# Specification and Modeling (3)
# Dateflow

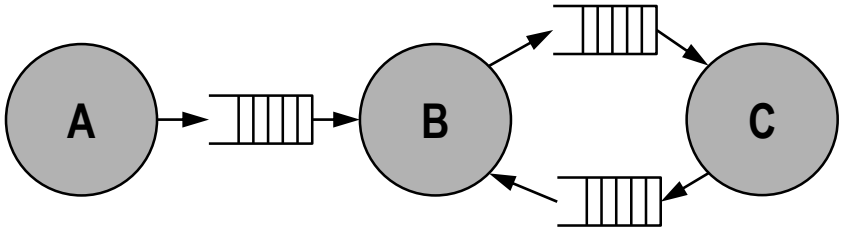
**Kai Huang**

# Outline

- Model of Computation (MoC)
- StateCharts
- Data-Flow Models
  - Kahn Process Network
  - Synchronous Dataflow

# Dataflow Language Model

- Processes communicating through FIFO buffers



# Philosophy of Dataflow Languages

- Drastically different way of looking at computation:
  - Imperative language style: program counter is king
  - Dataflow language: movement of data is the priority
  - Scheduling responsibility of the system, not the programmer

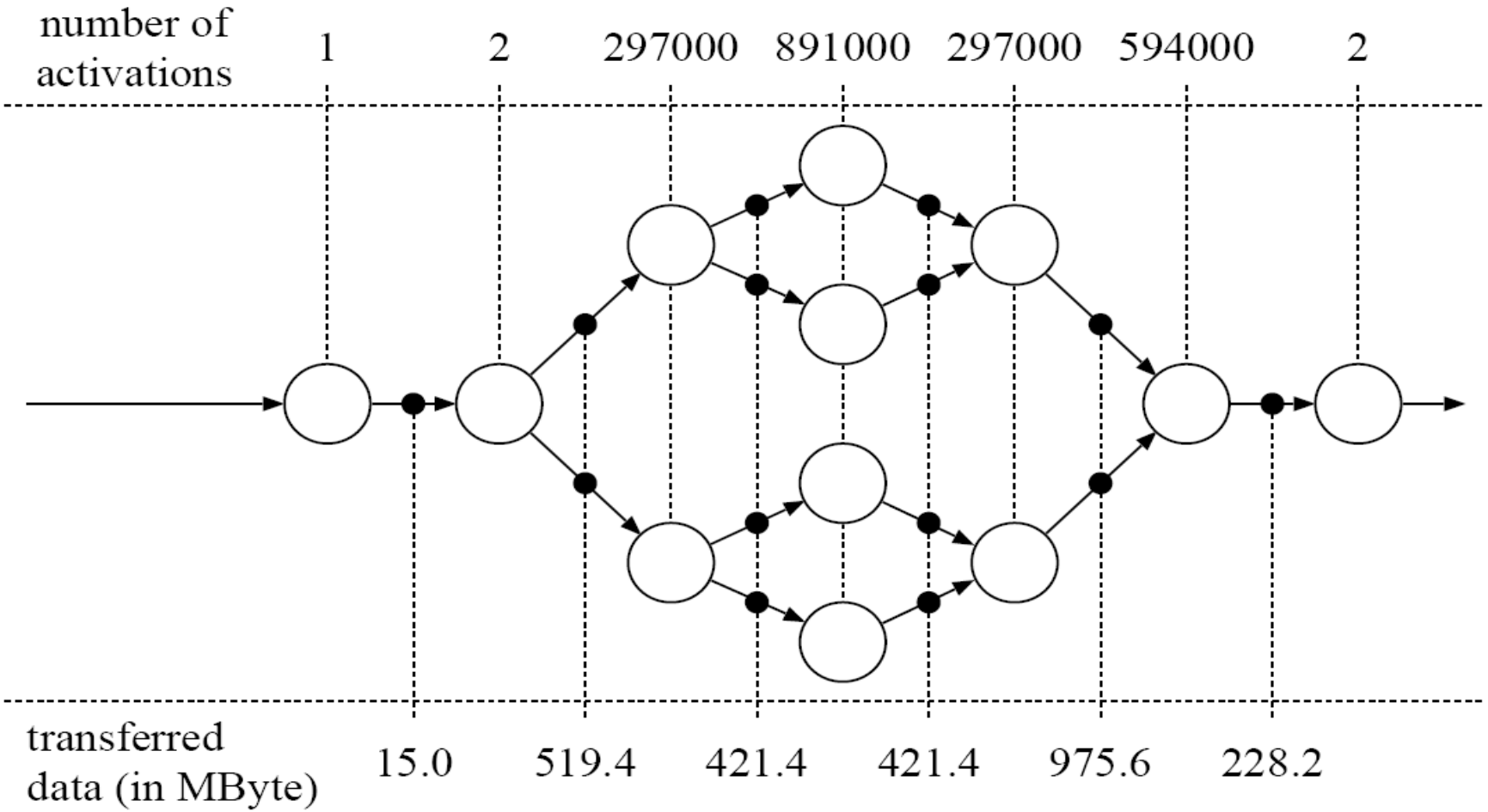
- Basic characteristic:
  - All processes run “simultaneously”
  - Processes can be described with imperative code
  - Processes can only communicate through buffers
  - Sequence of read tokens is identical to the sequence of written tokens

# Dataflow Languages

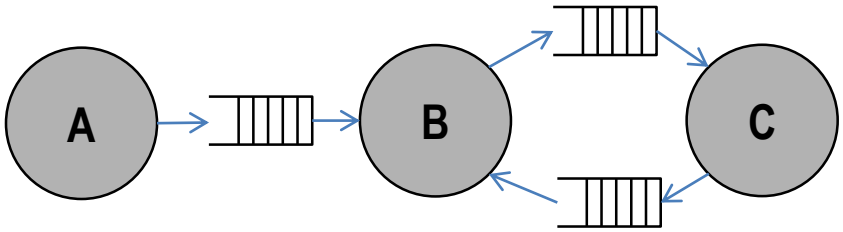
- Appropriate for applications that deal with streams of data:
  - Fundamentally concurrent: maps easily to parallel hardware
  - Perfect fit for block-diagram specifications (control systems, signal processing)
  - Matches well current and future trend towards multimedia applications

- Representation:
  - Host Language (process description), e.g. C, C++, Java, .... .
  - Coordination Language (network description), usually 'home made', e.g. XML.

