Synchronous Data Flow Graphs

Synchronous Data Flow (SDF) graphs refer to systems where the number of tokens consumed and produced per actor firing is ***fixed*** and ***constant***

The term *synchronous* refers to the **fixed** consumption and production rate of tokens

Note that SDF will **not** be able to handle *control-flow* constructs, such as *if-then-else* statements in C without adding special operators (which we will discuss)

Despite this significant limitation, SDFs are very powerful (and popular), and more importantly, mathematical techniques can be used to verify certain properties

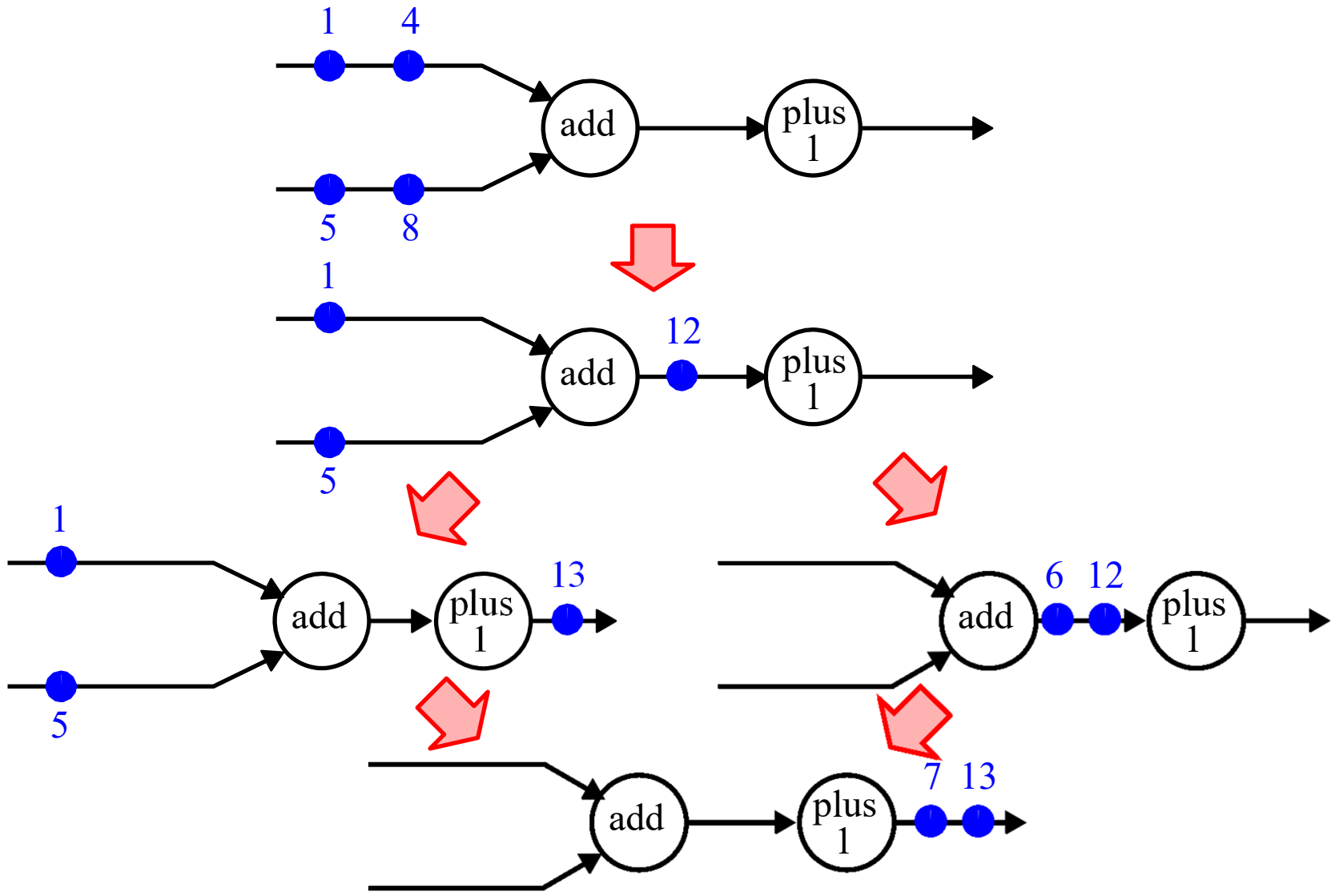
The first of these properties is **determinism**

