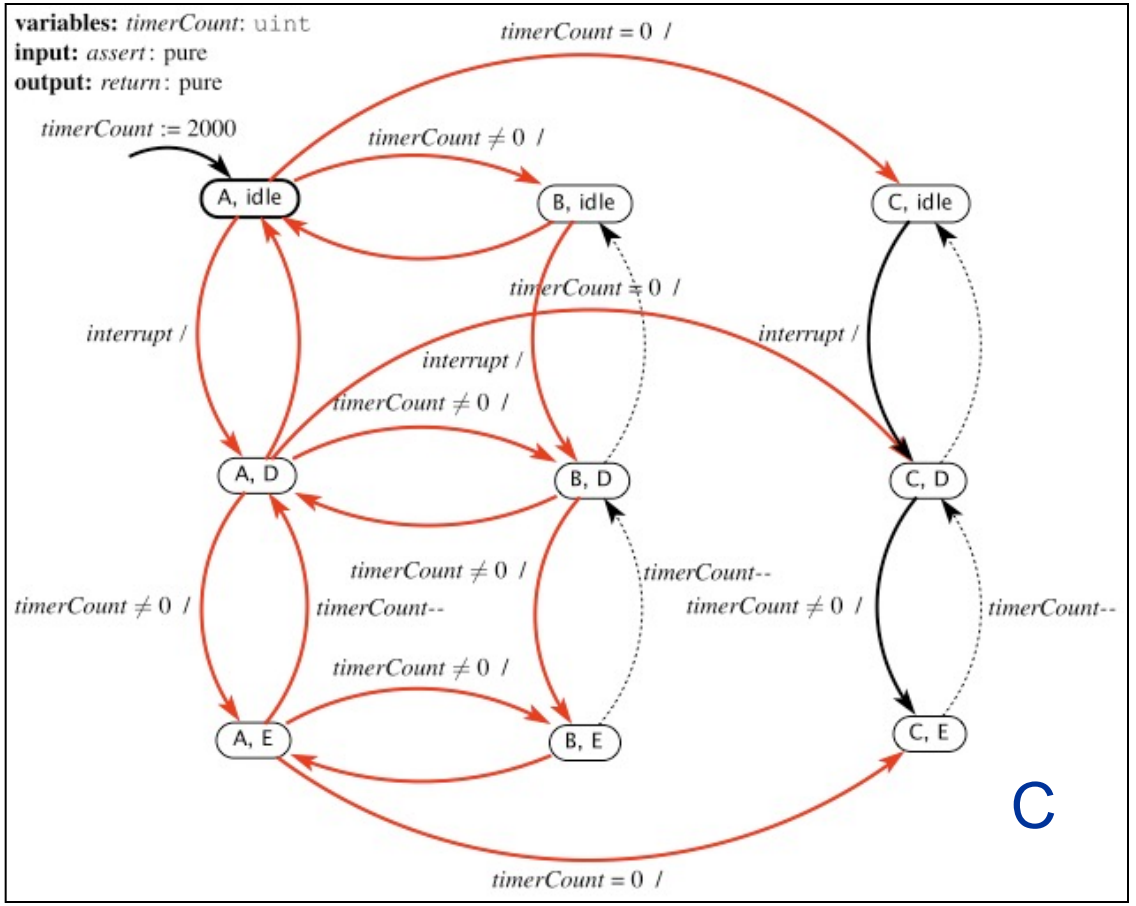
Asynchronous composition

$$S_C = S_A \times S_B$$

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





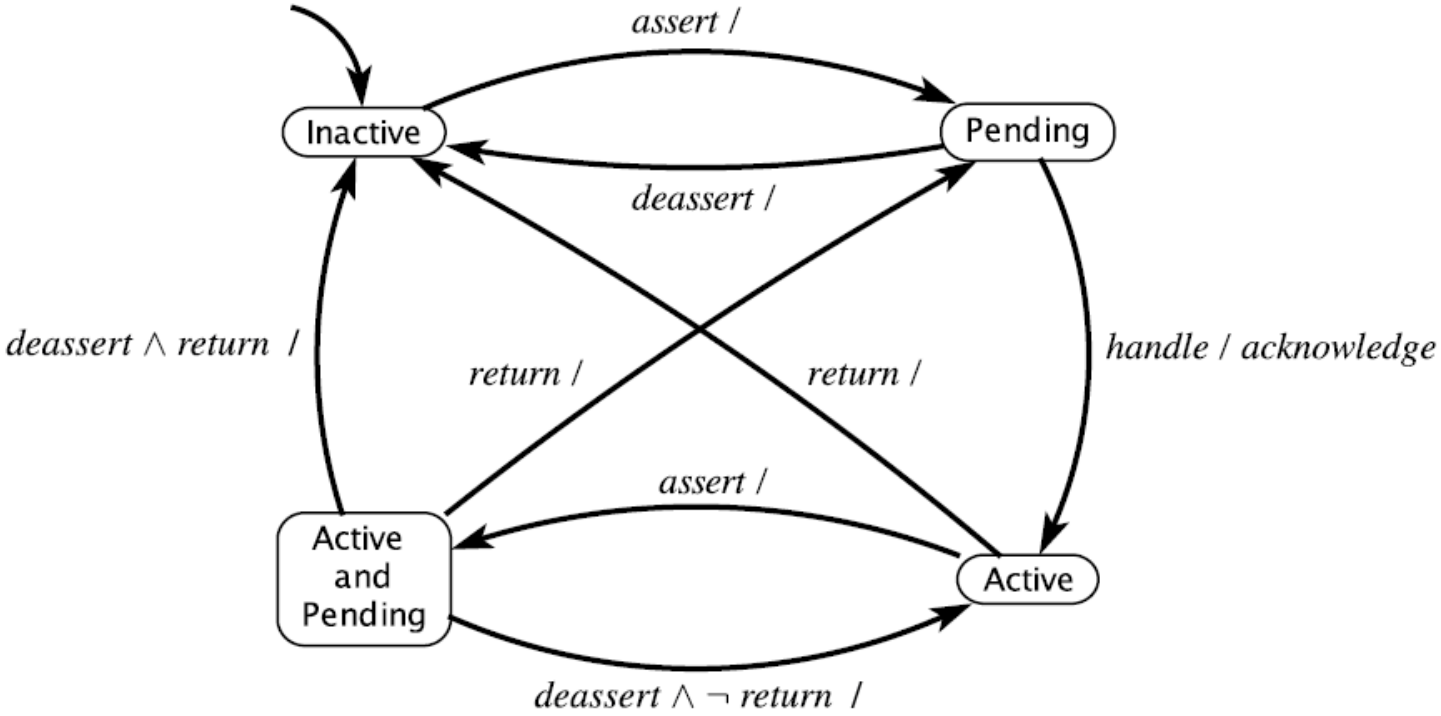


This has transitions that will not occur in practice, such as A,D to B,D. Interrupts have priority over application code.