# Example: MPEG-2 Video Decoder



# Kahn Process Networks

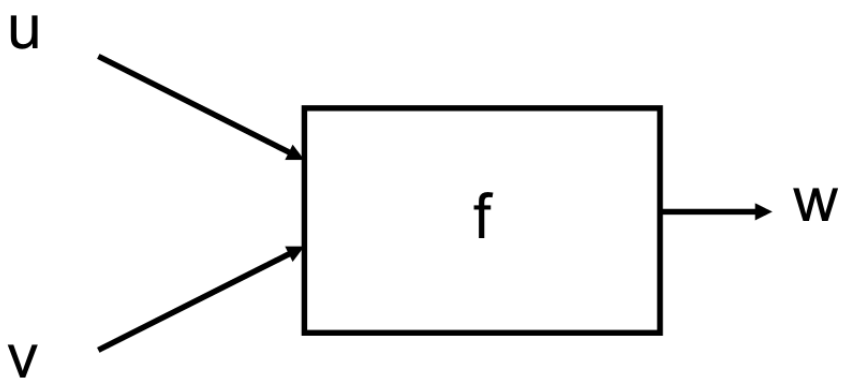



- Proposed by Kahn in 1974 as a general-purpose scheme for parallel programming:
  - **read**: destructive and blocking (reading an empty channel blocks until data is available)
  - **write**: non-blocking
  - **FIFO**: infinite size
- Unique attribute: determinate

# A Kahn Process (1)

- From Kahn's original 1974 paper

```
process f(in int u, in int v, out int w)
{
    int i; bool b = true;
    for (;;) {
        i = b ? wait(u) : wait(v);
        printf("%i\n", i);
        send(i, w);
        b = !b;
    }
}
```



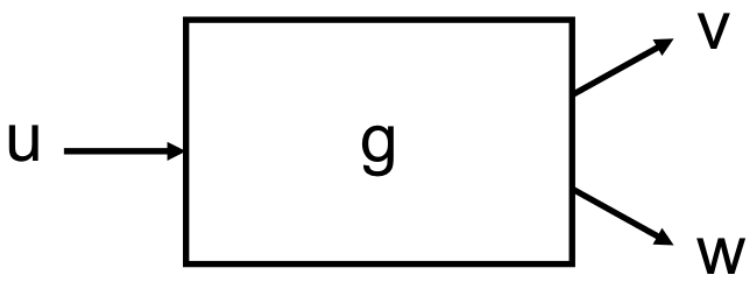

What does this do?

- Process alternately reads from u and v, prints the data value, and writes it to w

# A Kahn Process (2)

- From Kahn's original 1974 paper

```
process g(in int u, out int v, out int w)
{
    int i; bool b = true;
    for(;;) {
        i = wait(u);
        if (b)
            send(i, v);
        else
            send(i, w);
        b = !b;
    }
}
```




What does this do?

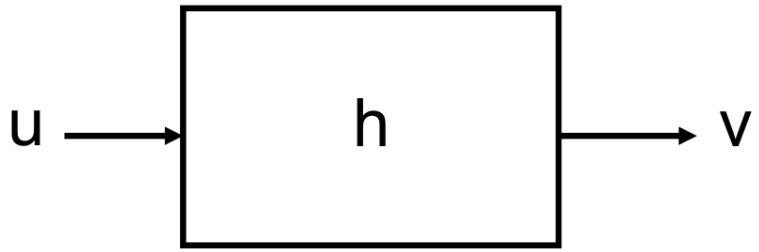
- Process reads from u and alternately copies it to v and w

# A Kahn Process (3)

- From Kahn's original 1974 paper

```
process h(in int u, out int v, int init)
{
    int i = init;
    send(i, v);

    for(;;) {
        i = wait(u);
        send(i, v);
    }
}
```

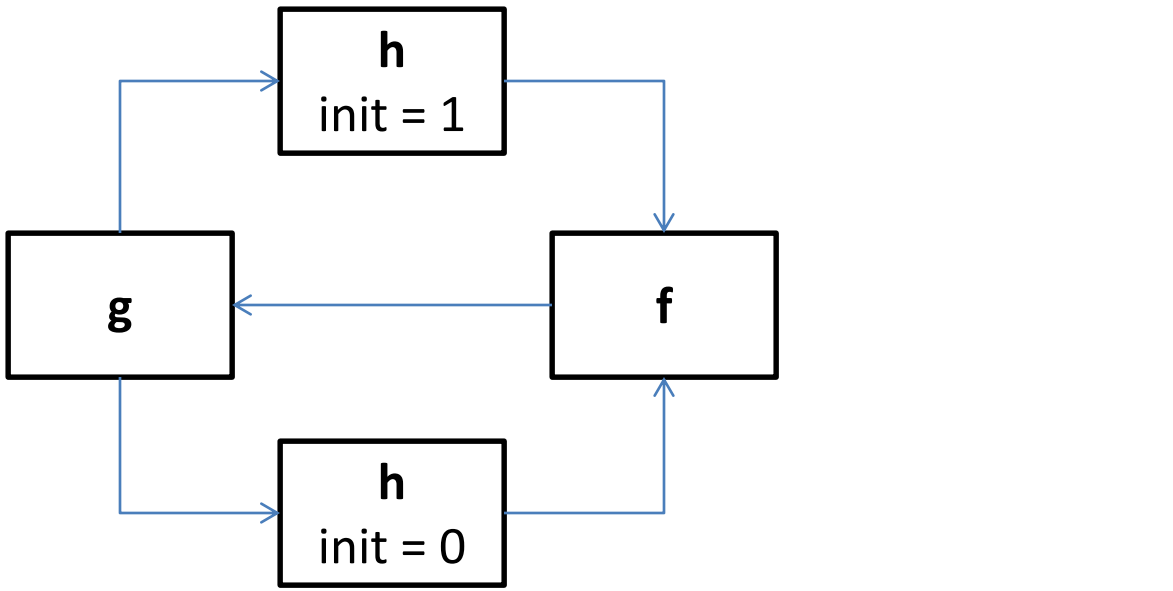



What does this do?

