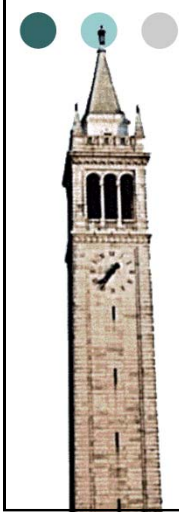


Introduction to Embedded Systems

Sanjit A. Seshia
UC Berkeley
EECS 149/249A
Fall 2015

Chapter 6 - Models of Computation: Synchronous/Reactive and Dataflow

Concurrent Composition: Alternatives to Threads

Threads yield incomprehensible behaviors.

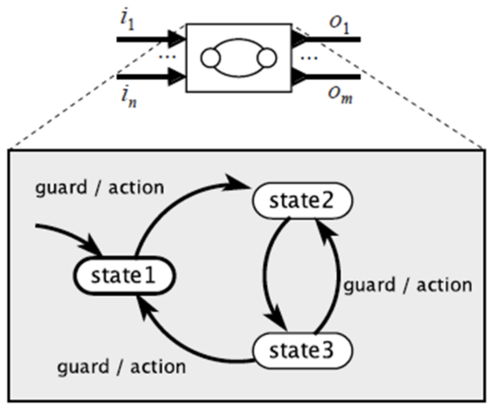
Composition of State Machines:

- Side-by-side composition
- Cascade composition
- Feedback composition

We will begin with synchronous composition, an abstraction that has been very effectively used in hardware design and is gaining popularity in software design.

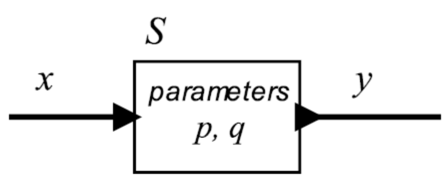
Recall: Actor Model for State Machines

Expose inputs and outputs, enabling composition:



Recall: Actor Model of Continuous-Time Systems

A *system* is a function that accepts an input *signal* and yields an output signal.



The domain and range of the system function are sets of signals, which themselves are functions.

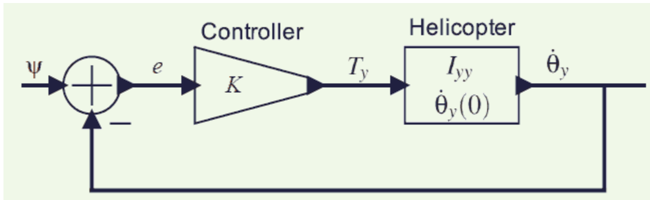
$$x: \mathbb{R} \to \mathbb{R}, \quad y: \mathbb{R} \to \mathbb{R}$$

$$S: X \to Y$$

$$X = Y = (\mathbb{R} \to \mathbb{R})$$

Parameters may affect the definition of the function *S*.

Recall: Composition of Actors

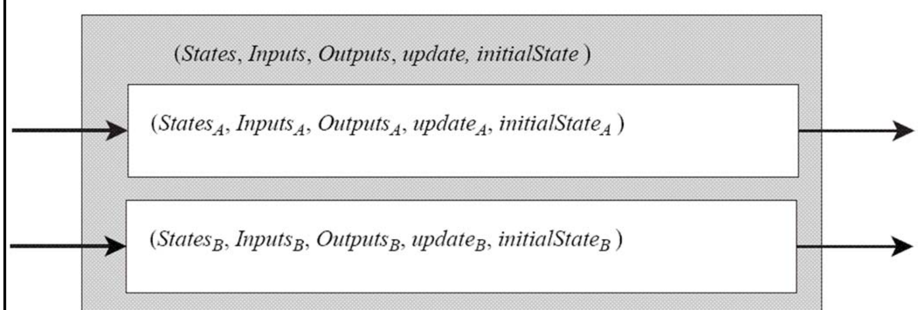


$$\begin{aligned}\dot{\theta}_y(t) &= \dot{\theta}_y(0) + \frac{1}{I_{yy}}\int_0^t T_y(\tau)d\tau \\ &= \dot{\theta}_y(0) + \frac{K}{I_{yy}}\int_0^t (\psi(\tau) - \dot{\theta}_y(\tau))d\tau\end{aligned}$$

Angular velocity appears on both sides. The *semantics* (meaning) of the model is the solution to this equation.

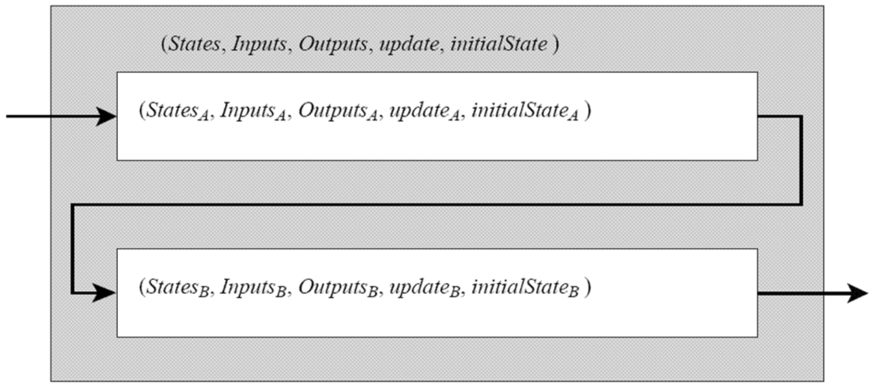
We will now generalize this notion of composition.

Side-by-Side Composition



Synchronous composition: the machines react simultaneously and instantaneously.