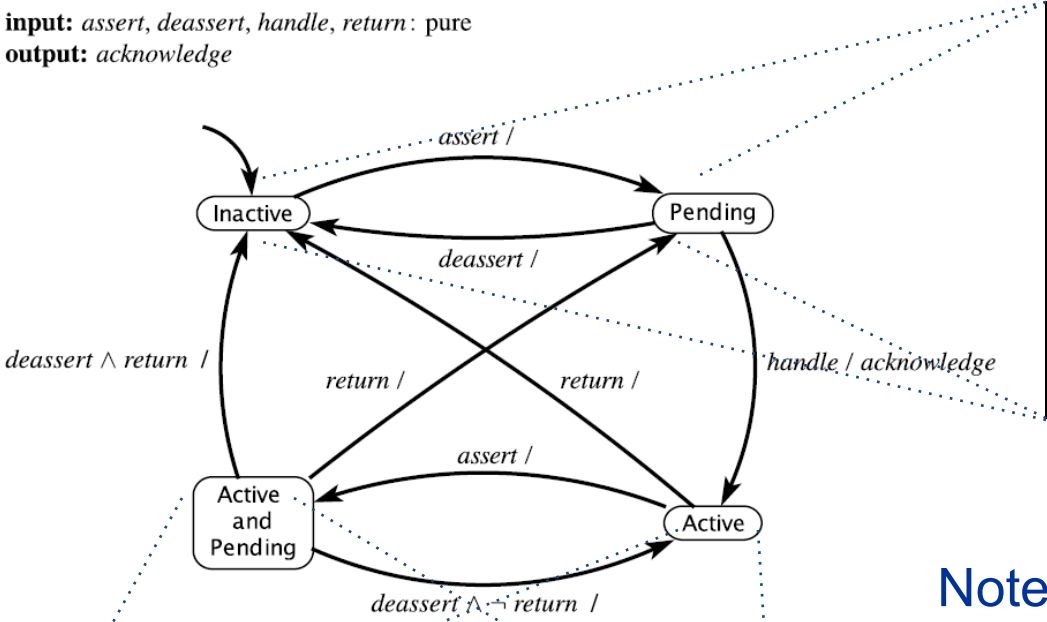
Modeling an interrupt controller

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

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



Modeling an interrupt controller

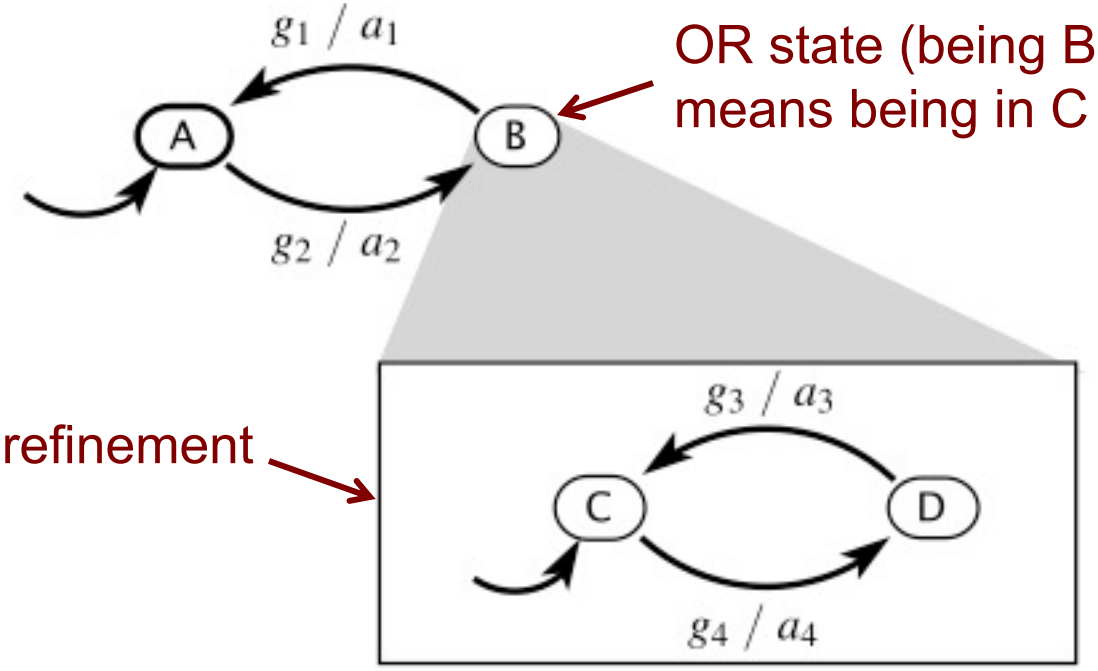



```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

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

Note that states can share refinements.

Hierarchical State Machines



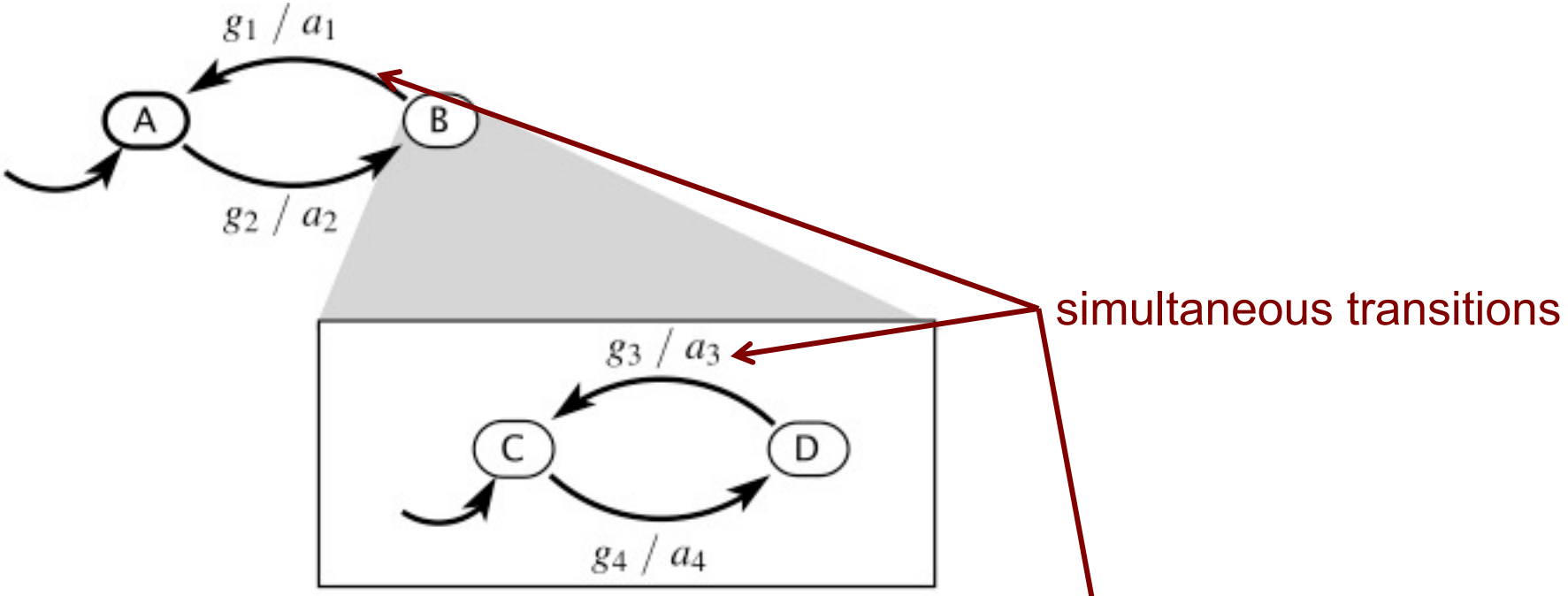
Reaction:

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

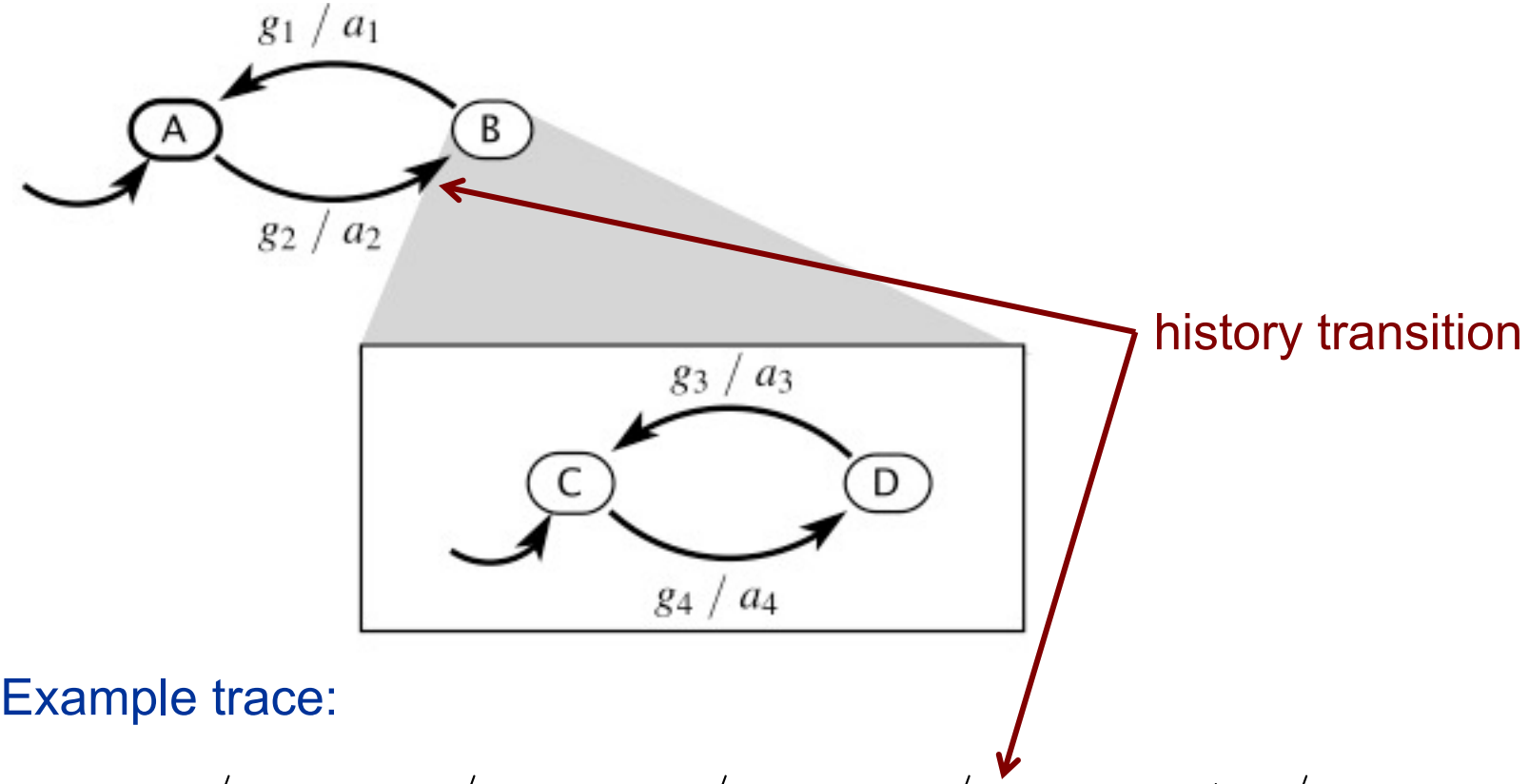


Example trace:

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

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

Hierarchical State Machines



Example trace:

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

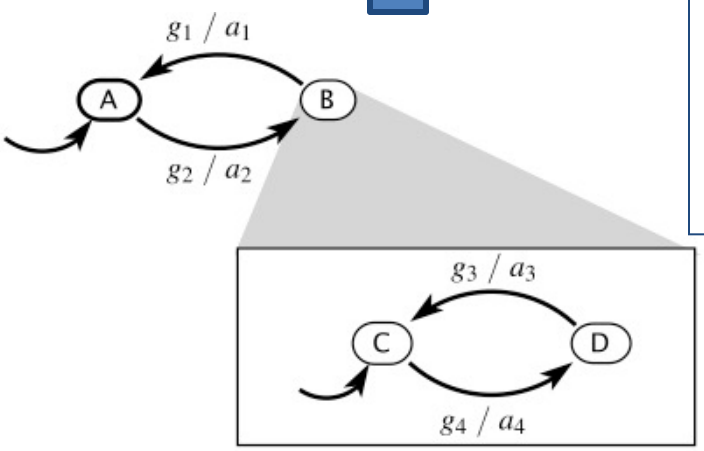
A history transition implies that when a state with a refinement is left, it is nonetheless necessary to remember the state of the refinement.