- Process sends initial value, then passes through values.

# A Kahn Process Network

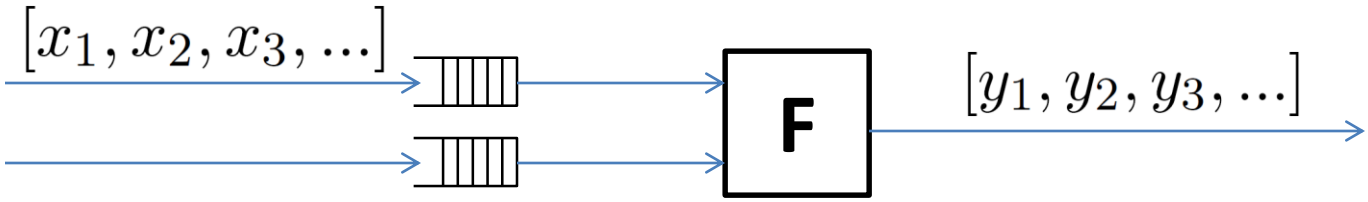
- What does this do?
  - Prints an alternating sequence of 0's and 1's.



# Determinacy (1)

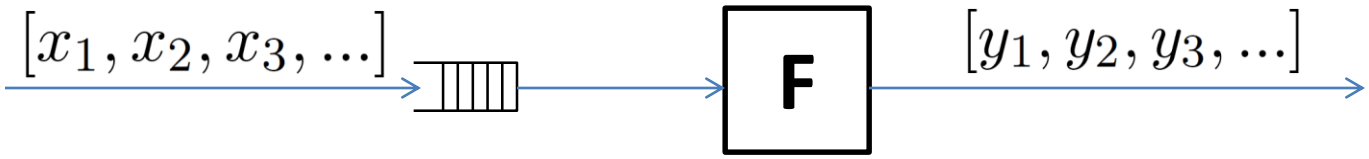
- Random:
  - A system is random if the information known about the system and its inputs is not sufficient to determine its outputs.
- Determinate:
  - Define the history of a channel to be the sequence of tokens that have been both written and read. A process network is said to be determinate if the histories of all channels depend only on the histories of the input channels.
- Importance:
  - Functional behavior is independent of timing (scheduling, communication time, execution time of processes).
  - Separation of functional properties and timing.

# Determinacy (2)




- Kahn Process: "monotonic mapping" of input sequence to output sequences:
  - Process uses prefix of input sequences to produce prefix of output sequences. In other words, if we consider two different scenarios, where the second one has just additional input tokens. Then the output sequence of the second scenario equals that of the first one, but with possible additional tokens appended.
- Process will not wait forever before producing an output (i.e., it will not wait for completion of an infinite input sequence).

# Determinacy (3)



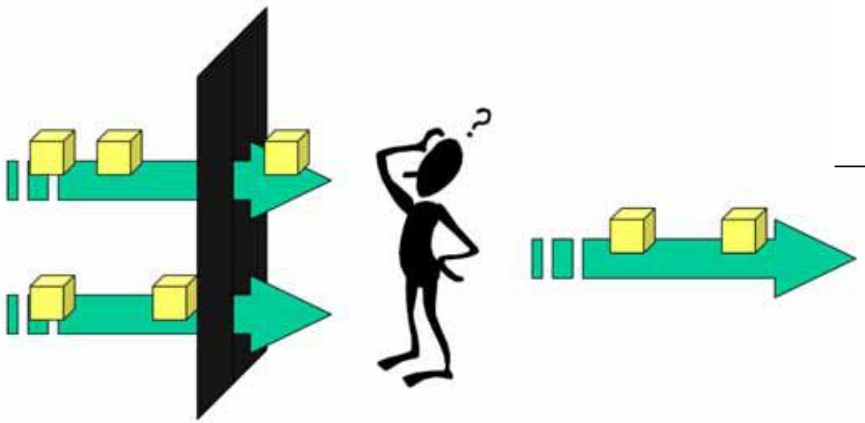

- Formal definition:
  - sequence (stream): $X = [x_1, x_2, x_3, ...]$
  - prefix ordering: $[x_1] \subseteq [x_1, x_2] \subseteq [x_1, x_2, x_3, ...]$
  - p-tuple of sequences: $\mathbf{X} = (X_0, X_1, ..., X_p) \in S^p$
  - ordered set of sequences: $\mathbf{X} \subseteq \mathbf{X}' \ if \ (\forall X_i \subseteq X_i')$
  - process: $F = S^p \rightarrow S^q$
  - monotonic process: $\mathbf{X} \subseteq \mathbf{X}' \Longrightarrow F(\mathbf{X}) \subseteq F(\mathbf{X}')$

# Proof of Determinism

- Why is a Kahn Process Network (consisting of monotonic processes) determinate?
  - A network of monotonic processes itself defines a monotonic process.
  - A monotonic process is clearly determinate.
- Reasoning:
  - If I'm a process, I am only affected by the sequence of tokens on my inputs
  - I can't tell whether they arrive early, late, or in what order
  - I will behave the same in any case
  - Thus, the sequence of tokens I put on my outputs is the same regardless of the timing of the tokens on my inputs