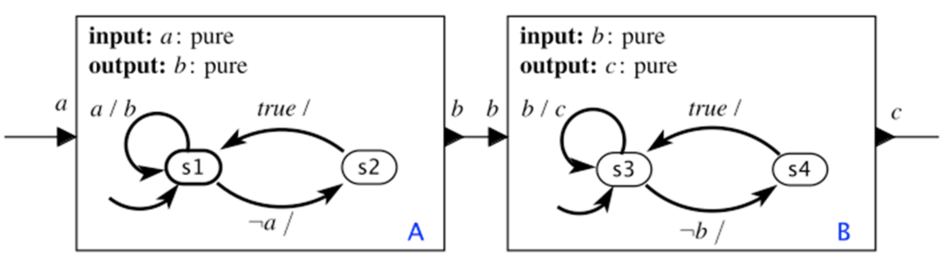
Cascade Composition



Synchronous composition: the machines react simultaneously and instantaneously, despite the apparent causal relationship!

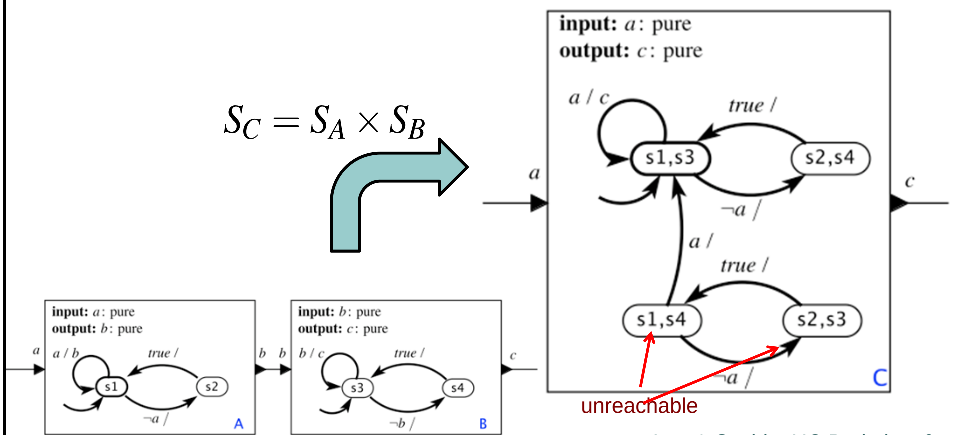
Synchronous Composition: Reactions are *Simultaneous* and *Instantaneous*

Consider a cascade composition as follows:

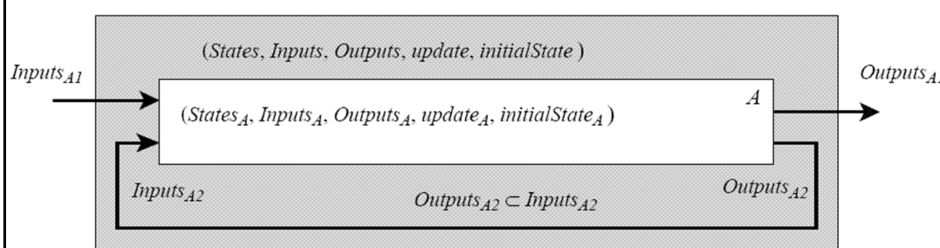


Synchronous Composition: Reactions are *Simultaneous* and *Instantaneous*

In this model, you must not think of machine A as reacting before machine B. If it did, the unreachable states would not be unreachable.

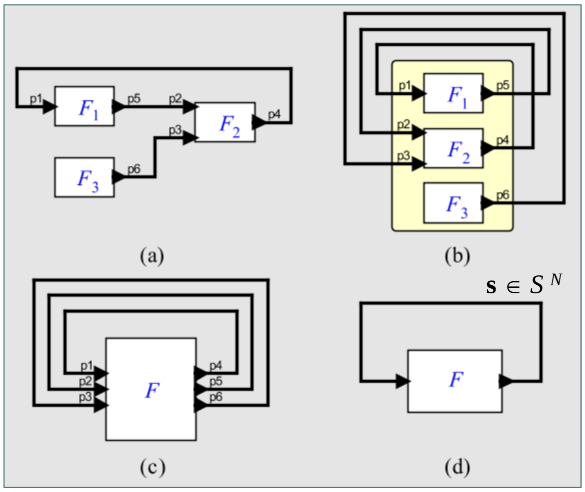


Feedback Composition



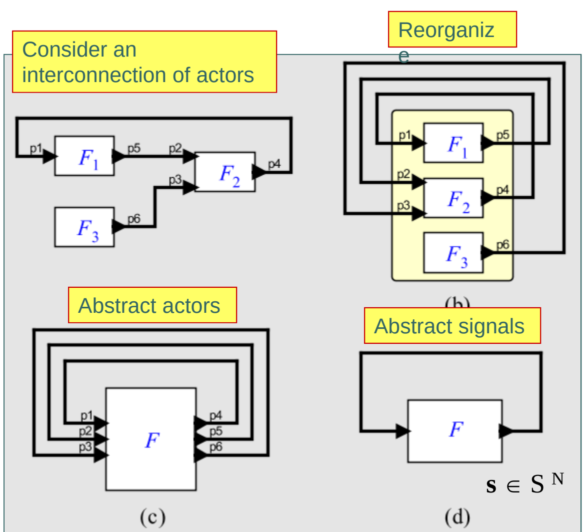
Turns out everything can be viewed as feedback composition…

Observation: Any Composition is a Feedback Composition



The behavior of the system is a "fixed point."

Fixed Point Semantics



We seek an $\mathbf{s} \in S^N$ that satisfies $F(\mathbf{s}) = \mathbf{s}$.

Such an **s** is called a *fixed point*.

We would like the fixed point to exist and be unique. And we would like a constructive procedure to find it.

It is the *behavior* of the system.

