Fundamental Algorithms for System Modeling, Analysis, and Optimization

Stavros Tripakis, Edward A. Lee
UC Berkeley
EECS 144/244
Fall 2014

Scheduling dataflow systems

Dataflow

Generic term in CS, multiple meanings.

Common theme: data flowing through some computing network.

These lectures: asynchronous processes communicating via FIFO queues.

Applications:

- Computer architecture: dataflow vs. von Neumann
- Signal processing (e.g., SDF)
- Distributed systems (e.g., Kahn process networks)
- …

A zoo of dataflow models

- 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]
- 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]
- Scenarios [Geilen, 2010]
- …

We will look at untimed, then timed SDF

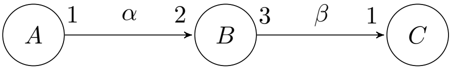
Synchronous Data Flow –

a better (and sometimes used) term is **Static Data Flow** (SDF)

- One of the most basic dataflow models
- Proposed in 1987 [Lee and Messerschmitt, 1987]
- Widely used: mainly in signal-processing applications
- Many many variants: SDF, CSDF, HSDF, SADF, ...
- Semantics: untimed, timed, probabilistic
  - untimed variants can be used for checking correctness of the system (e.g., consistency, deadlocks), and for design-space exploration (e.g., buffer sizing)
  - timed variants can be used for performance analysis:
    - worst-case
    - or, in the case of probabilistic models, average-case
- We look first at untimed, then at timed SDF

UNTIMED SDF

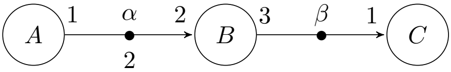
(Untimed) SDF: Synchronous (Static) Data Flow



An SDFG (SDF Graph)

- A, B, C: dataflow *actors*
- $1, 2, 3, \ldots$: *token* production/consumption *rates*
- α, β: (a-priori unbounded) FIFO queues (*channels*)
  - each channel has unique producer/consumer
  - abstract from token values $\Rightarrow$ FIFO property ignored

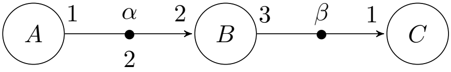
(Untimed) SDF: Synchronous (Static) Data Flow



An SDFG (SDF Graph)

- A, B, C: dataflow *actors*
- $1, 2, 3, \ldots$: *token* production/consumption *rates*
- α, β: (a-priori unbounded) FIFO queues (*channels*)
  - each channel has unique producer/consumer
  - abstract from token values $\Rightarrow$ FIFO property ignored
- Channels can have *initial tokens*

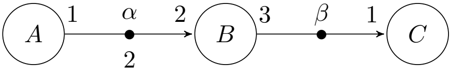
(Untimed) SDF



An SDFG (SDF Graph)

- Behavior (intuition):
  - Asynchronous interleaving: any actor that has enough input tokens available can *fire*.
  - A: can fire at any time; it produces 1 token every time it fires
  - B: needs 2 tokens in order to fire; it consumes 2 tokens and produces 3 tokens every time it fires;
  - C: needs 1 token in order to fire; it consumes 1 token each time it fires.

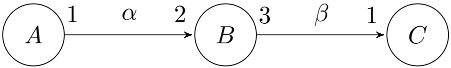
(Untimed) SDF



An SDFG (SDF Graph)

- Behavior (more formally):
  - An SDFG semantics = labeled transition system (LTS): states + labeled transitions.
  - **state**: # tokens in every channel
  - **transition** labeled A: actor A fires.

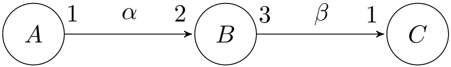
Example



This SDFG defines the following LTS:

$$(0, 0)$$

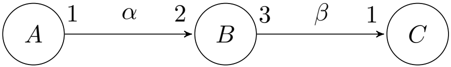
Example



This SDFG defines the following LTS:

$$(0,0) \xrightarrow{A} (1,0)$$

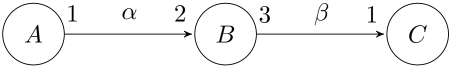
Example



This SDFG defines the following LTS:

$$(0,0) \xrightarrow{A} (1,0) \xrightarrow{A} (2,0)$$

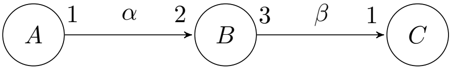
Example



This SDFG defines the following LTS:

$$(0,0) \xrightarrow{A} (1,0) \xrightarrow{A} (2,0) \xrightarrow{A} (3,0) \xrightarrow{A} (4,0) \cdots$$

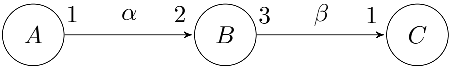
Example



This SDFG defines the following LTS:

$$\begin{tikzcd} (0,0) \arrow[r, "A"] & (1,0) \arrow[r, "A"] & (2,0) \arrow[r, "A"] \arrow[dr, "B"'] & (3,0) \arrow[r, "A"] & (4,0) \cdots \\ & & & (0,3) & \end{tikzcd}$$

Example



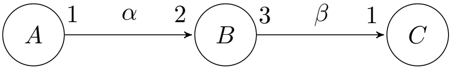
This SDFG defines the following LTS:

$$(0,0) \xrightarrow{A} (1,0) \xrightarrow{A} (2,0) \xrightarrow{A} (3,0) \xrightarrow{A} (4,0) \cdots$$

$$(2,0) \xrightarrow{B} (0,3) \xrightarrow{A} (1,3) \xrightarrow{A} (2,3) \cdots$$

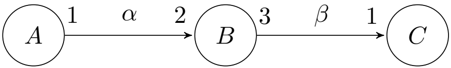
$$(0,3) \xrightarrow{C} (0,2) \cdots$$

Observations



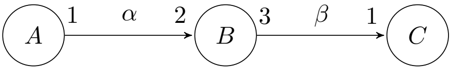
- There exist behaviors where queues grow unbounded

Observations



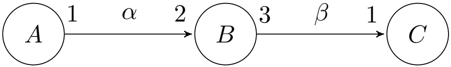
- There exist behaviors where queues grow unbounded
- But there are also behaviors where this doesn't happen
  - and all actors keep firing

Observations



- There exist behaviors where queues grow unbounded
- But there are also behaviors where this doesn't happen
  - and all actors keep firing
- These are the behaviors we want.