The entire SDF is deterministic under the condition that all of its *actors* implement a deterministic function

Determinism guarantees that the same results will always be produced **independent of the firing order**

SDF Graphs

Illustration of determinism:



SDF Graphs

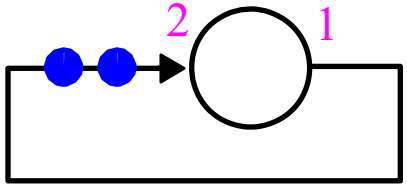
A significant benefit of determinism is that it allows arbitrary mappings of the *actors* onto parallel architectures while guaranteeing the same results

For example, correct results are obtained even if we execute the *add actor* on a fast processor and the *plus1 actor* on a slow processor

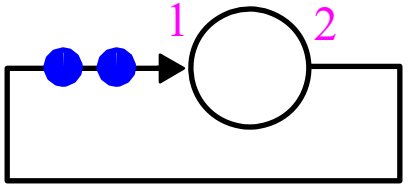
The second important property of SDF relates to an ***admissible schedule***

An **admissible** SDF is one that can run forever without *deadlock* (unbounded execution) or without overflowing any of the communication queues (bounded buffer)

Deadlock occurs when an SDF graph progresses to marking that prevents firings



Graph is deadlocked



Infinite # of tokens produced

Overflow occurs when tokens are produced faster than they are consumed

SDF Graphs

There is also a systematic method to determine whether a SDF graph is *admissible*

The method provides a closed form solution, i.e., no simulation is required

Lee proposed a method called **Periodic Admissible Schedules** (PASS), defined as:

- A *schedule* is the order in which the actors must fire
- An *admissible schedule* is a firing order that is deadlock-free with bounded buffers
- A *periodic admissible schedule* is a schedule that supports *unbounded execution*, i.e., is periodic in the sense that the same markings will recur

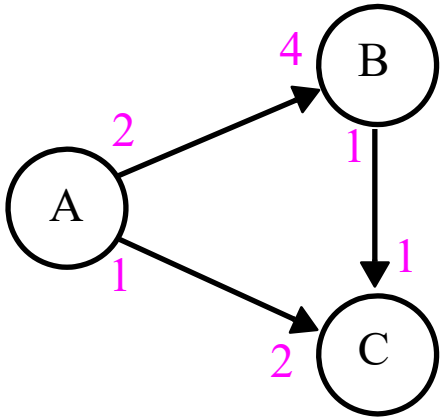
We also consider a special case called *Periodic Admissible* ***Sequential*** *Schedules* (PASSs) that supports a microprocessor implementation with one *actor* firing at a time

There are four steps to creating a PASS for an SDF graph:

- Create the topology matrix **G** of the SDF graph
- Verify the *rank* of the matrix to be one less than the number of nodes in the graph
- Determine a firing vector
- Try firing each actor in a *round robin* fashion, until the *firing count* given by the firing vector is reached

SDF Graphs

Consider the following example:



Step 1: Create a topology matrix for this graph:

The topology matrix has as many rows as there are *edges* (FIFO queues) and as many columns as there are *nodes*

The entry (*i*, *j*) will be positive if the node *j* **produces** tokens onto the edge *i* and negative if it consumes tokens

$$G = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{array}{l} \leftarrow edge(A,B) \\ \leftarrow edge(A,C) \\ \leftarrow edge(B,C) \end{array}$$

NOTE: This matrix do NOT need to be square

SDF Graphs

Step 2: The condition for a PASS to exist is that the *rank* of **G** has to be one less than the number of nodes in the graph

The *rank* of the matrix is the number of **independent equations** in **G**

For our graph, the rank is 2 -- verify by multiplying the first column by -2 and the second column by -1, and adding them to produce the third column

$$G = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \quad \Rightarrow \quad G = \begin{bmatrix} -4 & +4 & 0 \\ -2 & 0 & -2 \\ 0 & -1 & -1 \end{bmatrix}$$

Given that there are *three* nodes in the graph and the rank of the matrix is 2, a PASS is **possible**

This step effectively verifies that tokens can **NOT** accumulate on any edge of the graph