# Adding Non-Determinacy

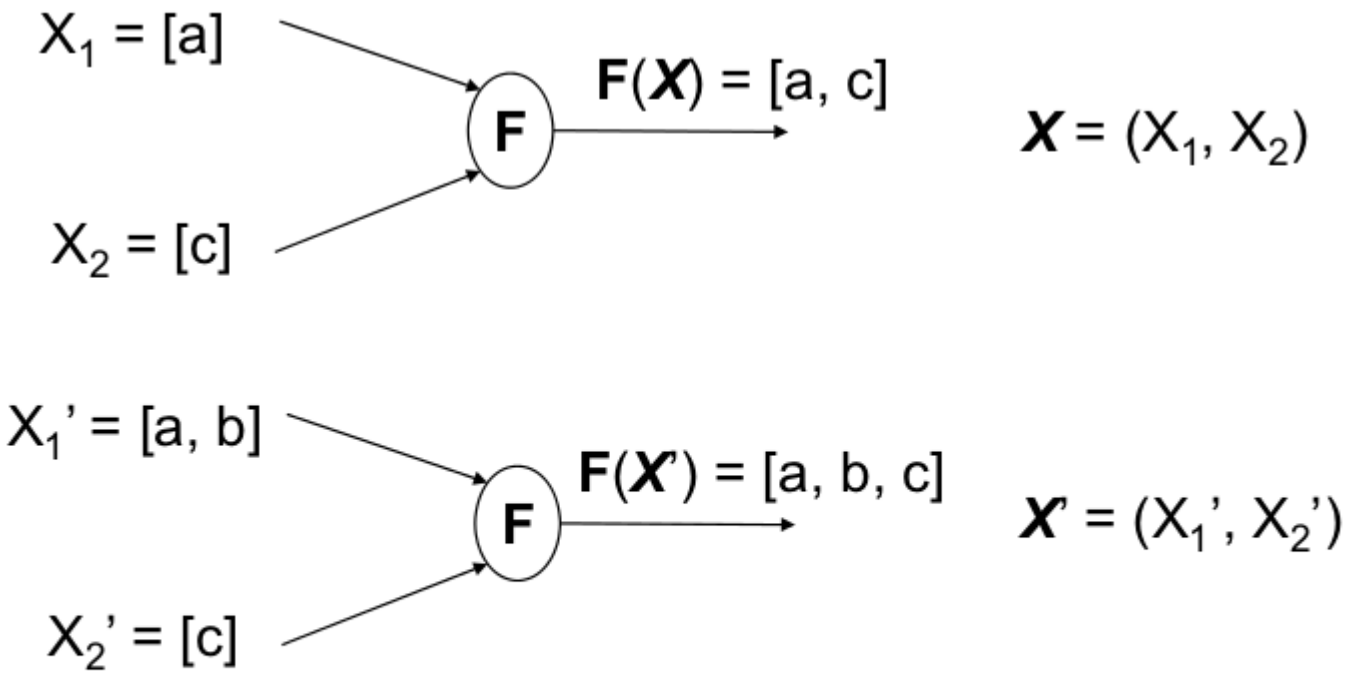
- There are several ways to introduce non-monotonic behavior:
  - Allow processes to test for emptiness
  - Allow more than one process to read from or to write to a channel
  - Allow processes to share a variable
- Example of fair merge:
  - don't let any channel starve



**Algorithm 1**

```
if test(L1) then
    int X = read(L1), write(X, out)
end if
if test(L2) then
    int Y = read(L2), write(Y, out)
end if
```