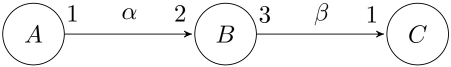
Observations



- There exist behaviors where queues grow unbounded
- But there are also behaviors where this doesn't happen
  - and all actors keep firing
- These are the behaviors we want.
- Represented by *periodic schedules*:

$$(AABCCC)^{\omega}$$

Observations

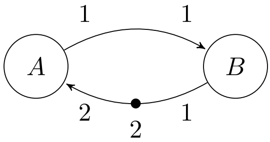


- There exist behaviors where queues grow unbounded
- But there are also behaviors where this doesn't happen
  - and all actors keep firing
- These are the behaviors we want.
- Represented by *periodic schedules*:

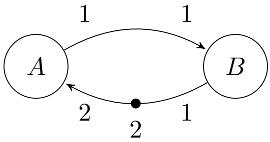
$$(AABCCC)^{\omega}$$

- Can we always find such schedules?

Deadlock



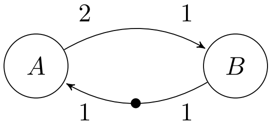
Deadlock



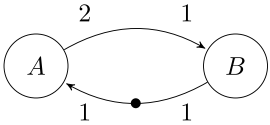
Behavior deadlocks!

$$(0,2) \xrightarrow{A} (1,0) \xrightarrow{B} (0,1)$$

Unbounded behavior



Unbounded behavior

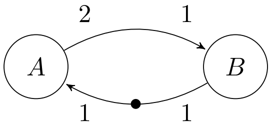


Queues keep growing!

$$(0,1) \xrightarrow{A} (2,0) \xrightarrow{B} (1,1) \xrightarrow{B} (0,2) \xrightarrow{A} (2,1) \xrightarrow{B} (1,2) \xrightarrow{B} (0,3) \cdots$$

ANALYZING UNTIMED SDF GRAPHS

Balance equations and repetition vectors



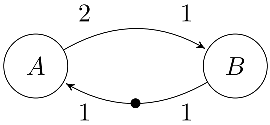
Balance equations:

- For each channel: tokens produced = tokens consumed
  - initial tokens don't matter for balance equations

$$
\begin{aligned}
q_A \cdot 2 &= q_B \cdot 1 && \text{// equation for channel } A \to B \\
q_B \cdot 1 &= q_A \cdot 1 && \text{// equation for channel } B \to A
\end{aligned}
$$

- q_A: number of times actor A fires
- Solution to balance equations is called **repetition vector**

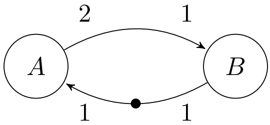
Balance equations and repetition vectors



$$\begin{aligned} 2q_A &= q_B \\ q_B &= q_A \end{aligned}$$

- Only trivial solution (always exists): $q_A = q_B = 0$
- SDF graph is *inconsistent*

Balance equations and repetition vectors

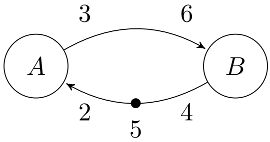


$$\begin{aligned} 2q_A &= q_B \\ q_B &= q_A \end{aligned}$$

- Only trivial solution (always exists): $q_A = q_B = 0$
- SDF graph is *inconsistent*
- Inconsistency $\Rightarrow$ cannot hope to find periodic schedule that keeps queues bounded and fires all actors infinitely often.

Balance equations and repetition vectors

Another example:

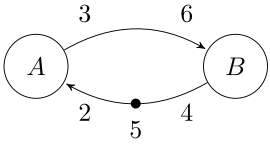


$$\begin{aligned} 3q_A &= 6q_B && \text{// equation for channel } A \to B \\ 2q_A &= 4q_B && \text{// equation for channel } B \to A \end{aligned}$$

- Non-zero solution: $q_B = 1$, $q_A = 2q_B = 2$
  - Any multiple is also a solution
- SDF graph is *consistent*

Balance equations and repetition vectors

Another example:

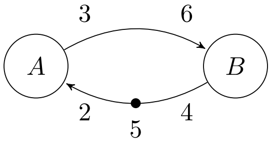


$$
\begin{aligned}
3q_A &= 6q_B && \text{// equation for channel } A \to B \\
2q_A &= 4q_B && \text{// equation for channel } B \to A
\end{aligned}
$$

- Non-zero solution: $q_B = 1$, $q_A = 2q_B = 2$
  - Any multiple is also a solution
- SDF graph is *consistent*
- In this case we also have a periodic schedule: $(AAB)^\omega$.

Balance equations and repetition vectors

Another example:

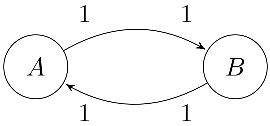


$$
\begin{aligned}
3q_A &= 6q_B && \text{// equation for channel } A \to B \\
2q_A &= 4q_B && \text{// equation for channel } B \to A
\end{aligned}
$$

- Non-zero solution: $q_B = 1$, $q_A = 2q_B = 2$
  - Any multiple is also a solution
- SDF graph is *consistent*
- In this case we also have a periodic schedule: $(AAB)^\omega$.
- Does consistency imply existence of valid periodic schedule?

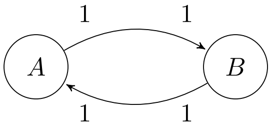
Consistency vs. deadlock

Consistency $\not\Rightarrow$ absence of deadlock:

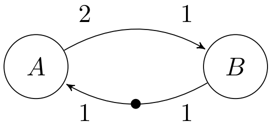


Consistency vs. deadlock

Consistency $\not\Rightarrow$ absence of deadlock:



Absence of deadlock $\not\Rightarrow$ consistency:



Consistency

- Can "chain SDFGs" be inconsistent?

Consistency

- Can "chain SDFGs" be inconsistent?
  No. Can obtain rate of down-stream actor relative to that of up-stream actor $\Rightarrow$ then use LCM (least common multiple) to normalize.

Consistency

- Can "chain SDFGs" be inconsistent?
  No. Can obtain rate of down-stream actor relative to that of up-stream actor $\Rightarrow$ then use LCM (least common multiple) to normalize.

- Can "tree SDFGs" be inconsistent?

Consistency

- Can "chain SDFGs" be inconsistent?
  No. Can obtain rate of down-stream actor relative to that of up-stream actor $\Rightarrow$ then use LCM (least common multiple) to normalize.