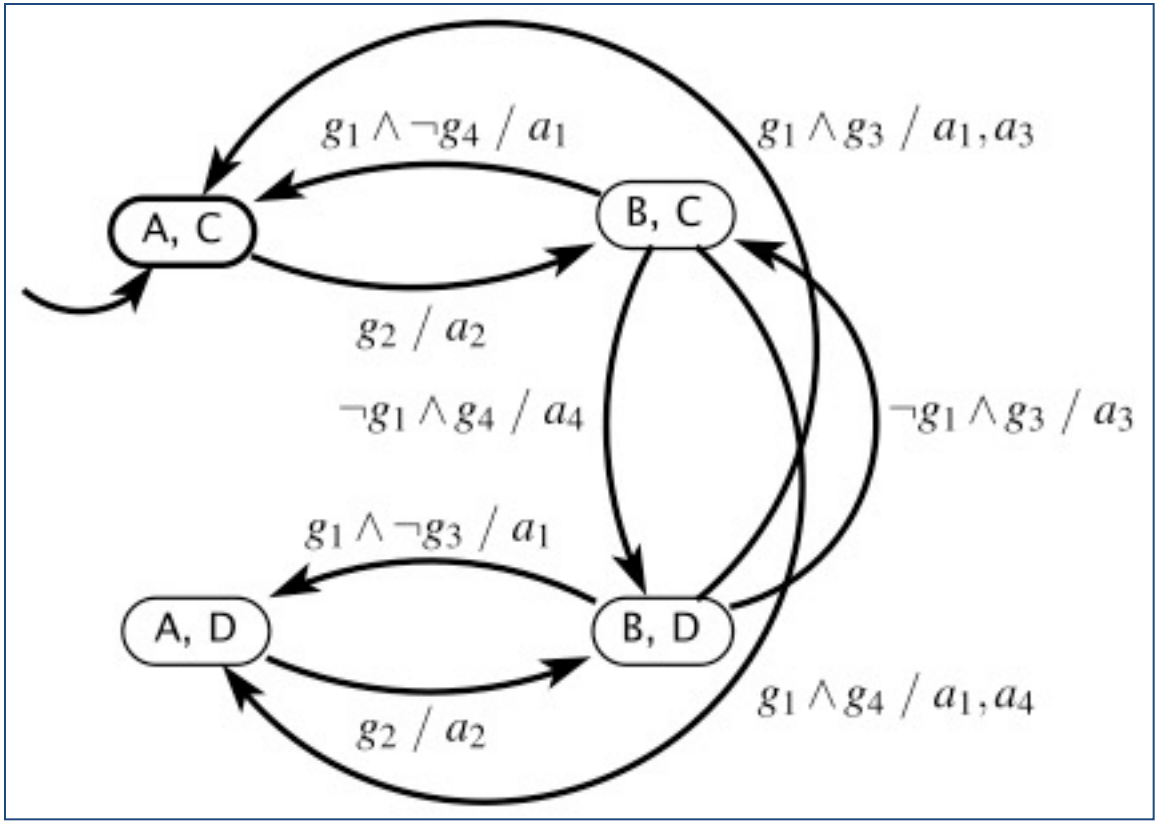
Equivalent Flattened State Machine

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

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

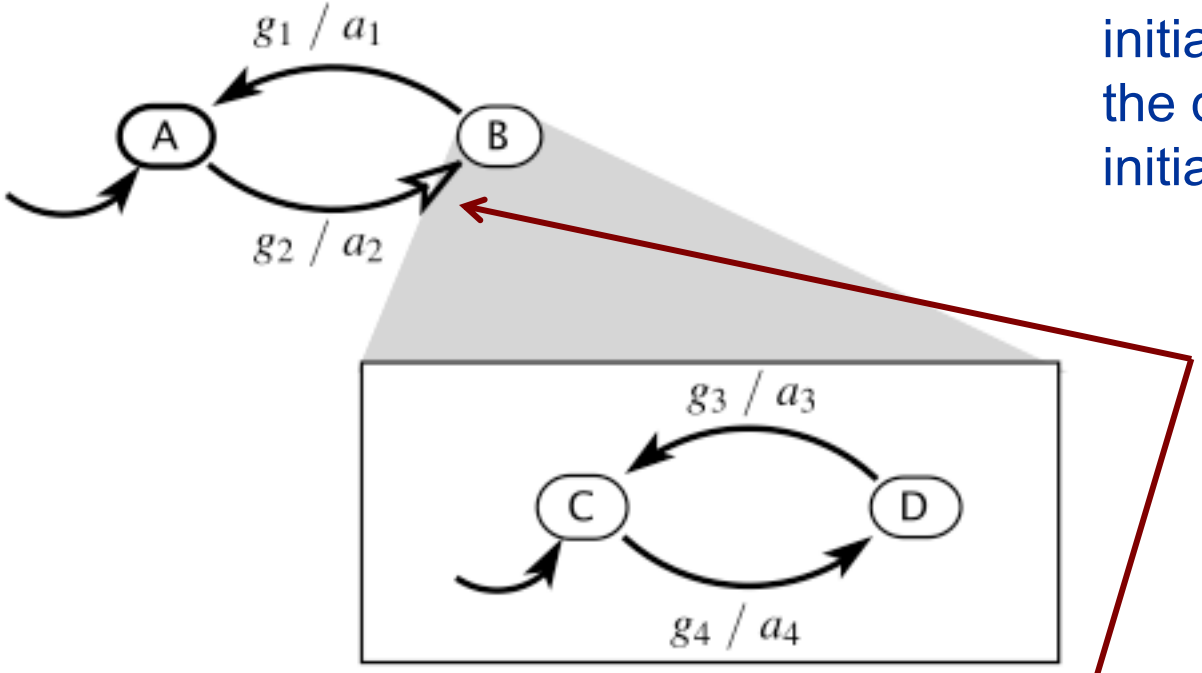
Flattening the state machine (assuming history transitions):





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 initialises the refinement of the destination state to its initial state.

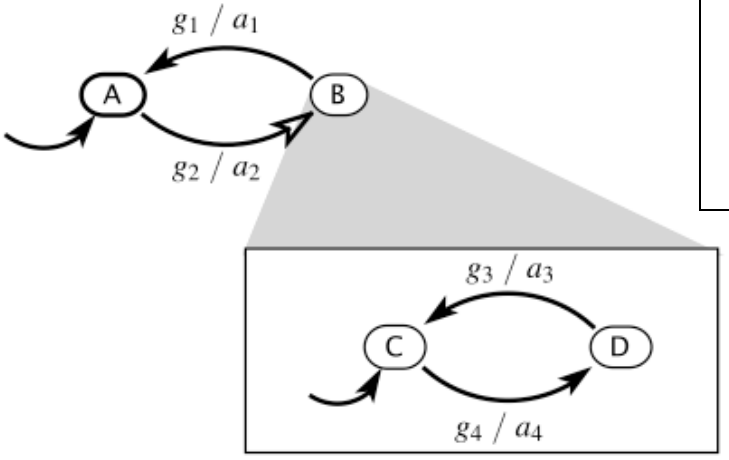
reset transition

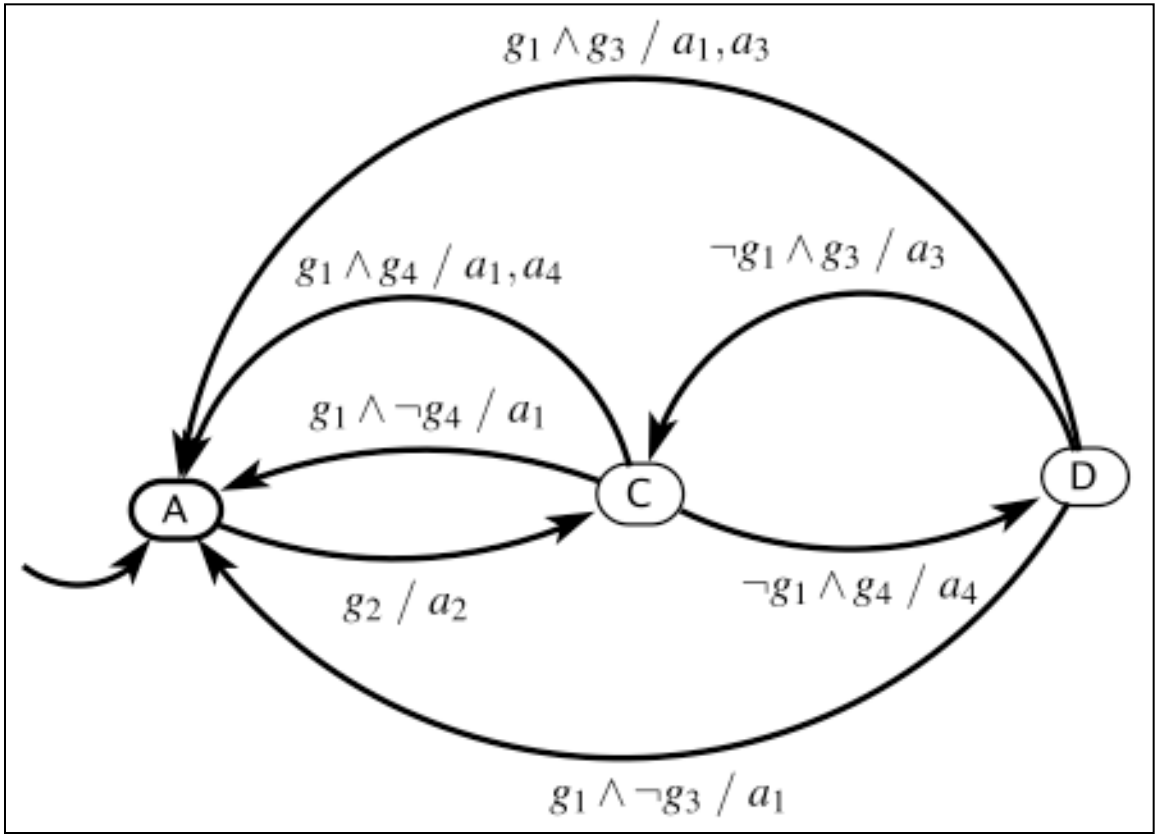
Example trace:

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

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

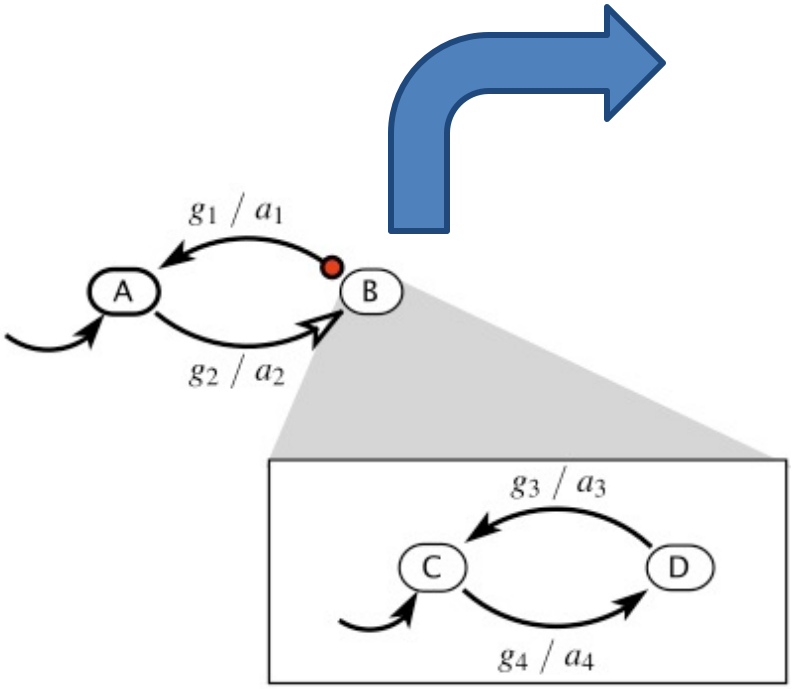
Flattening the state machine (assuming reset transitions):

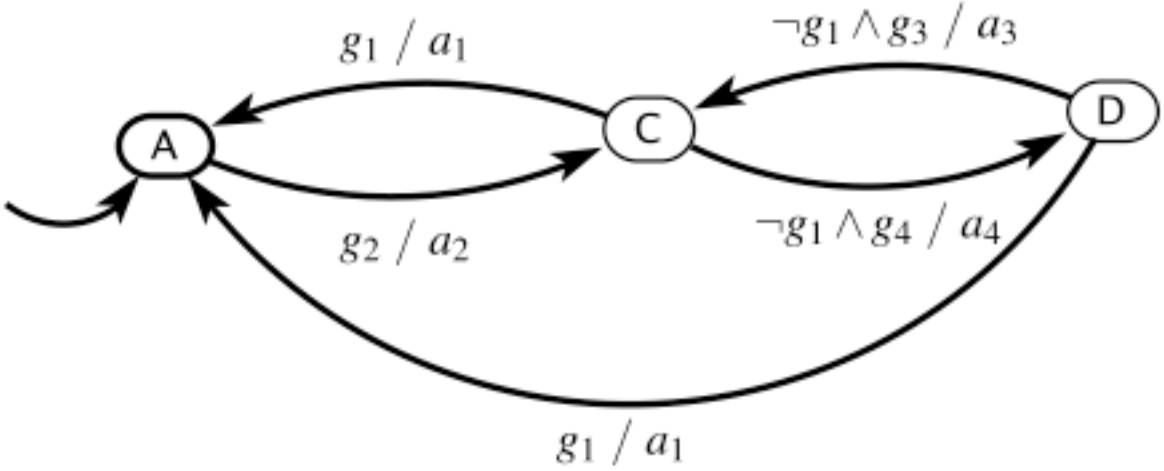




A reset transition implies that when a state with a refinement is left, it is not necessary to remember the state of the refinement. Hence there are fewer states.

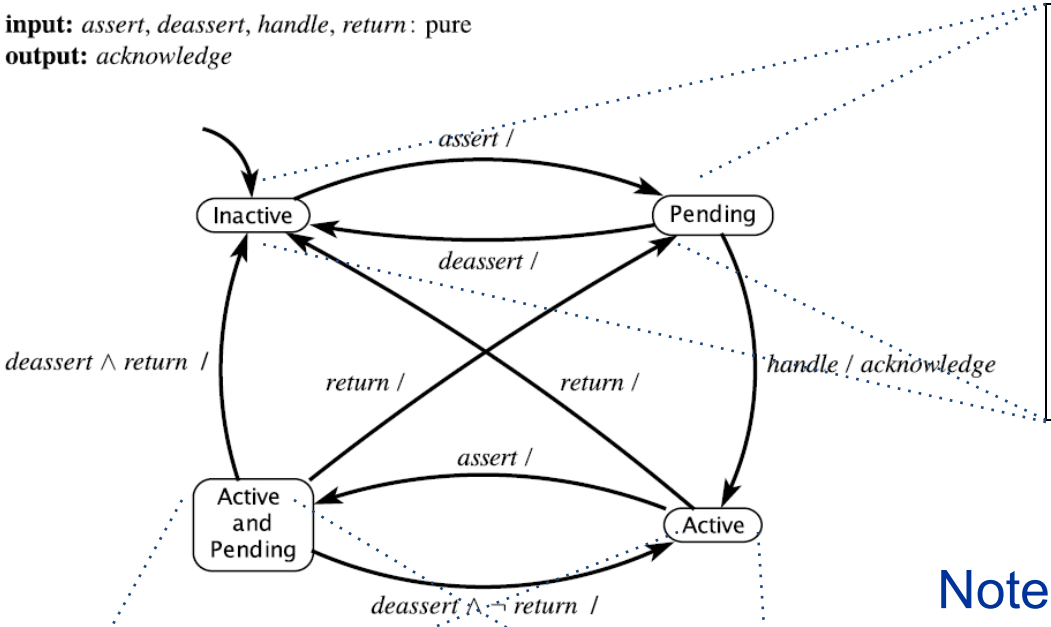
Preemptive Transitions