# Adding Non-Determinacy

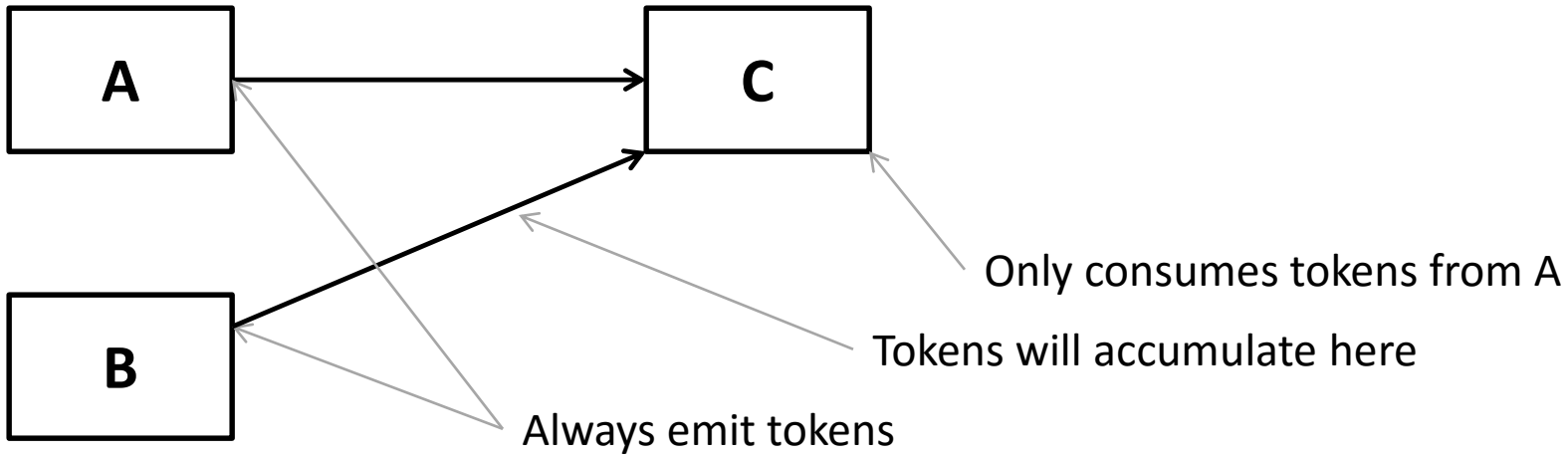



$X \subseteq X'$ as $([a], [c]) \subseteq ([a, b], [c])$

$F(X) \not\subseteq F(X')$ as $[a, c] \not\subseteq [a, b, c]$

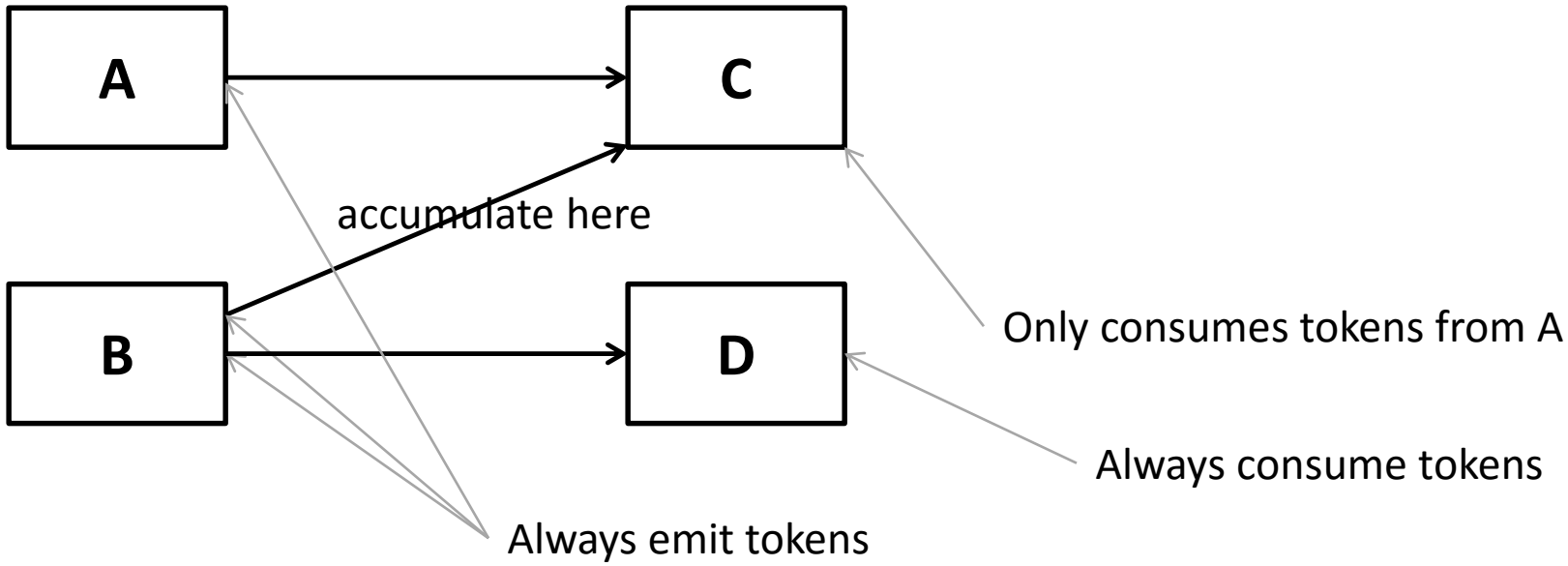
# Scheduling Kahn Networks

- Challenge is running processes without accumulating tokens and without producing a deadlock.



# Demand-driven Scheduling?

- Only run a process whose outputs are being actively solicited.
- However...

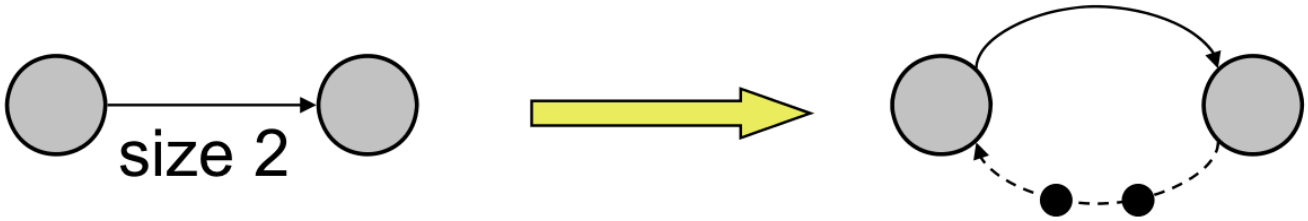


# Tom Parks' Algorithm

- Schedules a Kahn Process Network in bounded memory
- if it is possible:
  - Start with a network with bounded buffer sizes and blocking write (can be transformed into a conventional KPN).
  - Use any scheduling technique to schedule the execution of processes that does not stall if there is still a process that is not blocked.
  - As long as the process networks runs without deadlock caused by blocking write -> continue.
  - If system deadlocks because of blocking write, increase size of smallest buffer and continue.

# From Infinite to Finite Buffer Size

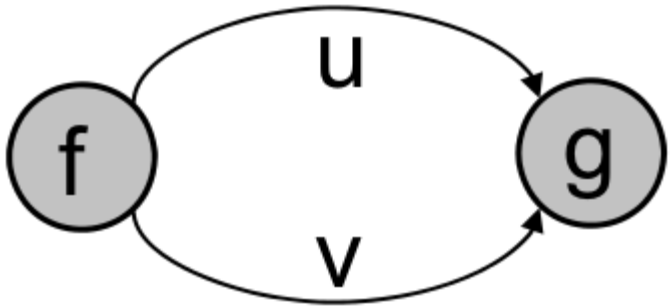
- A given KPN can be changed such that buffer sizes are finite:
  - To every channel between two processes an additional (virtual) channel in the reverse direction is added.
  - If the buffer size is restricted to n, then this new virtual channel has n initial data.
  - Every write (read) of the original channel leads to a (read) write of the virtual channel.
  - Invariant: The sum of token in both channels is constant.