- Can "tree SDFGs" be inconsistent?
  No. Idem.

Consistency

- Can "chain SDFGs" be inconsistent?
  No. Can obtain rate of down-stream actor relative to that of up-stream actor $\Rightarrow$ then use LCM (least common multiple) to normalize.

- Can "tree SDFGs" be inconsistent?
  No. Idem.

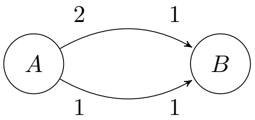
- Can arbitrary DAGs (directed **acyclic** graphs) be inconsistent?

Consistency

- Can "chain SDFGs" be inconsistent?
  No. Can obtain rate of down-stream actor relative to that of up-stream actor $\Rightarrow$ then use LCM (least common multiple) to normalize.

- Can "tree SDFGs" be inconsistent?
  No. Idem.

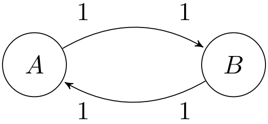
- Can arbitrary DAGs (directed **acyclic** graphs) be inconsistent?
  Yes:



CHECKING FOR DEADLOCKS

What about deadlock?

Consistency $\not\Rightarrow$ absence of deadlocks:

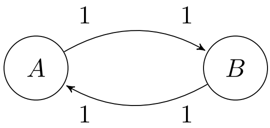


How to check whether a given SDF graph deadlocks?

Checking for deadlocks

1. Check consistency: if consistent, compute non-zero repetition vector $\mathbf{q}$
2. **Simulate** execution of SDFG, firing actors no more times than what $\mathbf{q}$ specifies ($\Rightarrow$ termination)
   - if we manage to complete execution then no deadlock: periodic schedule has been found
   - otherwise: SDFG deadlocks

Checking for deadlocks: example

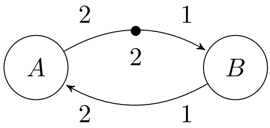


1. Graph consistent: $q_A = q_B = 1$
2. Simulate execution:

   from initial channel state $(0, 0)$ no firing possible

   $\Rightarrow$ SDFG deadlocks

Checking for deadlocks: another example



1. Graph consistent: $q_A = 1, q_B = 2$
2. Simulate execution (firing A at most once, B twice):

$$(2,0) \xrightarrow{B} (1,1) \xrightarrow{B} (0,2) \xrightarrow{A} (2,0)$$

$\Rightarrow$ SDFG is deadlock-free

$\Rightarrow$ schedule: $(BBA)^\omega$

Interesting questions on deadlock-checking algorithm

- Why is it enough to stop after one repetition vector?

Interesting questions on deadlock-checking algorithm

- Why is it enough to stop after one repetition vector?
  - channel state after one repetition = initial channel state

    $\Rightarrow$ schedule can be repeated forever

    $\Rightarrow$ found periodic schedule !

Interesting questions on deadlock-checking algorithm

- Why is it enough to stop after one repetition vector?
  - channel state after one repetition = initial channel state

    $\Rightarrow$ schedule can be repeated forever

    $\Rightarrow$ found periodic schedule !

- Does the order in which actors are fired during simulation matter?

Interesting questions on deadlock-checking algorithm

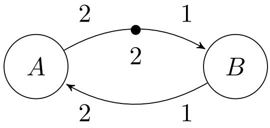
- Why is it enough to stop after one repetition vector?
  - channel state after one repetition = initial channel state
    $\Rightarrow$ schedule can be repeated forever
    $\Rightarrow$ found periodic schedule !
- Does the order in which actors are fired during simulation matter?
  - No.
  - Intuition:
    1. each channel has unique reader $\Rightarrow$ firing an actor cannot disable another actor
    2. each channel has unique writer $\Rightarrow$ firing $A; B$ has same effect as $B; A$
  - Deeper theory: Kahn Process Networks

BUFFER SIZE ANALYSIS

Buffer size analysis

- Consistency and no deadlocks $\Rightarrow$ periodic schedule $\Rightarrow$ queues remain bounded.
- Can we compute the size of buffers we need for each queue?
- What if we want to limit some queue to a pre-determined size?

Computing buffer sizes



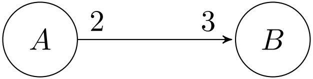
- Graph consistent: $q_A = 1, q_B = 2$
- Keep track of queue size during simulation:

$$(2,0) \xrightarrow{B} (1,1) \xrightarrow{B} (0,2) \xrightarrow{A} (2,0)$$

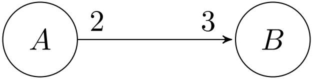
- Max queue size over all simulation steps needed for each queue.

Does buffer size depend on schedule?

Does buffer size depend on schedule?



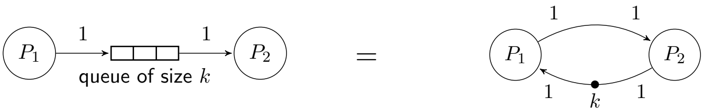
Does buffer size depend on schedule?



Compare schedules: $(AAABB)^\omega$ vs. $(AABAB)^\omega$.

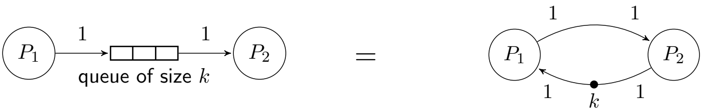
Modeling Finite Queues with Backward Channels

Modeling Finite Queues with Backward Channels



- Each time P_1 needs to write, it must first remove a token from the backward channel.
  - If there are no tokens left then it means that the (forward) queue is full.
- Each time P_2 reads, it puts a token into the backward channel.

Modeling Finite Queues with Backward Channels



- Each time P_1 needs to write, it must first remove a token from the backward channel.
  - If there are no tokens left then it means that the (forward) queue is full.
- Each time P_2 reads, it puts a token into the backward channel.
- Can be easily generalized to m, n tokens produced / consumed. How?