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



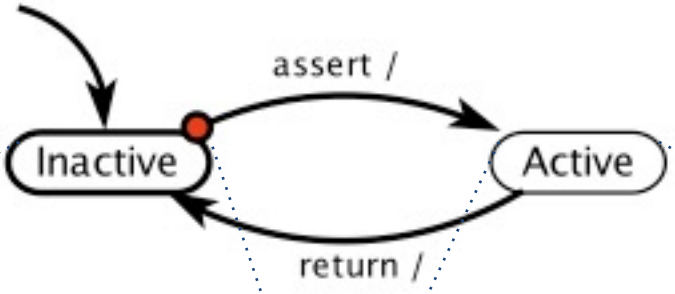

```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

```
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:




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

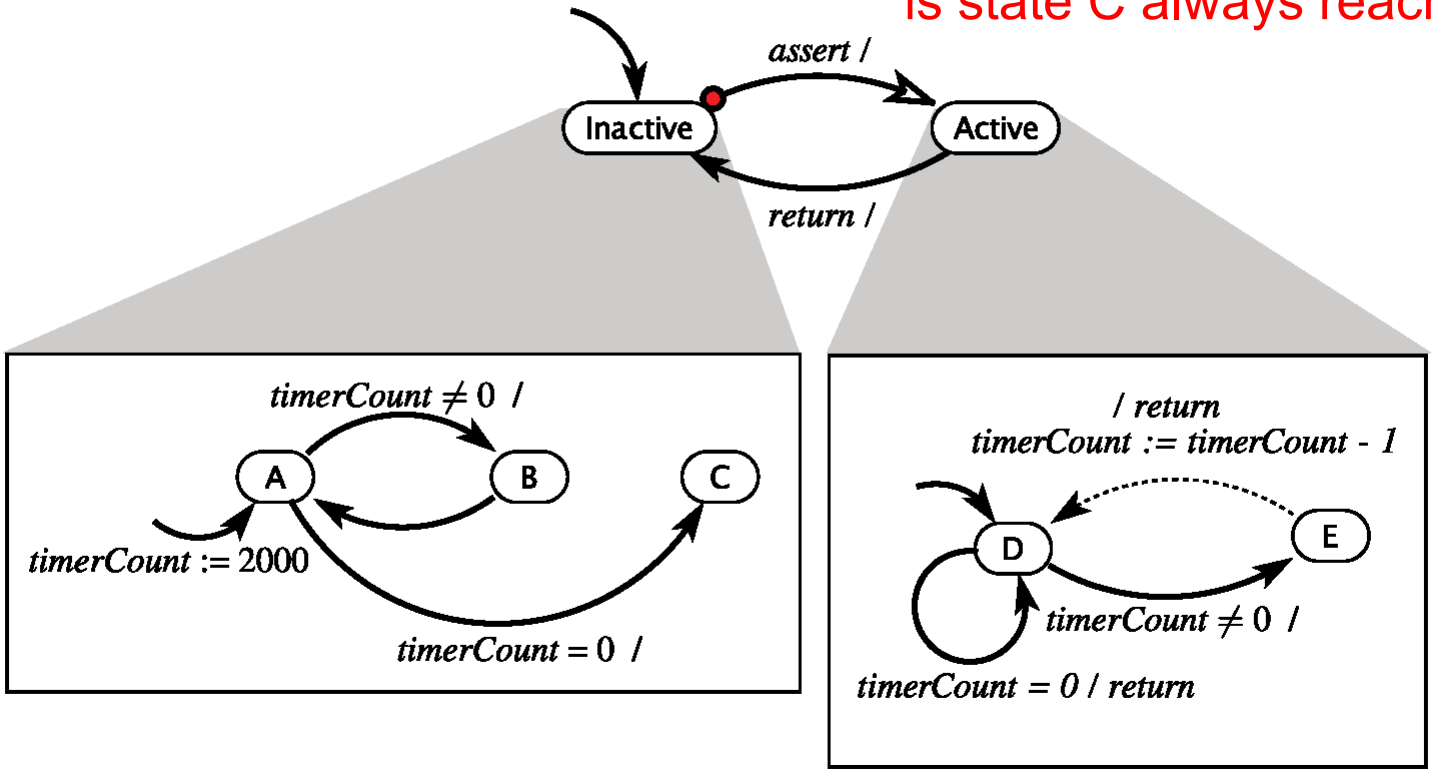
```
volatile uint timerCount = 0;
void ISR(void) {
D  … disable interrupts
   if(timerCount != 0) {
E     timerCount--;
   }
   … enable interrupts
}
```

Hierarchical interrupt controller

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

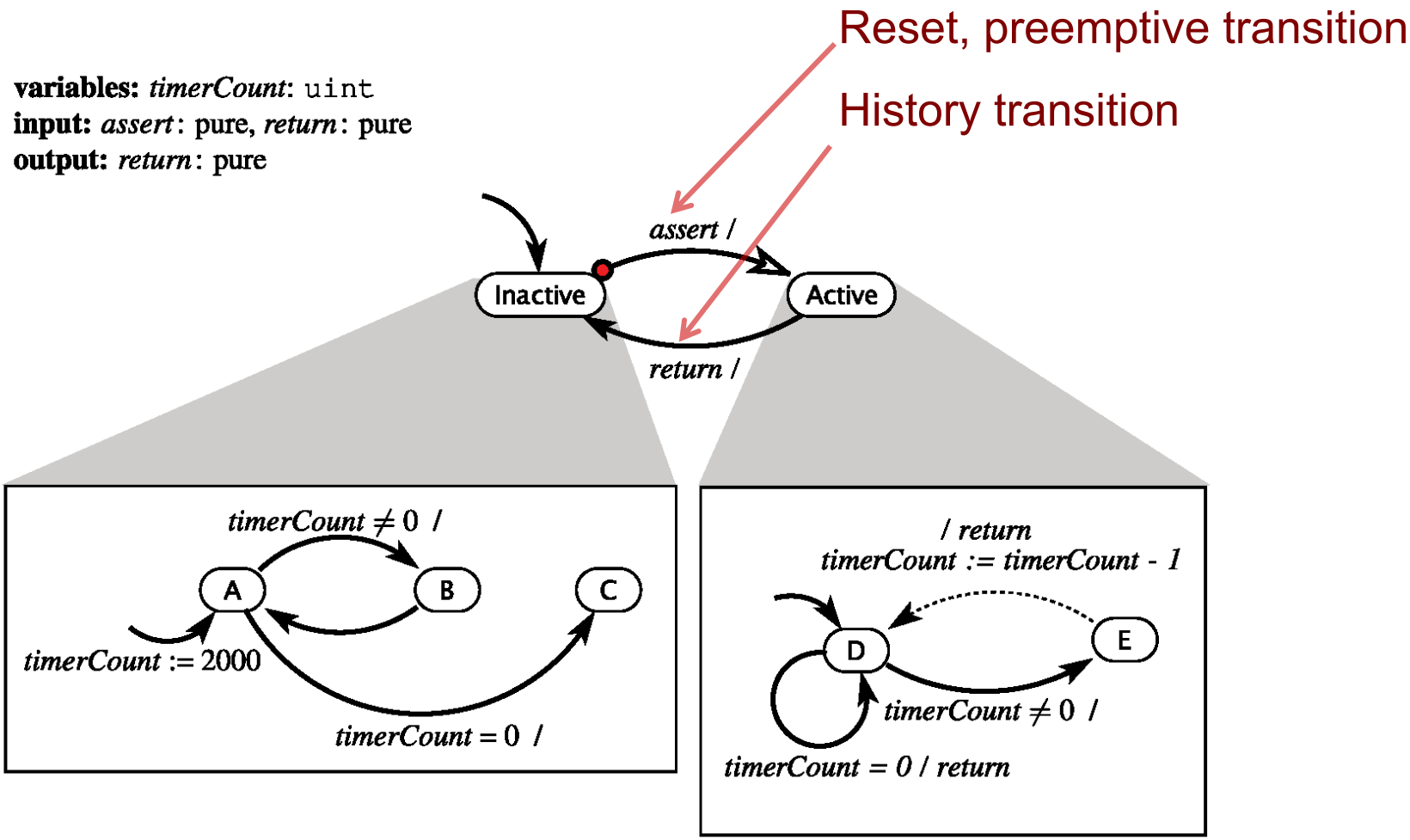
variables: *timerCount*: `uint`
input: *assert*: pure, *return*: pure
output: *return*: pure

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



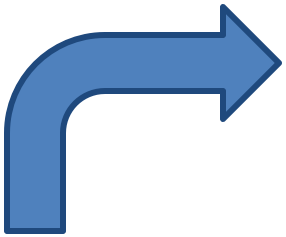
Hierarchical interrupt controller

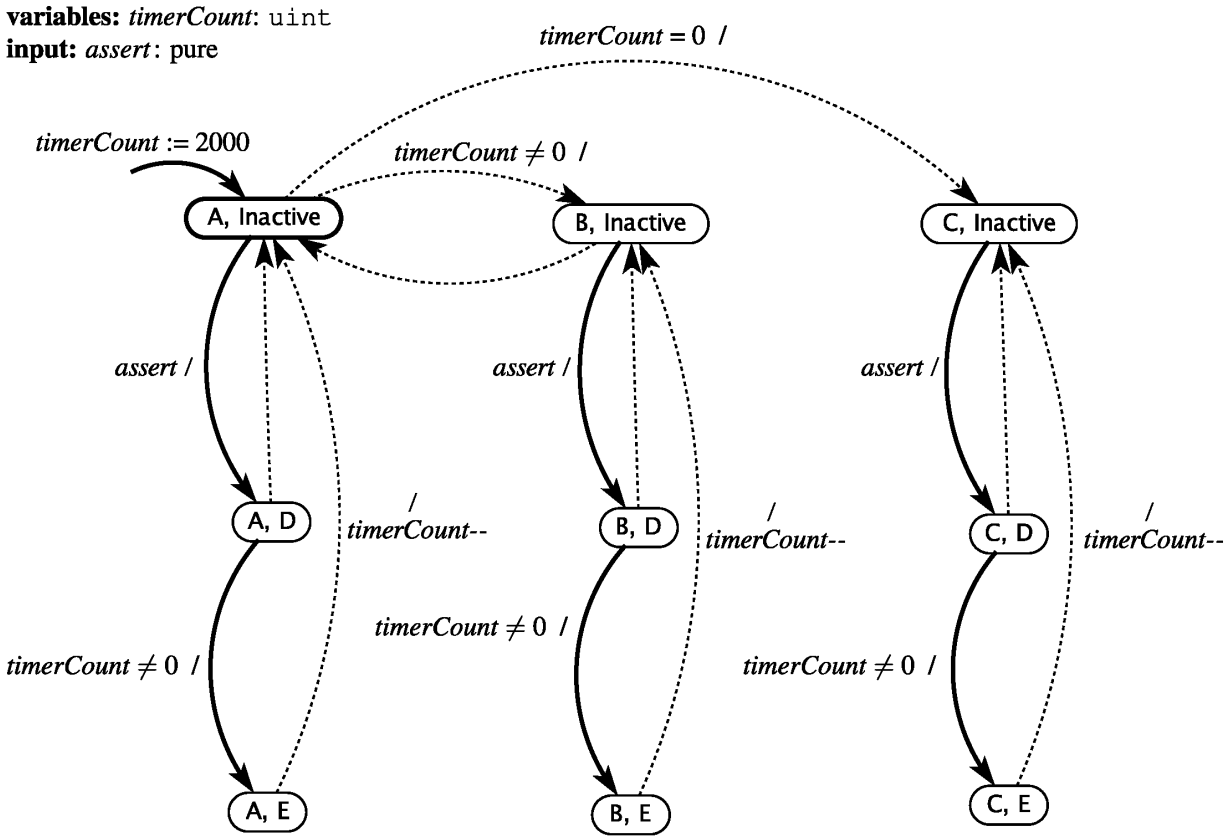
- This model assumes interrupts are disabled in the ISR:

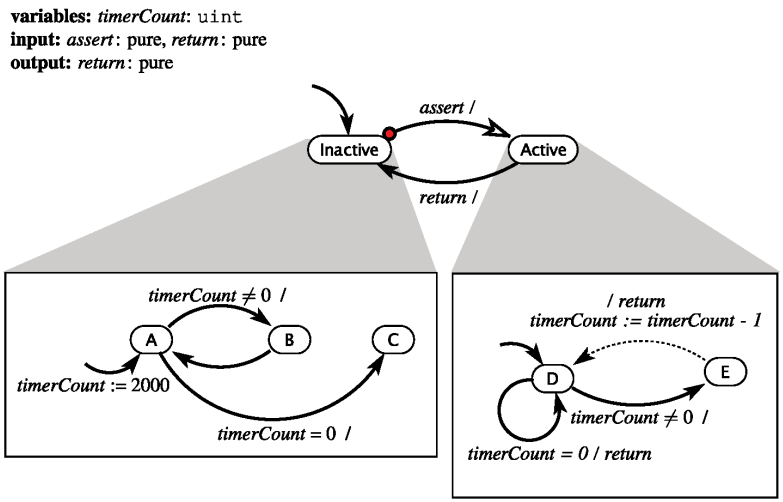


Hierarchical composition to model interrupts