  - The resulting KPN has the same functional behavior but it may deadlock because of the finite buffer sizes

# Deadlock Example

- An example of a KPN that produces a deadlock caused by a finite buffer size (size(u) < 2):

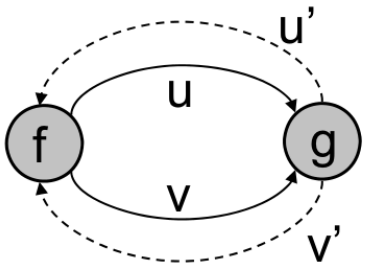



```
process f(out int a, out int b) {
    for (;;) {
        send(1, u);
        send(1, u);
        send(1, v);
    }
}
```

```
process g(in int a, in int b) {
    for (;;) {
        wait(v);
        wait(u);
        wait(u);
    }
}
```

# Example: Finite Size Buffers in KPN

- The previous process network with finite buffer size can be converted into a KPN:




size(u) = 2
size(v) = 1

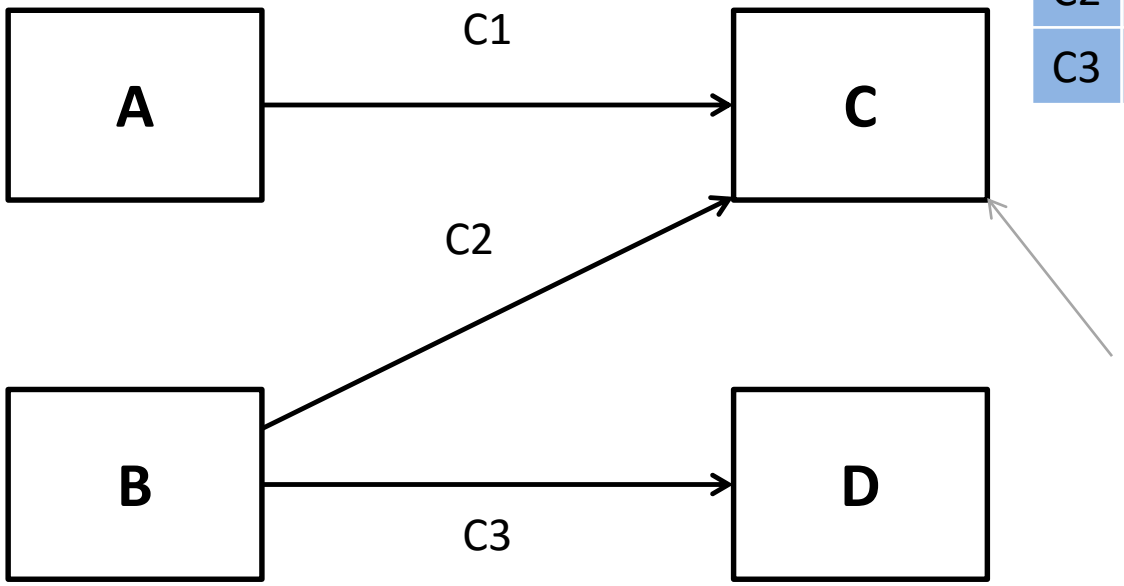
```
process f(...) {
    for (;;) {
        wait(u'); send(1, u);
        wait(u'); send(1, u);
        wait(v'); send(1, v);
    }
}
```

```
process g(...) {
    send(1,u'); send(1, u'); send(1, v');
    for (;;) {
        wait(v); send(1, v');
        wait(u); send(1, u');
        wait(u); send(1, u'); }
}
```

# Parks' Algorithm in Action

- Start with buffers of size 1.
- Run A, B, C, D.

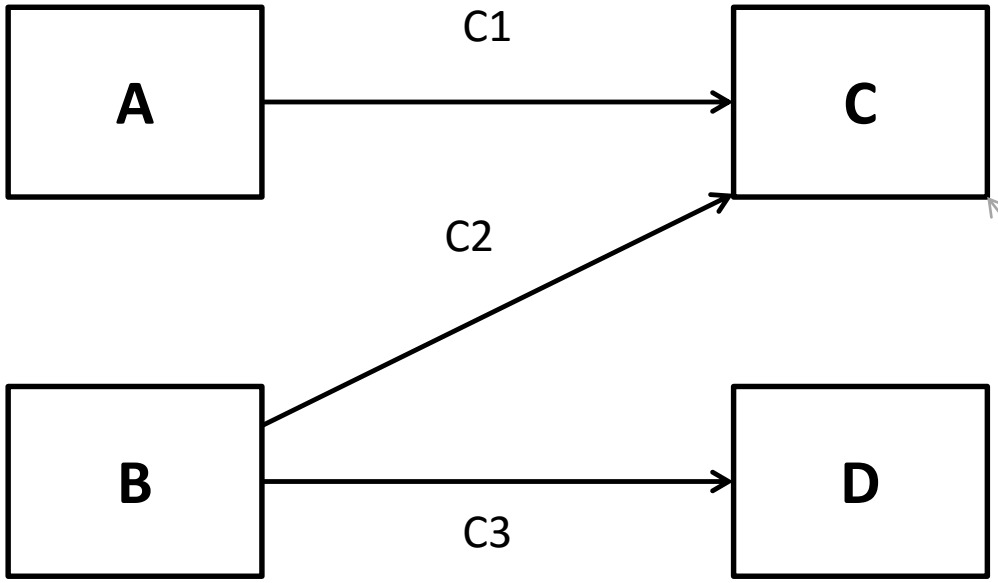
| | A | B | C | D |
|---|---|---|---|---|
| C1 | 1 | 1 | 0 | 0 |
| C2 | 0 | 1 | 1 | 1 |
| C3 | 0 | 1 | 1 | 0 |



# Parks' Algorithm in Action

- B blocked waiting for space in B->C buffer.
- System will run indefinitely.

| | A | B | C | D | A | C | A |
|---|---|---|---|---|---|---|---|
| C1 | 1 | 1 | 0 | 0 | 1 | 0 | … |
| C2 | 0 | 1 | 1 | 1 | 1 | 1 | … |
| C3 | 0 | 1 | 1 | 0 | 0 | 0 | … |