KAHN PROCESS NETWORKS

Kahn Process Networks

- Can be seen as generalization of SDF
  - although Kahn's work [Kahn, 1974] pre-dates SDF [Lee and Messerschmitt, 1987] by more than 10 years
  - Kahn's motivation: distributed systems / parallel programming

Kahn Process Networks

- Can be seen as generalization of SDF
  - although Kahn's work [Kahn, 1974] pre-dates SDF [Lee and Messerschmitt, 1987] by more than 10 years
  - Kahn's motivation: distributed systems / parallel programming
- Main idea:
  - generalize dataflow actors to *Kahn processes*
  - Kahn process: an arbitrary sequential program reading from and writing to queues
    - read is **blocking** (Kahn called it `wait`): if queue is empty, process blocks until queue becomes non-empty
    - write is **non-blocking** (Kahn called it `send`): queues are a-priori unbounded as in SDF

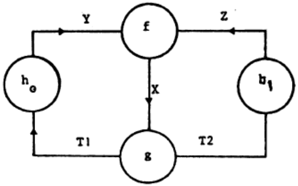
Kahn Process Networks

- Can be seen as generalization of SDF
  - although Kahn's work [Kahn, 1974] pre-dates SDF [Lee and Messerschmitt, 1987] by more than 10 years
  - Kahn's motivation: distributed systems / parallel programming
- Main idea:
  - generalize dataflow actors to *Kahn processes*
  - Kahn process: an arbitrary sequential program reading from and writing to queues
    - read is **blocking** (Kahn called it `wait`): if queue is empty, process blocks until queue becomes non-empty
    - write is **non-blocking** (Kahn called it `send`): queues are a-priori unbounded as in SDF
- Highly recommended reading: [Kahn, 1974] (on bcourses).

Kahn Process Network: Example

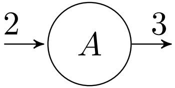
```
 Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
      Begin integer I ; logical B ;
            B := true ;
            Repeat Begin
(4)             I := if B then wait(U) else wait(V) ;
(7)             print (I) ;
(5)             send I on W ;
                B := ¬B ;
                end ;
      End ;
    Process g(integer in U ; integer out V, W) ;
      Begin integer I ; logical B ;
        B := true ;
        Repeat Begin
          I := wait (U) ;
          if B then send I on V else send I on  W ;
          B := ¬B ;
          End ;
      End ;
(3) Process h(integer in U;integer out V; integer INIT);
      Begin integer I ;
        send INIT on V ;
        Repeat Begin
          I := wait(U) ;
          send I on V ;
          End ;
      End ;
  Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1
  End ;
```

Fig.1. Sample parallel program S.



SDF actors are a special case of Kahn processes

SDF actor:

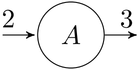


Corresponding Kahn process:

```
Process A(integer in U; integer out V);
Begin integer I1, I2, R1, R2, R3;
    Repeat Begin
        I1 := wait(U);
        I2 := wait(U);
        ... compute R1, R2, R3 ...
        send R1 on V;
        send R2 on V;
        send R3 on V;
    end;
End;
```

Kahn processes are more general than SDF actors

In SDF, token production/consumption rates are **static**:

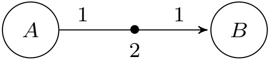


In Kahn processes, they are **dynamic**:

```
   Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
      Begin integer I ; logical B ;
            B := true ;
            Repeat Begin
(4)             I := if B then wait(U) else wait(V) ;
(7)             print (I) ;
(5)             send I on W ;
                B := ¬B ;
                end ;
      End ;
    Process g(integer in U ; integer out V, W) ;
      Begin integer I ; logical B ;
        B := true ;
        Repeat Begin
          I := wait (U) ;
          if B then send I on V else send I on W ;
          B := ¬B ;
          End ;
```

SDF graphs are a special case of Kahn process networks

SDFG:



Kahn process network:

```
Begin

Integer channel X;

Process A(integer out V);
Begin integer R1, R2, R;
    ... compute initial tokens R1,
R2 ...
    send R1 on V;
    send R2 on V;
    Repeat Begin
        ... compute R ...
        send R on V;
    end;
End;

Process B(integer in U);
...
End;

/* main: */
A(X) par B(X)
End;
```

Semantics of Kahn Process Networks

We can give both **operational** and **denotational** semantics.

Operational vs. Denotational Semantics

What is the meaning of a (say, C) program?

- Operational semantics answer: the sequence of steps that the program takes to compute its output from its input
- Denotational semantics answer: a function f that returns the right output for a given input

Semantics of Kahn Process Networks

Operational semantics:

- KPN defines a transition system (similar to the SDF semantics we defined above)
- global state = local states (local vars, program counters, ...) of each process + contents of all queues
- transition: one process makes a move
  - must define some kind of "atomic" moves for processes: for instance, one statement in the sequential program
  - *asynchronous (interleaving)* semantics.

Semantics of Kahn Process Networks

Operational semantics:

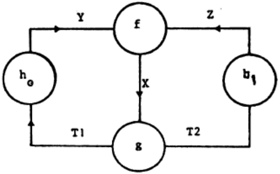
- KPN defines a transition system (similar to the SDF semantics we defined above)
- global state = local states (local vars, program counters, ...) of each process + contents of all queues
- transition: one process makes a move
  - must define some kind of "atomic" moves for processes: for instance, one statement in the sequential program
  - *asynchronous (interleaving)* semantics.

Denotational semantics: this is what we will focus on next.

Denotational Semantics of Kahn Process Networks

General idea:

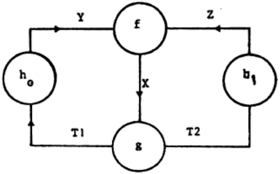
- Each process = a function on **streams**
- An entire (closed) network = a (big) function on (vectors of) streams
- Semantics = the stream computed by the network at every queue



Denotational Semantics of Kahn Process Networks

General idea:

- Each process = a function on **streams**
- An entire (closed) network = a (big) function on (vectors of) streams
- Semantics = the stream computed by the network at every queue



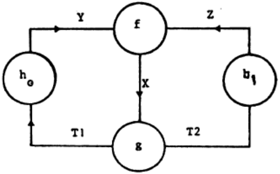
Major benefits:

- Can handle feedback loops in an elegant manner: fixpoint theory

Denotational Semantics of Kahn Process Networks

General idea:

- Each process = a function on **streams**
- An entire (closed) network = a (big) function on (vectors of) streams
- Semantics = the stream computed by the network at every queue



Major benefits:

- Can handle feedback loops in an elegant manner: fixpoint theory
- **Determinacy**: network has a unique solution
  - In terms of operational semantics: order of interleaving does not matter (as long as it is fair)
  - Example: if one execution deadlocks, **all** executions deadlock