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



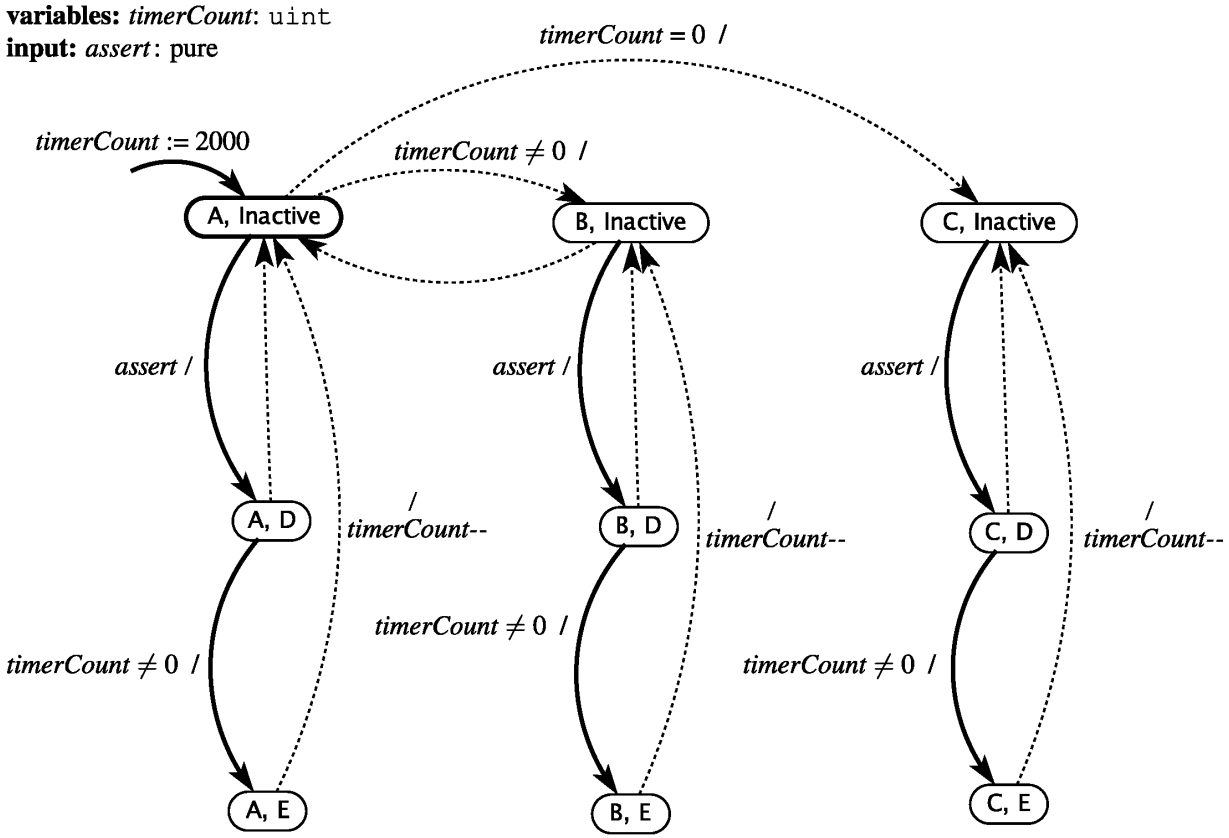


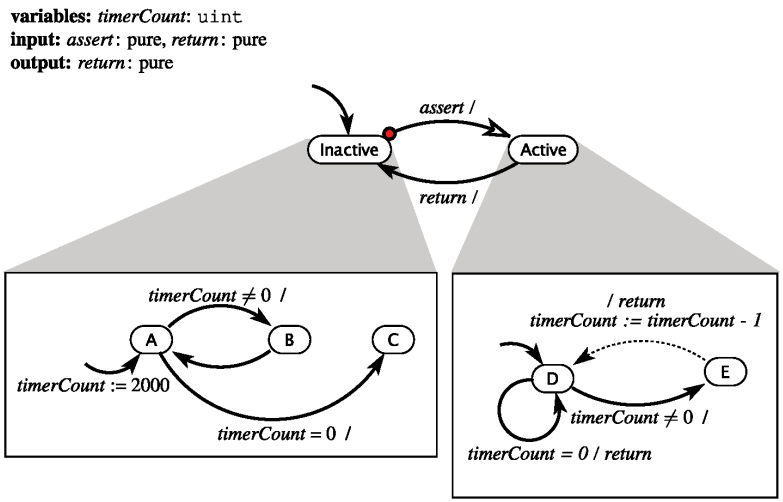


Examining this composition machine, it is clear that C is not necessarily reached if the interrupt occurs infinitely often. If assert is present on every reaction, C is never reached.

Hierarchical composition to model interrupts

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





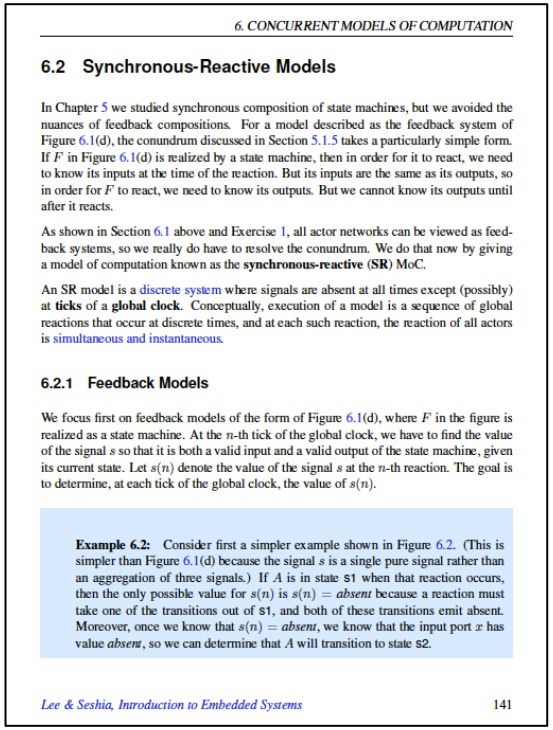
Under what assumptions/model of "assert" would C be reached?

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 analyse by hand.
- Algorithmic techniques are needed to analyse large state spaces (e.g., *reachability analysis* and *model checking*, see Chapter 13 of Lee & Seshia).

Things to do …

- If you haven't already done, read Chapter 6.2
- Read over Workshop 5 for next week



6. CONCURRENT MODELS OF COMPUTATION

6.2 Synchronous-Reactive Models

