Composition of State Machines

Hongwei Zhang

http://www.cs.wayne.edu/~hzhang/

Ack.: this lecture is prepared in part based on slides of Lee, Sangiovanni-Vincentelli, Seshia.
Outline

Concurrent composition
Hierarchical state machines
Outline

Concurrent composition
Hierarchical state machines
Composition of State Machines

How do we construct complex state machines out of simpler “building blocks”? 

Two aspects of composition:
1. **Spatial**: how do the components communicate between each other?
2. **Temporal**: when do the components execute, relative to each other?
Temporal Composition of State Machines

Sequential vs. Concurrent

If Concurrent, Asynchronous vs. Synchronous
Hybrid Systems Provide *Sequential* Composition

Modal models: sequencing between modes

https://www.youtube.com/watch?v=iD3QgGpzzIM

[Tomlin et al.]
For concurrent composition, we need an interface: Actor Model for State Machines

Expose inputs and outputs, enabling concurrent composition:
Spatial Composition of State Machines

- Side-by-side composition
- Cascade composition
- Feedback composition
Side-by-Side Composition:
state machines being composed do not communicate

A key question: When do these machines react?

How the reactions of composed machines are coordinated is called a “Model of Computation” (MoC).
When do these machines react? Two of many possibilities:

• Together, in lock step (synchronous, concurrent composition)

• Independently (asynchronous, concurrent composition)
Synchronous Composition

\[ S_C \subseteq S_A \times S_B \]

outputs: \( a, b \) (pure)
Synchronous Composition

\[ S_C \subseteq S_A \times S_B \]

Note that these two states are not reachable.

Synchronous composition
Asynchronous Composition

\[ S_C \subseteq S_A \times S_B \]

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

Asynchronous composition using interleaving semantics
Asynchronous Composition

Note that now all states are reachable.

Asynchronous composition using interleaving semantics:
A reaction of C is a reaction of one of A or B, where the choice is nondeterministic.

\[ S_C \subseteq S_A \times S_B \]
Syntax vs. Semantics

Synchronous or Asynchronous composition?

If asynchronous, does it allow simultaneous transitions in A & B?
How to choose whether A or B reacts when C reacts (i.e., scheduling/priority)?

The answers to these questions defines the MoC being used.
Cascade Composition

Output port(s) of A connected to input port(s) of B
Example: Time-Triggered Pedestrian Light

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

\[ pcount := 0 \quad pcount \geq 55 / \text{pedR} \]

\[ pcount := pcount + 1 \]

This light stays green for 55 seconds, then goes red. Upon receiving a \( \text{sigR} \) input, it repeats the cycle.
Example: Time-Triggered Car Light

variable: count: \{0, \ldots, 60\}
inputs: pedestrian : pure
outputs: sigR, sigG, sigY : pure

\begin{itemize}
  \item count \geq 60 / sigG
  \quad count := 0
  \item pedestrian \land count < 60 /
  \quad count := count + 1
  \item count := count + 1
  \item count := 0
  \item count \geq 5 / sigR
  \quad count := 0
  \item count := count + 1
  \item count := count + 1
  \item count := count + 1
\end{itemize}
What is the size of the state space of the composite machine?
Synchronous cascade composition

unsafe states
Synchronous composition with unreachable states removed
Shared Variables: Two Servers

**shared variable:** pending: int  
**input:** request: pure  
**outputs:** doneA, doneB : pure

**input:** request: pure  
**output:** done: pure  

\[ \neg \text{request} \land \text{pending} = 0 / \text{done} \]
\[ \neg \text{request} \land \text{pending} > 0 / \text{done} \]
\[ \text{pending} := \text{pending} - 1 \]
\[ \text{request} / \]
\[ \text{pending} := \text{pending} + 1 \]

A

**input:** request: pure  
**output:** done: pure  

\[ \neg \text{request} \land \text{pending} = 0 / \text{done} \]
\[ \neg \text{request} \land \text{pending} > 0 / \text{done} \]
\[ \text{pending} := \text{pending} - 1 \]
\[ \text{request} / \]
\[ \text{pending} := \text{pending} + 1 \]

B

**input:** request: pure  
**output:** done: pure  

\[ \neg \text{request} \land \text{pending} = 0 / \text{done} \]
\[ \neg \text{request} \land \text{pending} > 0 / \text{done} \]
\[ \text{pending} := \text{pending} - 1 \]
\[ \text{request} / \]
\[ \text{pending} := \text{pending} + 1 \]

C
Feedback Composition

Reasoning about feedback composition can be very subtle. (more about this later)
Spatial Composition of State Machines

Side-by-side composition

Cascade composition

Feedback composition
Outline

Concurrent composition
Hierarchical state machines
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

```c
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
    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
    }
    whatever comes next
}

A key question: Assuming interrupt can occur infinitely often, is position C always reached?
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
    }
    ... whatever comes next
}

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

Is asynchronous composition the right thing to do here?
Asynchronous composition

\[ S_C = S_A \times S_B \]

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
    }
}
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
    }
}
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

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

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

Note that states can share refinements.
Hierarchical State Machines

Reaction: (one of many possible models)
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:

\[ A \xrightarrow{g_2/a_2} C \xrightarrow{g_4/a_4} D \xrightarrow{g_1/a_1} A \xrightarrow{g_2/a_2} D \xrightarrow{g_3 \land g_1/a_3, a_1} A \ldots \]

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

Example trace:

\[
\begin{align*}
A & \xrightarrow{g_2/a_2} C \xrightarrow{g_4/a_4} D \xrightarrow{g_1/a_1} A \xrightarrow{g_2/a_2} D \xrightarrow{g_3 \land g_1/a_3, a_1} A \cdots
\end{align*}
\]

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.
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:

\[
A \xrightarrow{g_2/a_2} C \xrightarrow{g_4/a_4} \quad D \xrightarrow{g_1/a_1} \quad A \xrightarrow{g_2/a_2} \quad C \xrightarrow{g_4 \land g_1/a_4,a_1} \quad A \ldots
\]

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.
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 refinement 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
    }
}

Note that states can share refinements.

volatile uint timerCount = 0;
void ISR(void) {
    ... disable interrupts
    if(timerCount != 0) {
        timerCount--;
    }
    ... enable interrupts
}
Simplified interrupt controller

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

- **variables:** \textit{timerCount}: uint
- **input:** \textit{assert}: pure, \textit{return}: pure
- **output:** \textit{return}: pure

![Model Diagram]

- **Transition Labels:**
  - Reset, preemptive transition
  - History transition

- **State Diagram:**
  - \textit{Inactive} to \textit{Active} with \textit{assert}
  - \textit{Active} to \textit{Inactive} with \textit{return}
  - \textit{timerCount \neq 0} / transition
  - \textit{timerCount := 2000}
  - \textit{timerCount := timerCount - 1}
  - \textit{timerCount = 0} /
  - \textit{return}
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).
Summary

- Composition enables building complex systems from simpler ones.
  - Concurrent composition
  - Hierarchical state machines: 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).
Assignment

Exercise #7

- Chapter 5: 4, 5, 6