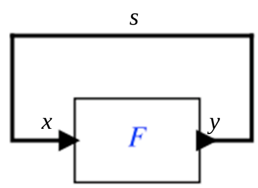
Data Types

As with any connection, we require compatible data types:

$$V_y \subseteq V_x$$

Then the signal on the feedback loop is a function

$$s\colon \mathbb{N} \to V_y \cup \{absent\}$$



Then we seek s such that

$$F(s) = s$$

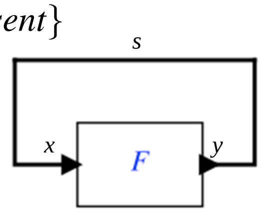
where F is the actor function, which for determinate systems has form

$$F\colon (\mathbb{N} \to V_x \cup \{absent\}) \to (\mathbb{N} \to V_y \cup \{absent\})$$

Firing Functions

With synchronous composition of determinate state machines, we can break this down by reaction. At the n-th reaction, there is a (state-dependent) function

$$f(n)\colon V_x \cup \{absent\} \to V_y \cup \{absent\}$$



such that

$$s(n) = (f(n))(s(n))$$

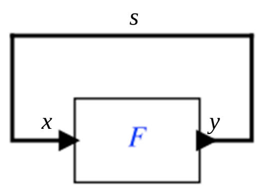
This too is a fixed point.

Well-Formed Feedback

At the n-th reaction, we seek $s(n) \in V_y \cup \{absent\}$ such that

$$s(n) = (f(n))(s(n))$$

There are two potential problems:

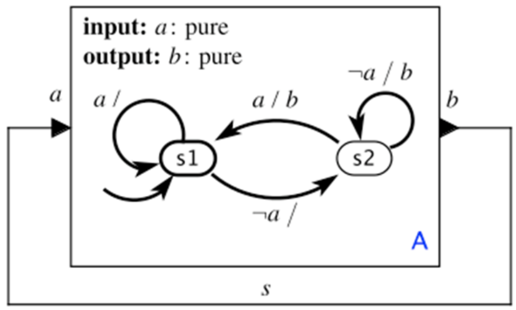


1. It does not exist.
2. It is not unique.

In either case, we call the system **ill formed**. Otherwise, it is **well formed**.

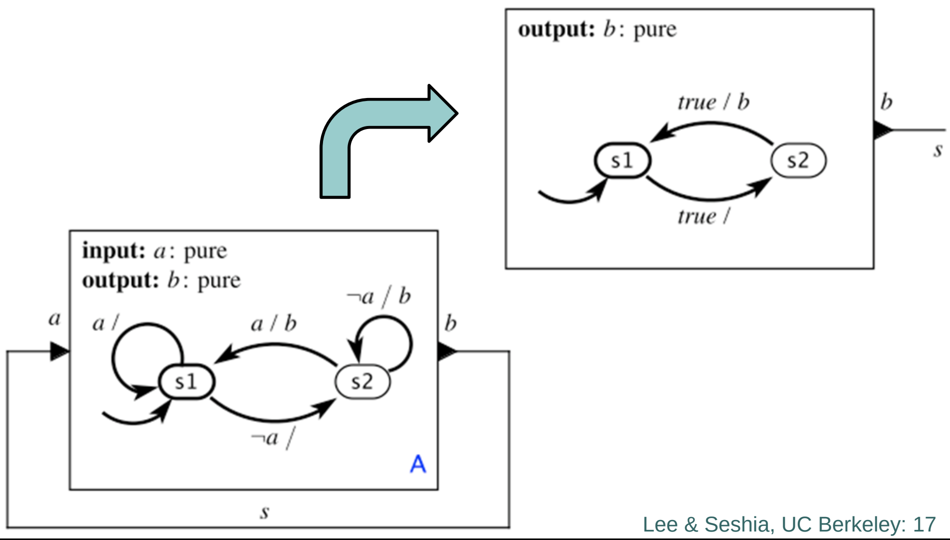
Note that if a state is not reachable, then it is irrelevant to determining whether the machine is well formed.

Well-Formed Example

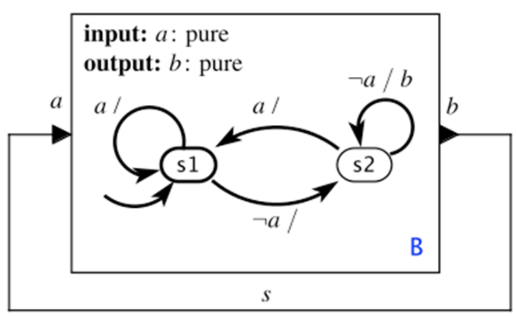


In state s1, we get the unique $s(n) = absent$.
In state s2, we get the unique $s(n) = present$.
Therefore, s alternates between *absent* and *present*.

Composite Machine

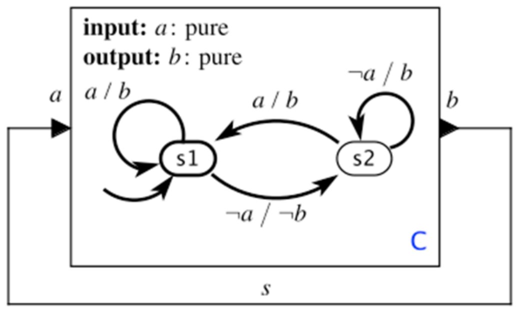


Ill-Formed Example 1 (Existence)



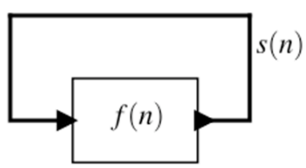
In state s1, we get the unique $s(n) = absent$.
In state s2, there is no fixed point.
Since state s2 is reachable, this composition is ill formed.

Ill-Formed Example 2 (Uniqueness)



In s1, both $s(n) = absent$ and $s(n) = present$ are fixed points.
In state s2, we get the unique $s(n) = present$.
Since state s1 is reachable, this composition is ill formed.

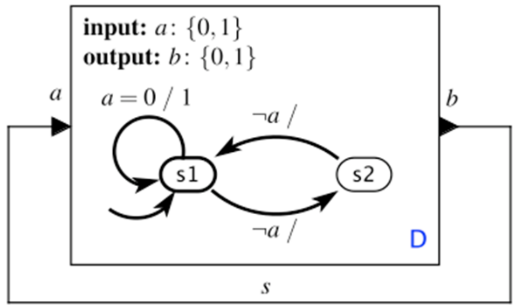
Constructive Semantics: Single Signal



1. Start with $s(n)$ *unknown*.
2. Determine as much as you can about $(f(n))(s(n))$.
3. If $s(n)$ becomes known (whether it is present, and if it is not pure, what its value is), then we have a unique fixed point.