In Chapter 5 we studied synchronous composition of state machines, but we avoided the nuances of feedback compositions. For a model described as the feedback system of Figure 6.1(d), the conundrum discussed in Section 5.1.5 takes a particularly simple form. If F in Figure 6.1(d) is realized by a state machine, then in order for it to react, we need to know its inputs at the time of the reaction. But its inputs are the same as its outputs, so in order for F to react, we need to know its outputs. But we cannot know its outputs until after it reacts.

As shown in Section 6.1 above and Exercise 1, all actor networks can be viewed as feedback systems, so we really do have to resolve the conundrum. We do that now by giving a model of computation known as the **synchronous-reactive (SR)** MoC.

An SR model is a discrete system where signals are absent at all times except (possibly) at **ticks** of a **global clock**. Conceptually, execution of a model is a sequence of global reactions that occur at discrete times, and at each such reaction, the reaction of all actors is simultaneous and instantaneous.

6.2.1 Feedback Models

We focus first on feedback models of the form of Figure 6.1(d), where F in the figure is realized as a state machine. At the n-th tick of the global clock, we have to find the value of the signal s so that it is both a valid input and a valid output of the state machine, given its current state. Let $s(n)$ denote the value of the signal s at the n-th reaction. The goal is to determine, at each tick of the global clock, the value of $s(n)$.

Example 6.2: Consider first a simpler example shown in Figure 6.2. (This is simpler than Figure 6.1(d) because the signal s is a single pure signal rather than an aggregation of three signals.) If A is in state `S1` when that reaction occurs, then the only possible value for $s(n)$ is $s(n) = absent$ because a reaction must take one of the transitions out of `S1`, and both of these transitions emit absent. Moreover, once we know that $s(n) = absent$, we know that the input port x has value *absent*, so we can determine that A will transition to state `S2`.

Lee & Seshia, Introduction to Embedded Systems 141