The actual number of tokens can be determined by choosing a firing vector and carrying out a matrix multiplication

SDF Graphs

For example, the tokens produced/consumed by firing A twice and B and C zero times is given by:

$$\text{firing vector } q = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \Rightarrow Gq = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 0 \end{bmatrix}$$

This vector produces 4 tokens on edge(A,B) and 2 tokens on edge(A,C)

Step 3: Determine a periodic firing vector

The *firing vector* given above is not a good choice to obtain a PASS because it leaves tokens in the system

We are instead interested in a firing vector that leaves no tokens:

$$Gq_{\text{PASS}} = 0$$

Note that since the *rank* is less than the number of nodes, there are an infinite number of solutions to the matrix equation

SDF Graphs

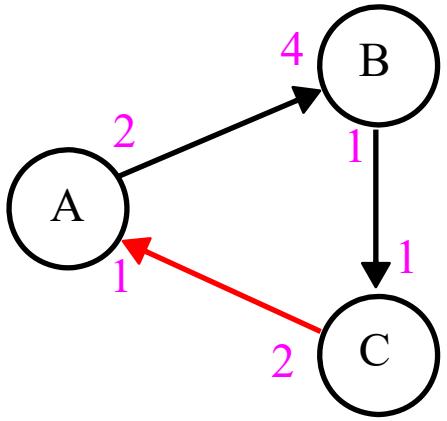
Step 3: Determine a periodic firing vector (cont.)

This is true b/c, intuitively, if *firing vector* (*a, b, c*) is a PASS, then so should be firing vectors (*2a, 2b, 2c*), (*3a, 3b, 3c*), etc.

Our task is to find the simplest one -- for this example, it is:

$$q_{\text{PASS}} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \Rightarrow Gq_{\text{PASS}} = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Note that the existence of a PASS firing vector does **not** guarantee that a PASS will also exist



Here, we reversed the (A,C) edge

We would find the same q_{PASS} but the resulting graph is **deadlocked** -- all nodes are waiting for each other

SDF Graphs

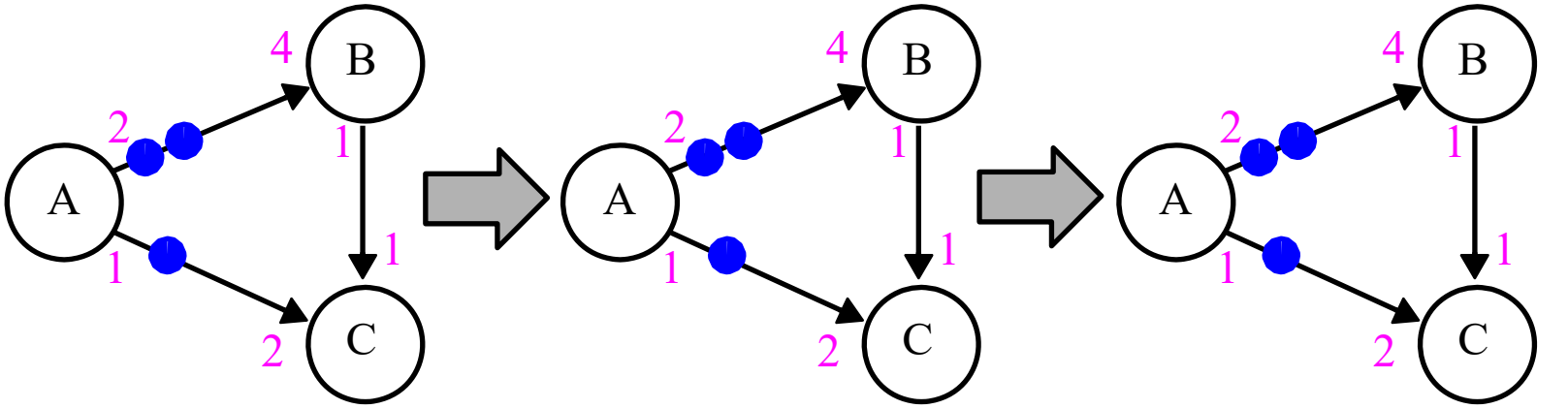
Step 4: Construct a *valid* PASS.