A state machine for which this procedure works is said to be **constructive**.

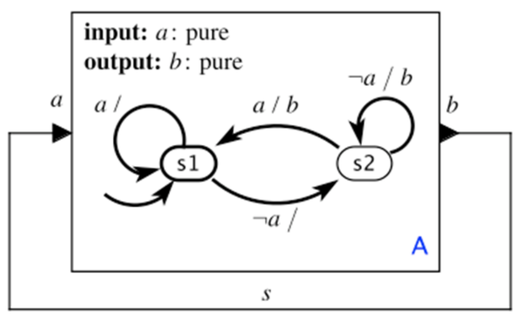
Non-Constructive Well-Formed State Machine



In state s1, if the input is unknown, we cannot immediately tell what the output will be. We have to try all the possible values for the input to determine that in fact $s(n) = absent$ for all n.

For non-constructive machines, we are forced to do **exhaustive search**. This is only possible if the data types are finite, and is only practical if the data types are small.

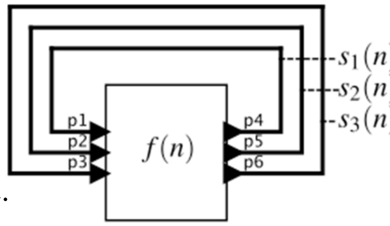
Must / May Analysis



For the above constructive machine, in state s1, we can immediately determine that the machine *may not* produce an output. Therefore, we can immediately conclude that the output is *absent*, even though the input is unknown.

In state s2, we can immediately determine that the machine *must* produce an output, so we can immediately conclude that the output is *present*.

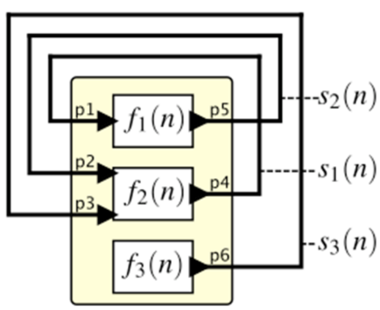
Constructive Semantics: Multiple Signals



1. Start with $s_1(n), \cdots, s_N(n)$ *unknown*.
2. Determine as much as you can about $(f(n))(s_1(n), \cdots, s_N(n))$.
3. Using new information about $s_1(n), \cdots, s_N(n)$, repeat step (2) until no information is obtained.
4. If $s_1(n), \cdots, s_N(n)$ all become known, then we have a unique fixed point and a constructive machine.

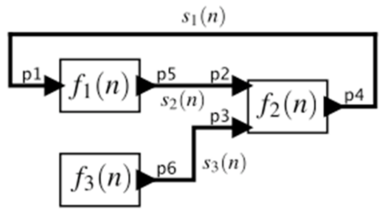
A state machine for which this procedure works is said to be **constructive**.

Constructive Semantics: Multiple Actors



Procedure is the same.

Constructive Semantics: Arbitrary Structure



Procedure is the same.

A state machine language with constructive semantics will reject all compositions that in any iteration fail to make all signals known.

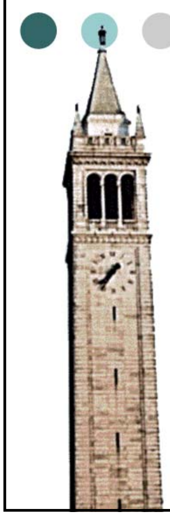
Such a language rejects some well-formed compositions.

Synchronous Reactive Models: Conclusions

The emphasis of synchronous composition, in contrast with threads, is on *determinate* and *analyzable* concurrency.

Although there are subtleties with synchronous programs, all constructive synchronous programs have a unique and well-defined meaning.

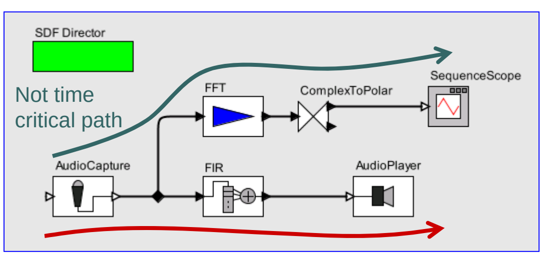
Automated tools can systematically explore *all* possible behaviors. This is not possible in general with threads.



Introduction to Embedded Systems

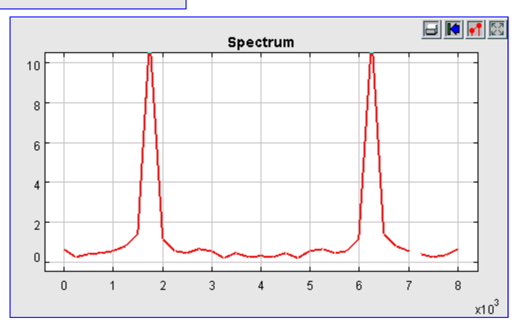
Chapter 6: Dataflow Models

Simple Example: Spectrum Analysis

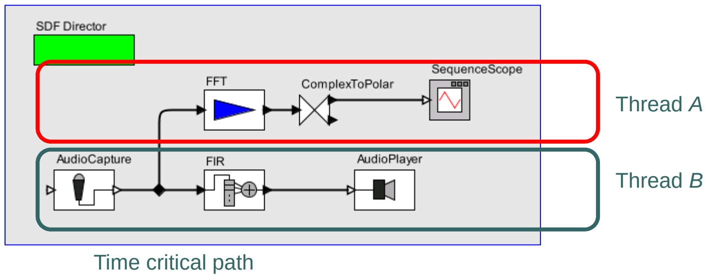


Time critical path

How do we keep the non-time critical path from interfering with the time-critical path?