Only consumes tokens from A

# Parks' Algorithm Summary

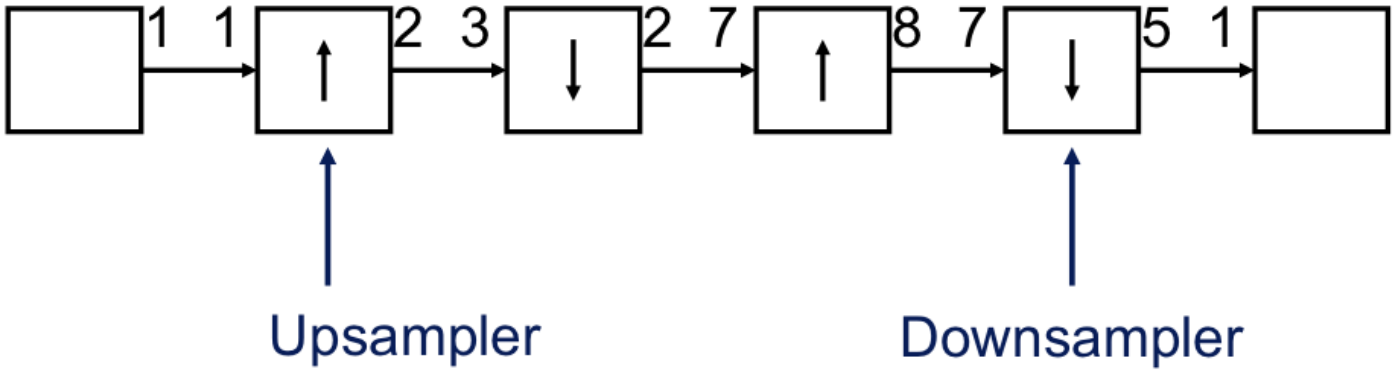
- Parks' algorithm gives priority to non-terminating execution over bounded execution, and bounded execution is given priority over complete execution

# Evaluation of Kahn Process Networks

- Pro:
  - Their beauty is that the scheduling algorithm does not affect their functional behavior
  - Matches stream-based processing
  - Makes coarse grained parallelism explicit
  - Favors mapping to multi-processor and distributed platforms
- Con:
  - Difficult to schedule because of need to balance relative process rates
  - System inherently gives the scheduler few hints about appropriate rates
  - Parks' algorithm expensive and fussy to implement

# Synchronous Dataflow (SDF)

- Edward Lee and David Messerschmitt, Berkeley, 1987:
  - Restriction of Kahn Networks to allow compile-time scheduling.
  - Each process reads and writes a fixed number of tokens each time it fires; firing is an atomic process.
- Example: DAT-to-CD rate converter (converts a 44.1 kHz sampling rate to 48 kHz)




( 44.1*2/1*2/3*8/7*5/7 = 48 )

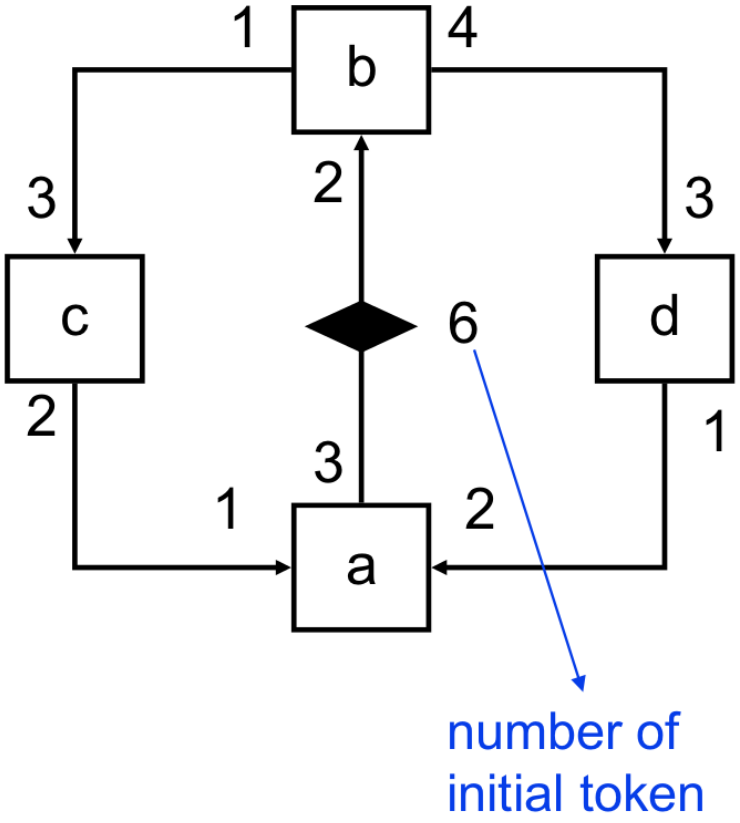
# SDF Scheduling

- Schedule can be determined completely at compile time (before the system runs).
- Two steps:
  1. Establish relative execution rates by solving a system of linear equations (balancing equations).
  2. Determine periodic schedule by simulating system for a single round (returns the number of tokens in each buffer to their initial state).
- Result: the schedule can be executed repeatedly without accumulating tokens in buffers

# Balancing Equations

Each arc imposes a constraint:

$$3a - 2b = 0$$
$$4b - 3d = 0$$
$$b - 3c = 0$$
$$2c - a = 0$$
$$d - 2a = 0$$




$$\begin{bmatrix} 3 & -2 & 0 & 0 \\ 0 & 4 & 0 & -3 \\ 0 & 1 & -3 & 0 \\ -1 & 0 & 2 & 0 \\ -2 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