Kahn processes as functions on streams: example

What is the stream function associated with this process?

```
(2) Process f(integer  in U,V; integer out W) ;
      Begin integer I ; logical B ;
              B := true ;
              Repeat Begin
(4)               I := if B then wait(U) else wait(V) ;
(7)               print (I) ;
(5)               send I on W ;
                  B := ¬B ;
                  end ;
      End ;
```

Kahn processes as functions on streams: example

What is the stream function associated with this process?

```
Process g(integer in U ; integer out V, W) ;
  Begin integer I ; logical B ;
    B := true ;
    Repeat Begin
      I := wait (U) ;
      if B then send I on V else send I on W ;
      B := ¬B ;
      End ;
  End ;
```

Kahn processes as functions on streams: example

What is the stream function associated with this process?

```
(3) Process h(integer in U;integer out V; integer INIT);
      Begin integer I ;
        send INIT on V ;
        Repeat Begin
          I := wait(U) ;
          send I on V ;
          End ;
      End ;
```

Summary of the rest

Kahn processes are monotonic and continuous functions on streams:

Prefix order on streams: $s \leq s'$ if s is a prefix of s'.

Summary of the rest

Kahn processes are monotonic and continuous functions on streams:

Prefix order on streams: $s \leq s'$ if s is a prefix of s'.

ε: the empty stream (empty sequence).

Summary of the rest

Kahn processes are monotonic and continuous functions on streams:

Prefix order on streams: $s \leq s'$ if s is a prefix of s'.

ε: the empty stream (empty sequence).

Kahn processes: **monotonic** functions on streams.

- The more tokens added to the input, the more added to the output.

Summary of the rest

Kahn processes are monotonic and continuous functions on streams:

Prefix order on streams: $s \leq s'$ if s is a prefix of s'.

ε: the empty stream (empty sequence).

Kahn processes: **monotonic** functions on streams.

- The more tokens added to the input, the more added to the output.

Kahn processes: **continuous** functions on streams.

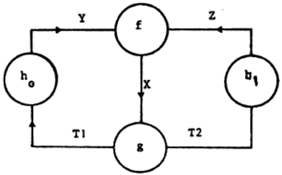
- A process cannot "wait forever" to produce an output.

Summary of the rest

Theorem

A continuous function $f : (D^{**})^n \to (D^{**})^n$ *has a least fixpoint* s*.*
Moreover, $\mathrm{s} = \lim_{n\to\infty} f^n(\varepsilon, \varepsilon, ..., \varepsilon)$*.*

The denotational semantics of a Kahn process network K is the least fixpoint of the corresponding function f_K.



Note: the least fixpoint may contain ε or infinite streams.

Streams

- Stream $=$ a finite or infinite sequence of values.
- D: set of values
- D^*: set of all finite sequences of values from D
- D^ω: set of all infinite sequences of values from D
- $D^{**} = D^* \cup D^\omega$

Streams

- Stream = a finite or infinite sequence of values.

- D: set of values
- D^*: set of all finite sequences of values from D
  - This includes the empty sequence ε
- D^ω: set of all infinite sequences of values from D

- $D^{**} = D^* \cup D^\omega$

Streams

- Stream $=$ a finite or infinite sequence of values.

- D: set of values
- D^*: set of all finite sequences of values from D
  - This includes the empty sequence ε
- D^ω: set of all infinite sequences of values from D
  - Note: $D^\omega = D^{\mathbb{N}} =$ set of all total functions from $\mathbb{N}$ to D
- $D^{**} = D^* \cup D^\omega$

Streams: Examples

Let $D = \{0, 1\}$.
Consider the streams:

$$0 \qquad \in D^*$$

Streams: Examples

Let $D = \{0, 1\}$.
Consider the streams:

$$\begin{aligned} 0 \qquad & \in D^* \\ 00000 = 0^5 \qquad & \in D^* \end{aligned}$$

Streams: Examples

Let $D = \{0, 1\}$.
Consider the streams:

$$\begin{aligned} 0 & \qquad \in D^* \\ 00000 = 0^5 & \qquad \in D^* \\ 010101 \cdots = (01)^\omega & \qquad \in D^\omega \end{aligned}$$

Streams: Examples

Let $D = \{0, 1\}$.
Consider the streams:

$$
\begin{aligned}
0 \qquad & \in D^* \\
00000 = 0^5 \qquad & \in D^* \\
010101 \cdots = (01)^\omega \qquad & \in D^\omega \\
0100100010^4 10^5 1 \cdots \qquad & \in D^\omega
\end{aligned}
$$

Prefix Order on Streams

Let $s \cdot s'$ denote stream concatenation:

- if $s' = \varepsilon$, then $s \cdot s' = s$ (s could be infinite in this case)
- if $s' \neq \varepsilon$, then s must be finite (s' could be finite or infinite)

s_1 is a **prefix** of s_2, denoted $s_1 \leq s_2$, iff

$$\exists s_3 : s_2 = s_1 \cdot s_3$$

Note that s_3 may be the empty string, in which case $s_1 = s_2$

- so $s \leq s$ for any stream s

Prefix Order on Streams

Let $s \cdot s'$ denote stream concatenation:

- if $s' = \varepsilon$, then $s \cdot s' = s$ (s could be infinite in this case)
- if $s' \neq \varepsilon$, then s must be finite (s' could be finite or infinite)

s_1 is a **prefix** of s_2, denoted $s_1 \leq s_2$, iff

$$\exists s_3 : s_2 = s_1 \cdot s_3$$

Note that s_3 may be the empty string, in which case $s_1 = s_2$

- so $s \leq s$ for any stream s

Examples:

$$000 \quad \leq \quad 0001$$

Prefix Order on Streams

Let $s \cdot s'$ denote stream concatenation:

- if $s' = \varepsilon$, then $s \cdot s' = s$ (s could be infinite in this case)
- if $s' \neq \varepsilon$, then s must be finite (s' could be finite or infinite)

s_1 is a **prefix** of s_2, denoted $s_1 \leq s_2$, iff

$$\exists s_3 : s_2 = s_1 \cdot s_3$$

Note that s_3 may be the empty string, in which case $s_1 = s_2$

- so $s \leq s$ for any stream s

Examples:

$$\begin{array}{lcl} 000 & \leq & 0001 \\ 000 & \leq & 0001^\omega \end{array}$$

Prefix Order on Streams

Let $s \cdot s'$ denote stream concatenation:

- if $s' = \varepsilon$, then $s \cdot s' = s$ (s could be infinite in this case)
- if $s' \neq \varepsilon$, then s must be finite (s' could be finite or infinite)

s_1 is a **prefix** of s_2, denoted $s_1 \leq s_2$, iff

$$\exists s_3 : s_2 = s_1 \cdot s_3$$

Note that s_3 may be the empty string, in which case $s_1 = s_2$

- so $s \leq s$ for any stream s

Examples:

$$\begin{array}{ccc} 000 & \leq & 0001 \\ 000 & \leq & 0001^{\omega} \\ 000 & \not\leq & 0010 \end{array}$$

Prefix Order on Streams

Let $s \cdot s'$ denote stream concatenation:

- if $s' = \varepsilon$, then $s \cdot s' = s$ (s could be infinite in this case)
- if $s' \neq \varepsilon$, then s must be finite (s' could be finite or infinite)

s_1 is a **prefix** of s_2, denoted $s_1 \leq s_2$, iff

$$\exists s_3 : s_2 = s_1 \cdot s_3$$

Note that s_3 may be the empty string, in which case $s_1 = s_2$

- so $s \leq s$ for any stream s

Examples:

$$\begin{array}{rcl} 000 & \leq & 0001 \\ 000 & \leq & 0001^\omega \\ 000 & \not\leq & 0010 \\ 000\cdots = 0^\omega & \not\leq & 10^\omega = 1000\cdots \end{array}$$

Prefix order is a partial order

$\leq$ is a **partial order** on D^{**}, i.e., it is

- *reflexive* : $\forall s : s \leq s$
- *antisymmetric* : $\forall s, s' : s \leq s' \wedge s' \leq s \Rightarrow s = s'$
- *transitive* : $\forall s, s', s'' : s \leq s' \wedge s' \leq s'' \Rightarrow s \leq s''$

Prefix order is a partial order

$\leq$ is a **partial order** on D^{**}, i.e., it is

- *reflexive* : $\forall s : s \leq s$
- *antisymmetric* : $\forall s, s' : s \leq s' \wedge s' \leq s \Rightarrow s = s'$
- *transitive* : $\forall s, s', s'' : s \leq s' \wedge s' \leq s'' \Rightarrow s \leq s''$

- The pair $(D^{**}, \leq)$ is a *poset* (partially-ordered set).
- In fact, it is a **CPO** (a complete poset) because it also satisfies

for any increasing chain $s_0 \leq s_1 \leq s_2 \leq \cdots$: $\lim_{n \to \infty} s_n$ is in D^{**}

Prefix order is a partial order

$\leq$ is a **partial order** on D^{**}, i.e., it is

- *reflexive* : $\forall s : s \leq s$
- *antisymmetric* : $\forall s, s' : s \leq s' \wedge s' \leq s \Rightarrow s = s'$
- *transitive* : $\forall s, s', s'' : s \leq s' \wedge s' \leq s'' \Rightarrow s \leq s''$

- The pair $(D^{**}, \leq)$ is a *poset* (partially-ordered set).
- In fact, it is a **CPO** (a complete poset) because it also satisfies

  for any increasing chain $s_0 \leq s_1 \leq s_2 \leq \cdots \; : \lim_{n\to\infty} s_n$ is in D^{**}

- It has a least element $\perp$ (which one is it?) such that $\forall s : \perp \leq s$.

Prefix order is a partial order

$\leq$ is a **partial order** on D^{**}, i.e., it is

- *reflexive* : $\forall s : s \leq s$
- *antisymmetric* : $\forall s, s' : s \leq s' \wedge s' \leq s \Rightarrow s = s'$
- *transitive* : $\forall s, s', s'' : s \leq s' \wedge s' \leq s'' \Rightarrow s \leq s''$

- The pair $(D^{**}, \leq)$ is a *poset* (partially-ordered set).
- In fact, it is a **CPO** (a complete poset) because it also satisfies

  for any increasing chain $s_0 \leq s_1 \leq s_2 \leq \cdots$: $\lim_{n\to\infty} s_n$ is in D^{**}

- It has a least element $\perp$ (which one is it?) such that $\forall s : \perp \leq s$.

  $\perp = \varepsilon$ (the empty sequence)

Functions on Streams

Function on streams (single-input / single-output):

$$f : D^{**} \rightarrow D^{**}$$

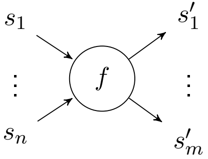
Functions on Streams

Function on streams (single-input / single-output):

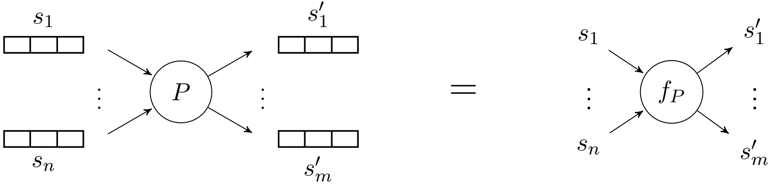
$$f : D^{**} \to D^{**}$$

Function on streams (general):

$$f : (D^{**})^n \to (D^{**})^m$$



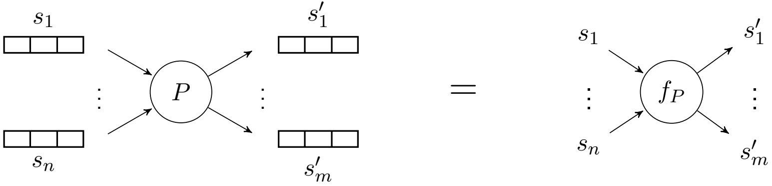
Kahn Processes = Functions on Streams



Basic idea:

- Fix the contents of input queues to $s_1, ..., s_n$.
- Let the process P run.
- Observe the contents of output queues: $s'_1, ..., s'_m$.

Kahn Processes = Functions on Streams



Basic idea:

- Fix the contents of input queues to $s_1, ..., s_n$.
- Let the process P run.
- Observe the contents of output queues: $s'_1, ..., s'_m$.
- Notice that Kahn processes are deterministic programs.

Monotonic Functions on Streams

A stream function $f : D^{**} \to D^{**}$ is **monotonic** (w.r.t. $\leq$) iff

$$\forall s, s' \in D^{**} : s \leq s' \Rightarrow f(s) \leq f(s')$$

Monotonic Functions on Streams

A stream function $f : D^{**} \to D^{**}$ is **monotonic** (w.r.t. $\leq$) iff

$$\forall s, s' \in D^{**} : s \leq s' \Rightarrow f(s) \leq f(s')$$

Stream functions defined by Kahn processes are monotonic. Why?