A Solution with Threads



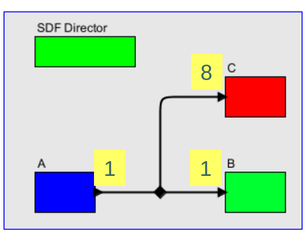
Create two threads:

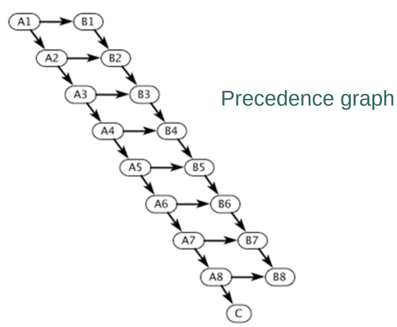
- A has low priority
- B has high priority

Why?

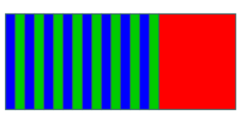
- RMS does not apply because there are dependencies.
- EDF with precedences applies and is optimal w.r.t. feasibility, except for how to assign deadlines.
- How to implement the communication between threads?

Abstracted Version of the Spectrum Example: EDF scheduling



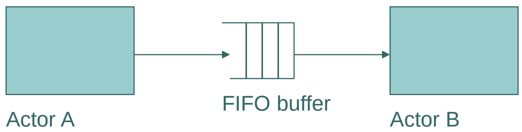


Suppose that C requires 8 data values from A to execute. Suppose further that C takes much longer to execute than A or B. EDF schedule:



schedule

Dataflow Models



Buffered communication between concurrent components (*actors*).

Static scheduling: Assign to each thread a sequence of actor invocations (*firings*) and repeat forever.

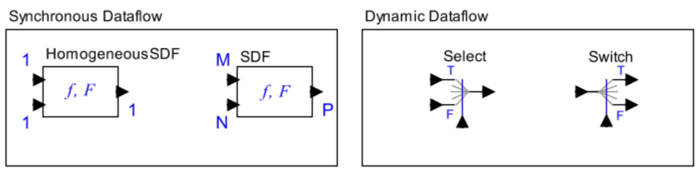
Dynamic scheduling: Each time dispatch() is called, determine which actor can fire (or is firing) and choose one.

May need to implement interlocks in the buffers.

Streams: The basis for Dataflow models

A stream is a signal $x \colon \mathbb{N} \to R$, for some set R. There is not necessarily any relationship between $x(n)$, an element in a stream, and $y(n)$, an element in another stream. Unlike discrete-time models or SR models, they are not "simultaneous."

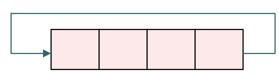
Dataflow



Each signal has form $x : \mathbb{N} \rightarrow R$. The function F maps such signals into such signals. The function f (the "firing function") maps prefixes of these signals into prefixes of the output. Operationally, the actor *consumes* some number of tokens and *produces* some number of tokens to construct the output signal(s) from the input signal(s). If the number of tokens consumed and produced is a constant over all firings, then the actor is called a *synchronous dataflow* (SDF) actor.

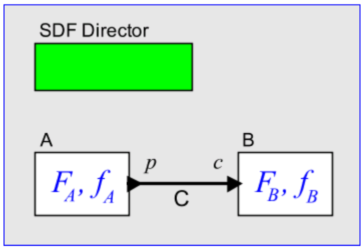
Firing rules: the number of tokens required to fire an actor.

Buffers for Dataflow



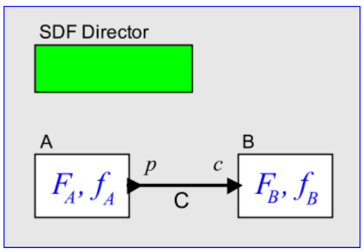
- Unbounded buffers require memory allocation and deallocation schemes.
- Bounded size buffers can be realized as *circular buffers* or *ring buffers,* in a statically allocated array.
  - A *read pointer* r is an index into the array referring to the first empty location. Increment this after each read.
  - A *fill count* n is unsigned number telling us how many data items are in the buffer.
  - The next location to write to is $(r + n)$ modulo buffer length.
  - The buffer is empty if $n == 0$
  - The buffer is full if $n ==$ buffer length
  - Can implement n as a semaphore, providing mutual exclusion for code that changes n or r.

Synchronous Dataflow (SDF)



If the number of tokens consumed and produced by the firing of an actor is constant, then static analysis can tell us whether we can schedule the firings to get a useful execution, and if so, then a finite representation of a schedule for such an execution can be created.

Balance Equations



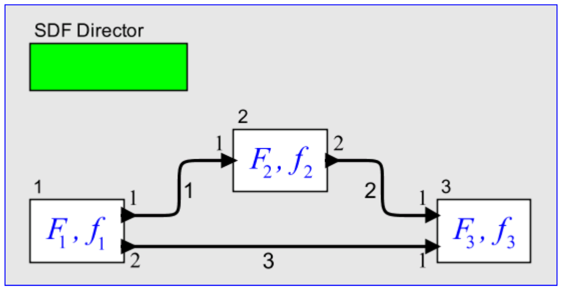
Let q_A, q_B be the number of firings of actors A and B.
Let p_C, c_C be the number of tokens produced and consumed on a connection C.
Then the system is *in balance* if for all connections C

$$q_A p_C = q_B c_C$$

where A produces tokens on C and B consumes them.

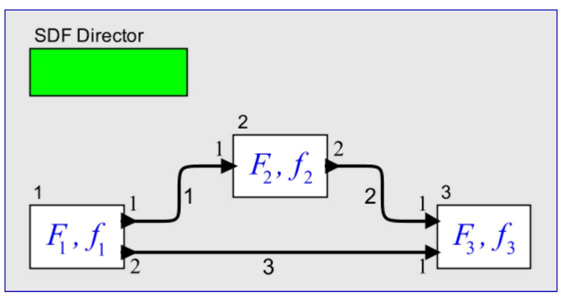
Example

Consider this example, where actors and arcs are numbered:



The balance equations imply that actor 3 must fire twice as often as the other two actors.