M

q

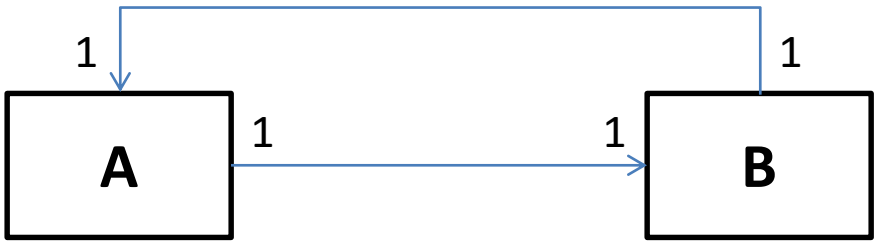
# Solving the Balancing Equation

- Main SDF scheduling theorem (Lee ‘87):
  - A connected SDF graph with n actors has a periodic schedule iff its topology matrix M has rank n-1
  - If M has rank n-1 then there exists a unique smallest positive integer solution q to M q = 0
- Inconsistent systems only have the all-zeros solution
- Disconnected systems have two- or higher-dimensional solutions
- Example:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & -2 & 0 & 0 \\ 0 & 4 & 0 & -3 \\ 0 & 1 & -3 & 0 \\ -1 & 0 & 2 & 0 \\ -2 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

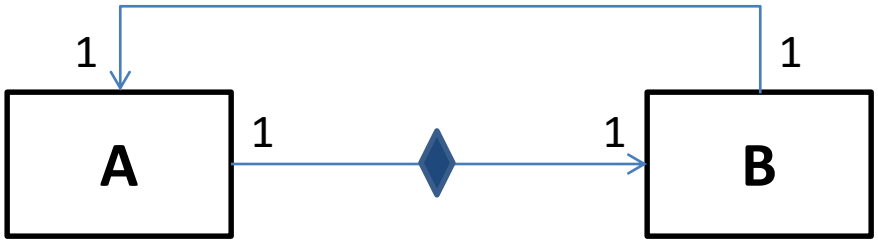
# Consistent Rates Not Enough

- A consistent system with no schedule
- Rates do not avoid deadlock
- Example: deadlock in consistent system




Initially No Tokens
a waits for b
b waits for a
Deadlock!!!

- Solution here: add an initial token (delay) on one of the arcs

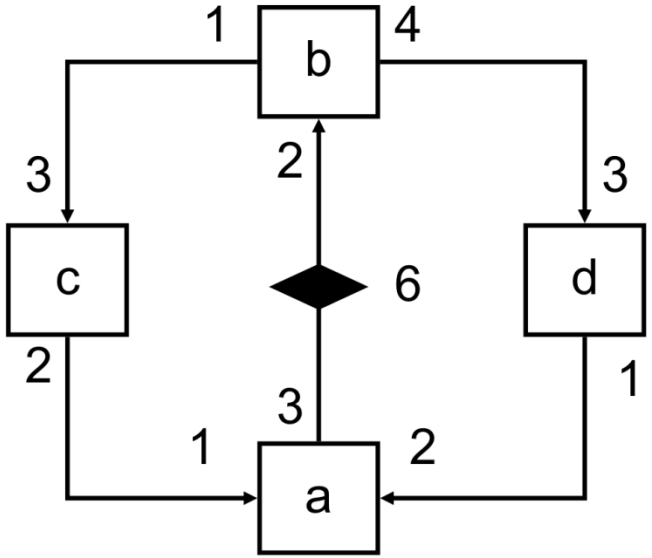



Initially 1 Token on
arc ab
b can fire

# Determine Periodic Schedule

- Possible schedules:
  - (BBBCDDDDAA)*
  - (BDBDBCADDA)*
  - (BBDDBDDCAA)*
  - ...

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 4 \end{bmatrix}$$




- Systems often have many possible schedules. How can we use this flexibility?
  - Reduced code size (loop structure, hierarchy)
  - Reduced buffer sizes

# Code Generation

- Simple approach:

  the schedule:

  BBBCDDDDAA
- would produce code without loops like

```
B;
B;
B;
C;
D;
D;
D;
D;
A;
A;
```

- Obvious improvement: use loops
- Rewrite the schedule in “looped” form:

  (3 B) C (4 D) (2 A)
- Generated code becomes

```
for (i=0; i<3; i++) B;
C;
for (i=0; i<4; i++) D;
for (i=0; i<2; i++) A;
```

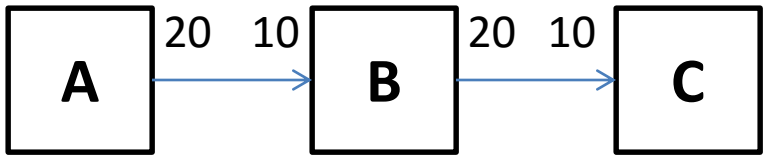
# Code Size optimization

- Find Single Appearance Schedule:
  - (3 B) C (4 D) (2 A)
- Often possible to choose a looped schedule in which each block appears exactly once
- Leads to efficient block-structured code
  - Only requires one copy of each block's code
- Does not always exist
- Often requires more buffer space than other schedules
- Generated program with efficient code size

```
for (i=0; i<3; i++) B;
C;
for (i=0; i<4; i++) D;
for (i=0; i<2; i++) A;
```

# Buffer Size optimization

- Find Minimum Memory Schedules
- Often increases code size (block-generated code)
- Static scheduling makes it possible to exactly predict memory requirements




| Schedule | Total buffer sizes |
|---|---|
| (1) ABCBCCC | 50 tokens |
| (2) A(2B)(4 C) | 60 tokens |
| (3) A(2(B (2C))) | 40 tokens |
| (4) A(2(BC))(2 C) | 50 tokens |

- Simultaneously improving code size, memory requirements, sharing buffers, etc. remains open research problems

# Evaluation of SDF

- Pros:
  - Compile-time schedurability verificaiton
- Cons:
  - Process semantics is too restricted
  - Lack of means to model and analyze Non-funtional properties
    - Energy, WCET, reliability