Monotonic Functions on Streams

A stream function $f : D^{**} \to D^{**}$ is **monotonic** (w.r.t. $\leq$) iff

$$\forall s, s' \in D^{**} : s \leq s' \Rightarrow f(s) \leq f(s')$$

Stream functions defined by Kahn processes are monotonic. Why?

- Once something is written to an output queue, it cannot be taken back.
- More inputs $\Rightarrow$ more outputs.

Continuous Functions on Streams

A stream function $f : D^{**} \to D^{**}$ is **continuous** (w.r.t. $\leq$) if f is monotonic and for any increasing chain $s_0 \leq s_1 \leq s_2 \leq \cdots$

$$f(\lim_{n\to\infty} s_n) = \lim_{n\to\infty} f(s_n)$$

Note: by monotonicity of f, and the fact that $s_0 \leq s_1 \leq s_2 \leq \cdots$ is a chain, $f(s_0) \leq f(s_1) \leq f(s_2) \leq \cdots$ is also a chain, so

$$\lim_{n\to\infty} f(s_n)$$

is well-defined.

Continuous Functions on Streams

By definition continuity implies monotonicity.[1]

But: monotonicity does not generally imply continuity.

Quiz: Can you think of

- a non-monotonic stream function?
- a non-continuous stream function?
- a monotonic but non-continuous stream function?

[1] Alternative definitions of continuity exist that imply monotonicity. See good textbook on order theory: [Davey and Priestley, 2002].

Continuous Functions on Streams

Stream functions defined by Kahn processes are continuous.

Quiz: Why?

Continuous Functions on Streams

Stream functions defined by Kahn processes are continuous.

Quiz: Why?

Kahn processes cannot "take forever" to produce an output. Every output they produce must be produced because of the finite sequence of inputs the process has read so far. If for all finite sequences the process produces no outputs, then the process produces no outputs at all, even for the infinite sequence.

Monotonic and Continuous Functions on Streams

The notions of monotonicity and continuity extend to functions of arbitrary arity:

$$f : (D^{**})^n \to (D^{**})^m$$

Basic idea:

- "Lift" $\leq$ to vectors element-wise

$$(s_1, s_2, ..., s_n) \leq (s'_1, s'_2, ..., s'_n)$$

iff

$$s_1 \leq s'_1 \wedge s_2 \leq s'_2 \wedge \cdots \wedge s_n \leq s'_n$$

Fixpoint Theorem

Theorem

A continuous function $f : (D^{**})^n \to (D^{**})^n$ *has a least fixpoint* s*.*
Moreover, $\mathrm{s} = \lim_{n\to\infty} f^n(\varepsilon, \varepsilon, ..., \varepsilon)$*.*

Least fixpoint s means:

- s is a fixpoint: $f(\mathrm{s}) = \mathrm{s}$.
- s is a least fixpoint: for any other fixpoint $s' = f(s')$, $\mathrm{s} \leq s'$.

Note: least implies s is unique.

Fixpoint Theorem

Theorem

A continuous function $f : (D^{**})^n \to (D^{**})^n$ *has a least fixpoint* s*.*
Moreover, $\mathrm{s} = \lim_{n\to\infty} f^n(\varepsilon, \varepsilon, ..., \varepsilon)$*.*

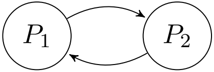
Least fixpoint s means:

- s is a fixpoint: $f(\mathrm{s}) = \mathrm{s}$.
- s is a least fixpoint: for any other fixpoint $s' = f(s')$, $\mathrm{s} \leq s'$.

Note: least implies s is unique.

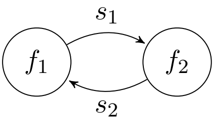
How is s related to the semantics of Kahn process networks?

From Process Network to Fixpoint Equations



process network

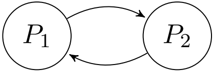
=



fixpoint equations

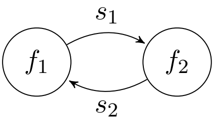
$$
\begin{aligned}
s_1 &= f_1(s_2) \\
s_2 &= f_2(s_1)
\end{aligned}
$$

From Process Network to Fixpoint Equations



process network

=



fixpoint equations

$$
\begin{aligned}
s_1 &= f_1(s_2) \\
s_2 &= f_2(s_1)
\end{aligned}
$$

Can be rewritten as:

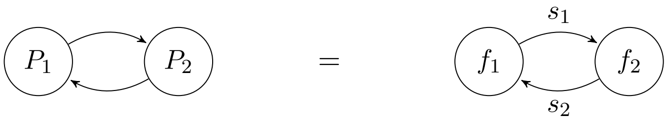
$$(s_1, s_2) = f(s_1, s_2)$$

where $f : (D^{**})^2 \to (D^{**})^2$ is defined as:

$$f(s_1, s_2) \mathrel{\widehat{=}} \big(f_1(s_2), f_2(s_1)\big)$$

Putting it all together

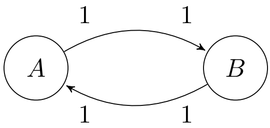
The denotational semantics of the Kahn process network



is the unique least fixpoint $(\mathbf{s_1}, \mathbf{s_2})$ of the set of equations:

$$
\begin{aligned}
s_1 &= f_1(s_2) \\
s_2 &= f_2(s_1)
\end{aligned}
$$

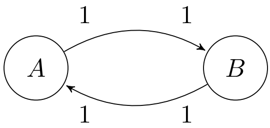
Example: denotational semantics of SDF graphs



Viewing A and B as functions on streams of tokens:

$$
\begin{aligned}
&A(\varepsilon) = B(\varepsilon) = \varepsilon \\
&A(\bullet) = B(\bullet) = \bullet \\
&A(\bullet\bullet) = B(\bullet\bullet) = \bullet\bullet \\
&A(\bullet\bullet\bullet) = B(\bullet\bullet\bullet) = \bullet\bullet\bullet \\
&\ldots
\end{aligned}
$$