Compactly Representing the Balance Equations



production/consumption matrix

Connector 1

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix}$$

Actor 1

$$q = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

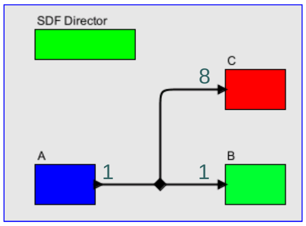
firing vector

balance equations

$$\Gamma q = \vec{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

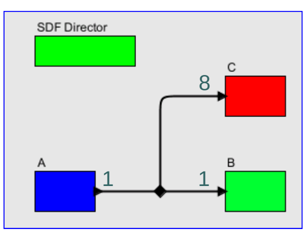
Question on initial example …

What is the production/consumption matrix in this case?



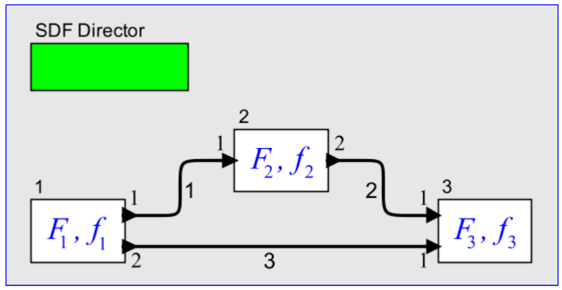
Question on initial example …

What is the production/consumption matrix in this case?



$$\Gamma = \begin{bmatrix} 1 & 0 & -1 \\ 1 & -8 & 0 \end{bmatrix}$$

Example

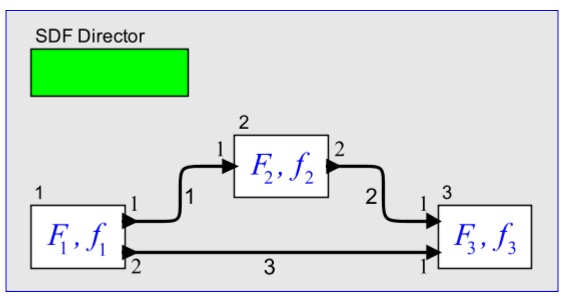


A solution to the balance equations:

$$q = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \qquad \Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \qquad \Gamma q = \vec{0}$$

This tells us that actor 3 must fire twice as often as actors 1 and 2.

Example



But there are many solutions to the balance equations:

$$q = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \quad q = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad q = \begin{bmatrix} 2 \\ 2 \\ 4 \end{bmatrix} \quad q = \begin{bmatrix} -1 \\ -1 \\ -2 \end{bmatrix} \quad q = \begin{bmatrix} \pi \\ \pi \\ 2\pi \end{bmatrix} \qquad \Gamma q = \vec{0}$$

For "well-behaved" models, there is a unique least positive integer solution.

Least Positive Integer Solution to the Balance Equations

Note that if p_C, c_C , the number of tokens produced and consumed on a connection C, are non-negative integers, then the balance equation,

$$q_A p_C = q_B c_C$$

implies:

- q_A is rational if and only if q_B is rational.
- q_A is positive if and only if q_B is positive.

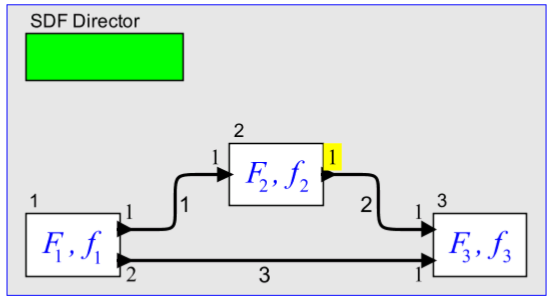
Consequence: Within any connected component, if there is any non-zero solution to the balance equations, then there is a unique least positive integer solution.

Consistent Models

An SDF model is *consistent* if there exists a non-zero solution to the balance equations.

Example of an Inconsistent Model: No Non-Trivial Solution to the Balance Equations

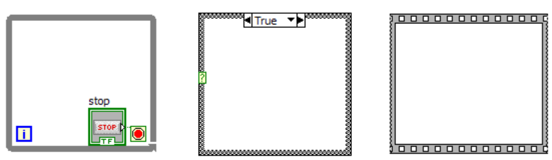
$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix}$$



There are no nontrivial solutions to the balance equations.

Note that this model can execute forever, but it requires unbounded memory.

Structured Dataflow



LabVIEW uses homogeneous SDF augmented with syntactically constrained forms of feedback and rate changes:

- While loops
- Conditionals
- Sequences

LabVIEW models are decidable.

Many other concurrent MoCs have been explored

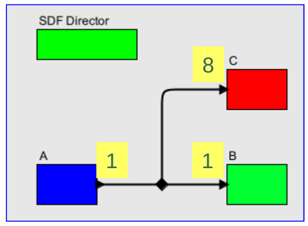
- (Kahn) process networks
- Communicating sequential processes (rendezvous)
- Time-driven models
- More dataflow variants:
  - cyclostatic
  - Heterochronous
  - …
- Petri nets
- …

Introduction to Embedded Systems

Material for Further Reading

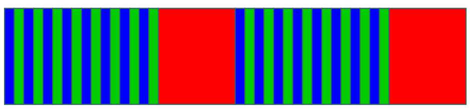
Abstracted Version of the Spectrum Example: Non-preemptive scheduling

Is this dataflow model dynamic?
Is it homogeneous?



Assume infinitely repeated invocations, triggered by availability of data at A.

Suppose that C requires 8 data values from A to execute. Suppose further that C takes much longer to execute than A or B. Then a schedule might look like this:

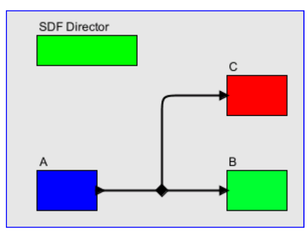


…

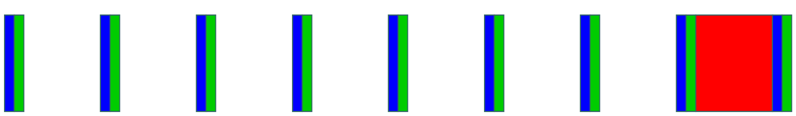
This suffers from jitter.

schedule Note that each period is EDF.

Uniformly Timed Schedule



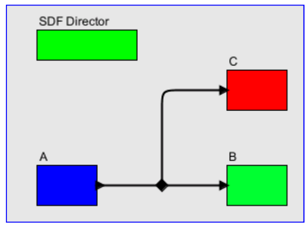
A preferable schedule would space invocations of A and B uniformly in time, as in:



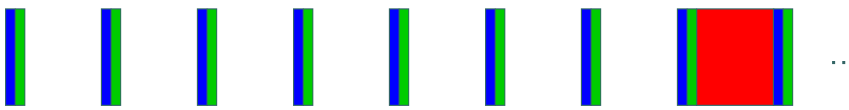
…

minimum latency

Non-Concurrent Uniformly Timed Schedule

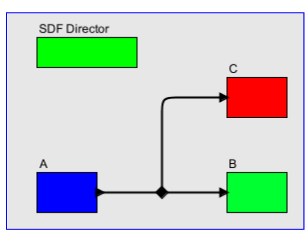


Notice that in this schedule, the rate at which A and B can be invoked is limited by the execution time of C.

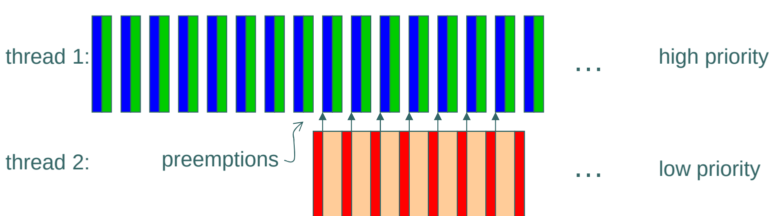


No jitter, but utilization is poor.

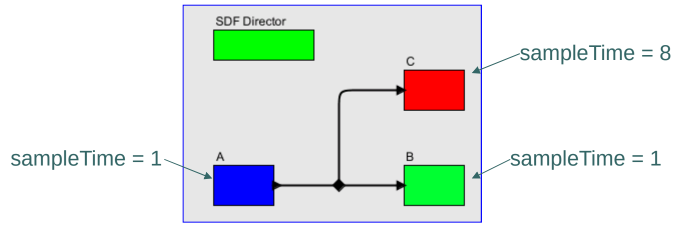
Concurrent Uniformly Timed Schedule: Preemptive schedule



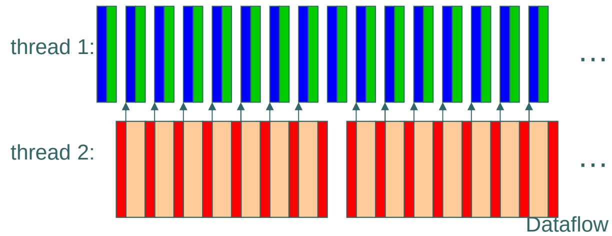
With preemption, the rate at which A and B can be invoked is limited only by total computation:



Ignoring Initial Transients, Abstract to Periodic Tasks

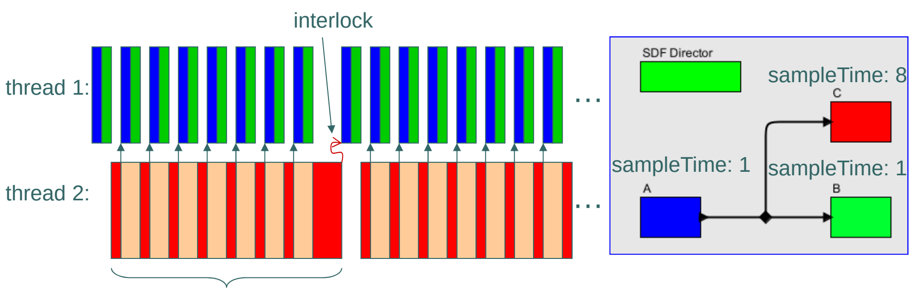


In steady-state, the execution follows a simple periodic pattern:



This follows the principles of rate-monotonic scheduling (RMS).

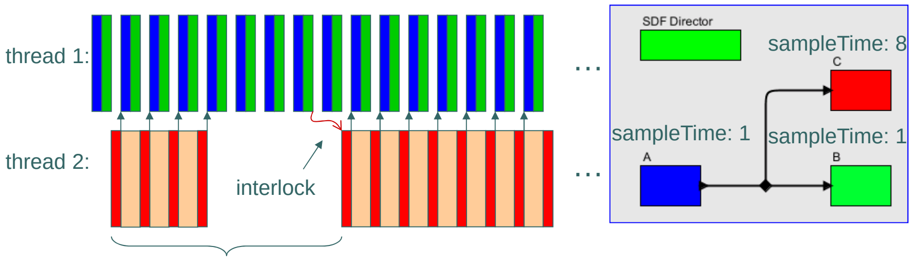
Requirement 1: Bounded Buffers



If the execution of C runs longer than expected, to keep the buffer bounded, thread 1 must be delayed accordingly. This can be accomplished with semaphore synchronization. But there are alternatives:

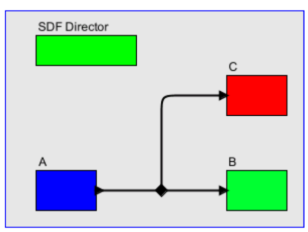
- Throw an exception to indicate timing failure.
- "Anytime" computation: use incomplete results of C

Requirement 2: Respect Data Dependence

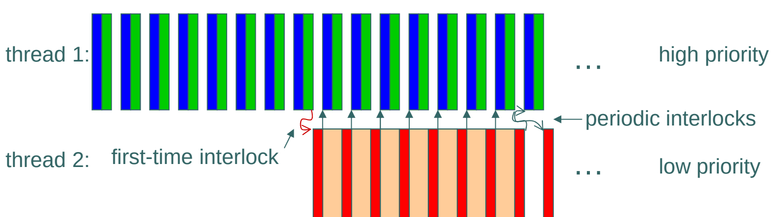


If the execution of C runs shorter than expected, data determinacy requires that thread 2 be delayed accordingly. That is, it must not start the next execution of C before the data is available.