Example: denotational semantics of SDF graphs



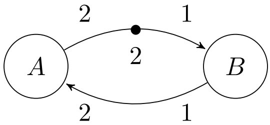
Viewing A and B as functions on streams of tokens:

$$
\begin{aligned}
&A(\varepsilon) = B(\varepsilon) = \varepsilon \\
&A(\bullet) = B(\bullet) = \bullet \\
&A(\bullet\bullet) = B(\bullet\bullet) = \bullet\bullet \\
&A(\bullet\bullet\bullet) = B(\bullet\bullet\bullet) = \bullet\bullet\bullet \\
&\ldots
\end{aligned}
$$

Computing the fixpoint:

$$f(\varepsilon, \varepsilon) \quad = \quad (A(\varepsilon), B(\varepsilon)) \quad = \quad (\varepsilon, \varepsilon)$$

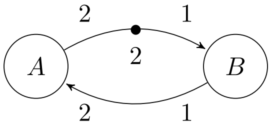
Example: denotational semantics of SDF graphs



Viewing A as a function on streams of tokens (B is as before):

$$
\begin{array}{rcll}
A(\varepsilon) & = & \bullet\,\bullet & \text{// this captures initial tokens} \\
A(\bullet) & = & \bullet\,\bullet & \\
A(\bullet\bullet) & = & \bullet\bullet\bullet\bullet & \\
A(\bullet\bullet\bullet) & = & \bullet\bullet\bullet\bullet & \\
A(\bullet\bullet\bullet\bullet) & = & \bullet\bullet\bullet\bullet\bullet\bullet & \\
\ldots & & &
\end{array}
$$

Example: denotational semantics of SDF graphs



Viewing A as a function on streams of tokens (B is as before):

$$
\begin{array}{rcll}
A(\varepsilon) & = & \bullet\bullet & \text{// this captures initial tokens} \\
A(\bullet) & = & \bullet\bullet & \\
A(\bullet\bullet) & = & \bullet\bullet\bullet\bullet & \\
A(\bullet\bullet\bullet) & = & \bullet\bullet\bullet\bullet & \\
A(\bullet\bullet\bullet\bullet) & = & \bullet\bullet\bullet\bullet\bullet\bullet & \\
\cdots & & &
\end{array}
$$

Computing the fixpoint:

$$
\begin{array}{rcll}
f(\varepsilon,\varepsilon) & = & (A(\varepsilon), B(\varepsilon)) & = (\bullet\bullet, \varepsilon) \\
f(\bullet\bullet,\varepsilon) & = & (A(\varepsilon), B(\bullet\bullet)) & = (\bullet\bullet, \bullet\bullet) \\
f(\bullet\bullet,\bullet\bullet) & = & (A(\bullet\bullet), B(\bullet\bullet)) & = (\bullet\bullet\bullet\bullet, \bullet\bullet) \\
\ldots & & \text{fixpoint is } (\bullet^\omega, \bullet^\omega). &
\end{array}
$$

Assessment

Why get excited about Kahn process networks?

- Elegant mathematics
  - (to compare, try to define formal operational semantics, or see papers that do that, e.g., [Lynch and Stark, 1989, Jonsson, 1994])

Assessment

Why get excited about Kahn process networks?

- Elegant mathematics
  - (to compare, try to define formal operational semantics, or see papers that do that, e.g., [Lynch and Stark, 1989, Jonsson, 1994])
- Caveats:
  - Still need to relate denotational to some operational semantics, to convince ourselves that they are equivalent [Lynch and Stark, 1989, Jonsson, 1994].
  - Framework difficult to extend to non-deterministic processes – c.f. famous "Brock-Ackerman anomaly" and related literature [Brock and Ackerman, 1981, Jonsson, 1994]

Assessment

Why get excited about Kahn process networks?

- Foundations of asynchronous message-passing paradigm.
- Deterministic concurrency.
  - Writing/debugging concurrent programs made easier.
  - Contrast this to threads [Lee, 2006].

Assessment

Why get excited about Kahn process networks?

- Foundations of asynchronous message-passing paradigm.
- Deterministic concurrency.
  - Writing/debugging concurrent programs made easier.
  - Contrast this to threads [Lee, 2006].
  - How is determinism achieved in Kahn Process Networks?
  - No shared memory! Channels have unique writer/reader processes.
  - Blocking read.
  - No "peeking" into input queues allowed, no removing data already written into output queues, etc.

Assessment

Why get excited about Kahn process networks?

- Foundations of asynchronous message-passing paradigm.
- Deterministic concurrency.
    - Writing/debugging concurrent programs made easier.
    - Contrast this to threads [Lee, 2006].
    - How is determinism achieved in Kahn Process Networks?
    - No shared memory! Channels have unique writer/reader processes.
    - Blocking read.
    - No "peeking" into input queues allowed, no removing data already written into output queues, etc.

Criticism: KPN use infinite queues but these do not exist in reality.

Assessment

Why get excited about Kahn process networks?

- Foundations of asynchronous message-passing paradigm.
- Deterministic concurrency.
  - Writing/debugging concurrent programs made easier.
  - Contrast this to threads [Lee, 2006].
  - How is determinism achieved in Kahn Process Networks?
  - No shared memory! Channels have unique writer/reader processes.
  - Blocking read.
  - No "peeking" into input queues allowed, no removing data already written into output queues, etc.

Criticism: KPN use infinite queues but these do not exist in reality.

Answer: finite queues can easily be modeled in KPN (also in SDF). How?

Kahn Process Networks vs. Petri Nets

[Ignore if you don't know what Petri nets are]

Petri nets:

- More abstract model: processes = transitions.
- Non-deterministic:
  - Each place can have many incoming / outgoing transitions.
  - Each place may be shared by multiple producers / consumers.

Bibliography

Brock, J. and Ackerman, W. (1981).
Scenarios: A model of non-determinate computation.
In *Proc. Intl. Colloq. on Formalization of Programming Concepts*, pages 252–259, London, UK. Springer-Verlag.

Davey, B. A. and Priestley, H. A. (2002).
Introduction to Lattices and Order.
Cambridge University Press, 2nd edition.

Jonsson, B. (1994).
Compositional specification and verification of distributed systems.
ACM Trans. Program. Lang. Syst., 16(2):259–303.

Kahn, G. (1974).
The semantics of a simple language for parallel programming.
In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland.

Lee, E. (2006).
The problem with threads.
IEEE Computer, 39(5):33–42.

Lee, E. and Messerschmitt, D. (1987).
Synchronous data flow.
Proceedings of the IEEE, 75(9):1235–1245.

Lynch, N. and Stark, E. (1989).
A Proof of the Kahn Principle for Input/Output Automata.
Information and Computation, 82:81–92.