In Dataflow, Interlocks and Built-in Buffering take care of these dependencies



For dataflow, a one-time interlock ensures sufficient data at the input of C for the first invocation:



Dataflow: many variants

- Computation graphs [Karp & Miller - 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow [Kodosky, 1986] now
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980's]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- Heterochronous dataflow [Girault, Lee, & Lee, 1997]
- Parameterized dataflow [Bhattacharya and Bhattacharyya 2001]
- Structured dataflow (again) [Thies et al. 2002]
- …

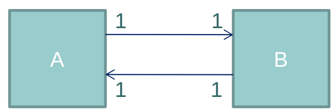
Lee 09: 6

Necessary and sufficient conditions

Consistency is a necessary condition to have a (bounded-memory) infinite execution.

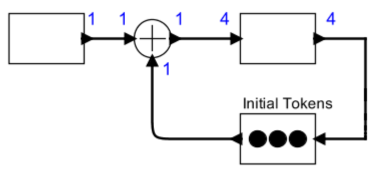
Is it sufficient?

Deadlock 1



Is this diagram consistent?

Deadlock 2



Some dataflow models cannot execute forever. In the above model, the feedback loop injects initial tokens, but not enough for the model to execute.

SDF: from static analysis to scheduling

Given: SDF diagram
Find: a bounded-buffer schedule, if it exists

Step 0: check whether diagram is consistent. If not, then no bounded-buffer schedule exists.

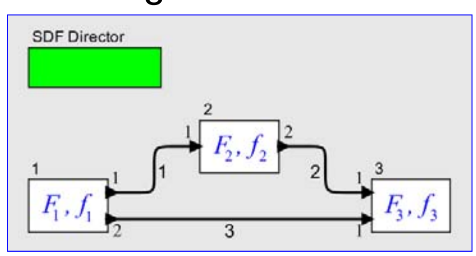
Step 1: find an integer solution to $\Gamma \boldsymbol{q}=\boldsymbol{0}$.

Step 2: "decompose" the solution q into a schedule, making sure buffers never become negative.

Step 2: "decomposing" the firing vector

Example 1:

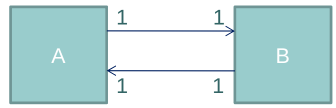
Schedule = (1;2;3;3)



$$b=\begin{bmatrix}0\\0\\0\end{bmatrix},\ q=\begin{bmatrix}1\\1\\2\end{bmatrix} \xrightarrow{\text{Fire 1}} b=\begin{bmatrix}1\\0\\2\end{bmatrix},\ q=\begin{bmatrix}0\\1\\2\end{bmatrix} \xrightarrow{\text{Fire 2}} b=\begin{bmatrix}0\\2\\2\end{bmatrix},\ q=\begin{bmatrix}0\\0\\2\end{bmatrix} \xrightarrow{\text{Fire 3}} b=\begin{bmatrix}0\\1\\1\end{bmatrix},\ q=\begin{bmatrix}0\\0\\1\end{bmatrix} \xrightarrow{\text{Fire 3}} b=\begin{bmatrix}0\\0\\0\end{bmatrix},\ q=\begin{bmatrix}0\\0\\0\end{bmatrix}$$

Step 2: “decomposing” the firing vector

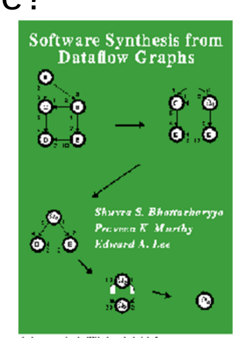
Example 2:



What happens if we try to run the previous procedure on this example?

So, we have both necessary and sufficient conditions for scheduling SDF graphs.

A Key Question: If More Than One Actor is Fireable in Step 2, How do I Select One?



Optimization criteria that might be applied:

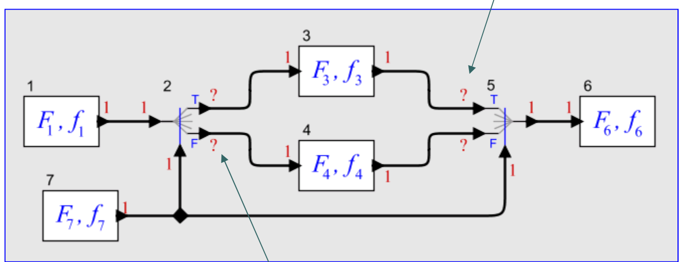
- Minimize buffer sizes.
- Minimize the number of actor activations.
- Minimize the size of the representation of the schedule (code size).

See S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs,* Kluwer Academic Press, 1996.

Beyond our scope here, but hints that it’s an interesting problem…

Dynamic Dataflow

What consumption rate?



What production rate?

Imperative equivalent:

```
while (true) {
   x = f1();
   b = f7();
   if (b) {
     y = f3(x);
   } else {
     y = f4(x);
   }
   f6(y);
}
```

The if-then-else model is not SDF. But we can clearly give a bounded *quasi-static* schedule for it:

(1, 7, 2, b?3, !b?4, 5, 6)

guard

Facts about (general) dynamic dataflow

- Whether there exists a schedule that does not deadlock is undecidable.
- Whether there exists a schedule that executes forever with bounded memory is undecidable.

Undecidable means that there is no algorithm that can answer the question in finite time for all finite models.