Here, we fire each node up to the number of times specified in q_{PASS}

Each node that is able to fire, i.e., has an adequate number of tokens, will fire If

we find that we can fire NO more nodes, and the firing count is **less** than the number in q_{PASS}, the resulting graph is **deadlocked**

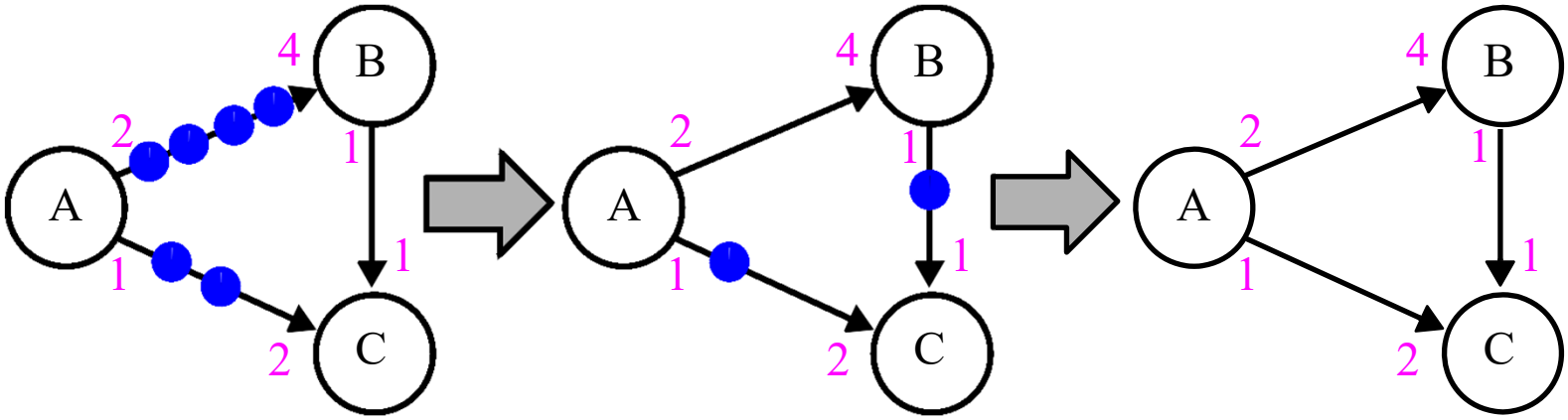
Trying this out on our graph, we fire *A* once, and then B and C



Fire *A* (succeeds) Fire *B* (FAILS -- not enough tokens) Fire *C* (FAILS)

SDF Graphs

Step 4: Construct a *valid* PASS.



Fire *A* AGAIN (succeeds) Fire *B* (succeeds) Fire *C* (succeeds)

So the PASS is (*A*, *A*, *B*, *C*)

Try this out on the **deadlocked** graph -- it aborts immediately on the first iteration because no node is able to fire successfully

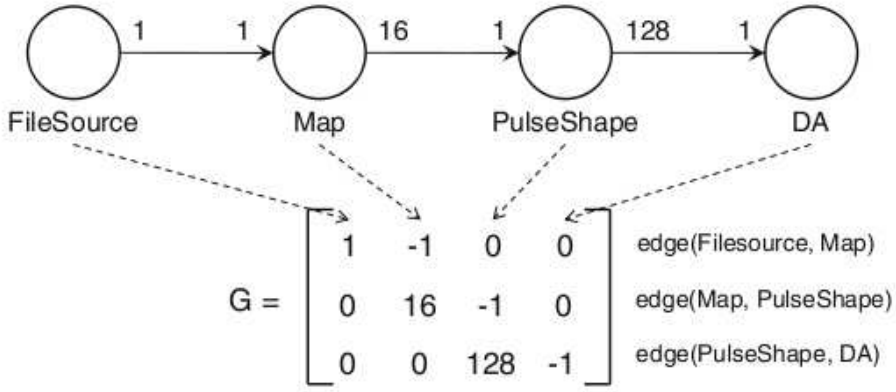
Note that the **determinate** property allows any ordering to be tried freely, e.g., *B*, *C* and then *A*

In some graphs (not ours), this may lead to additional PASS solutions

SDF Graphs: PAM-4 Example

Consider the *digital pulse-amplitude modulation system* (PAM-4) discussed earlier

The SDF for this system consists of 4 *actors*, and is a *multi-rate* Data Flow system:



The first step is to construct the *topology matrix G*

The *queues* correspond to the 3 rows and *actors* to the 4 columns

The second step is to verify the rank is the number of *actors* minus 1

It is easy to show that the 3 rows are independent, i.e., are not linear combina- tions of any other rows

This confirms that a PASS is possible

SDF Graphs: PAM-4 Example

The third step is to derive a feasible firing for the system

The firing vector, q_{PASS}, must yield a zero-vector when multiplied by the topology matrix

$$q_{\text{PASS}} = \begin{bmatrix} 1 \\ 1 \\ 16 \\ 2048 \end{bmatrix} \Rightarrow Gq_{\text{PASS}} = \begin{bmatrix} +1 & -1 & 0 & 0 \\ 0 & +16 & -1 & 0 \\ 0 & 0 & +12 & -\frac{1}{8} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 16 \\ 2048 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The fourth step is to derive a schedule -- there are two possibilities

- The first one is trivial, fire each *actor* in succession, from left to right Note that the *queue* (FIFO) sizes are 16 and 2048
- Alternatively, we can fire *FileSource* and *Map* once and then repeat the following sequence: Fire *PulseShape* once and then fire *DA* 128 times

  The benefit here is the reduced *queue* sizes, i.e., the *PulseShape* input *queue* reduces from 16 to 1 while the *DA* input *queue* reduces from 2048 to 128

In general, deriving the optimal schedule is a difficult problem for complex systems

Limits of SDF Models

In conclusion, SDFs are very useful

They allow a designer to determine certain important system properties, such as the *determinism*, *deadlock*, and *storage requirements*

Unfortunately, SDFs are **not** a universal specification mechanism, in particular, SDFs do not have constructs to allow **control-flow** modeling

Control appears in different forms in system design:

- Stopping and re-starting: An SDF model runs forever

  Stopping/re-starting is a control-flow property not addressed with SDFs
- Mode-switching: When a cell-phone switches from one standard to the other, the baseband processing (modeled as an SDF) needs to be reconfigured

  The topology of an SDF graph is fixed and **cannot** be modified at runtime
- Exceptions: Error conditions arrise in applications

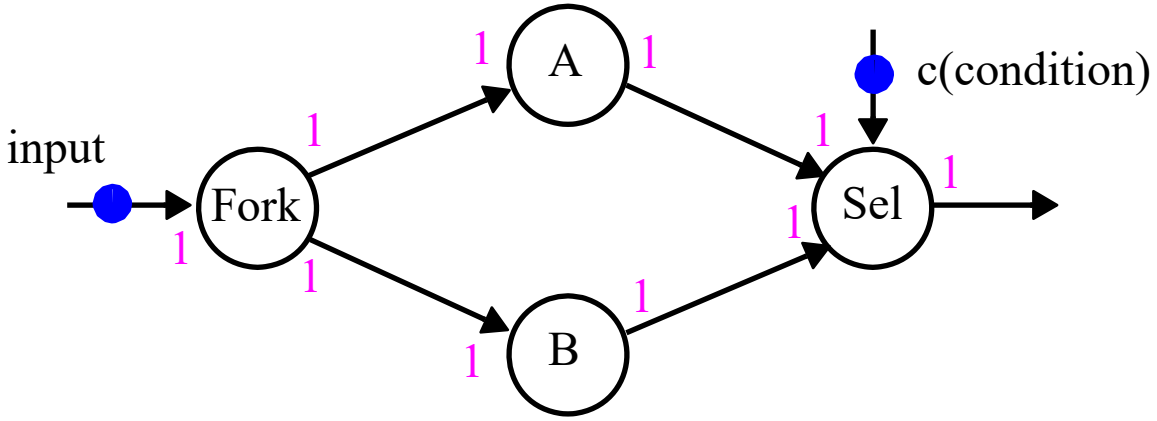
  SDFs cannot model exceptions that affect the entire graph, e.g., empty queues
- Run-time conditions: A simple *if-then-else* stmt cannot be modeled by SDFs An SDF node does not support conditional execution -- it is always active

Limits of SDF Models

There are two solutions to the problem of *control flow modeling* in SDFs

Solution 1: *emulate* control flow on top of the SDF semantics

Consider the stmt *if (c) then A else B*



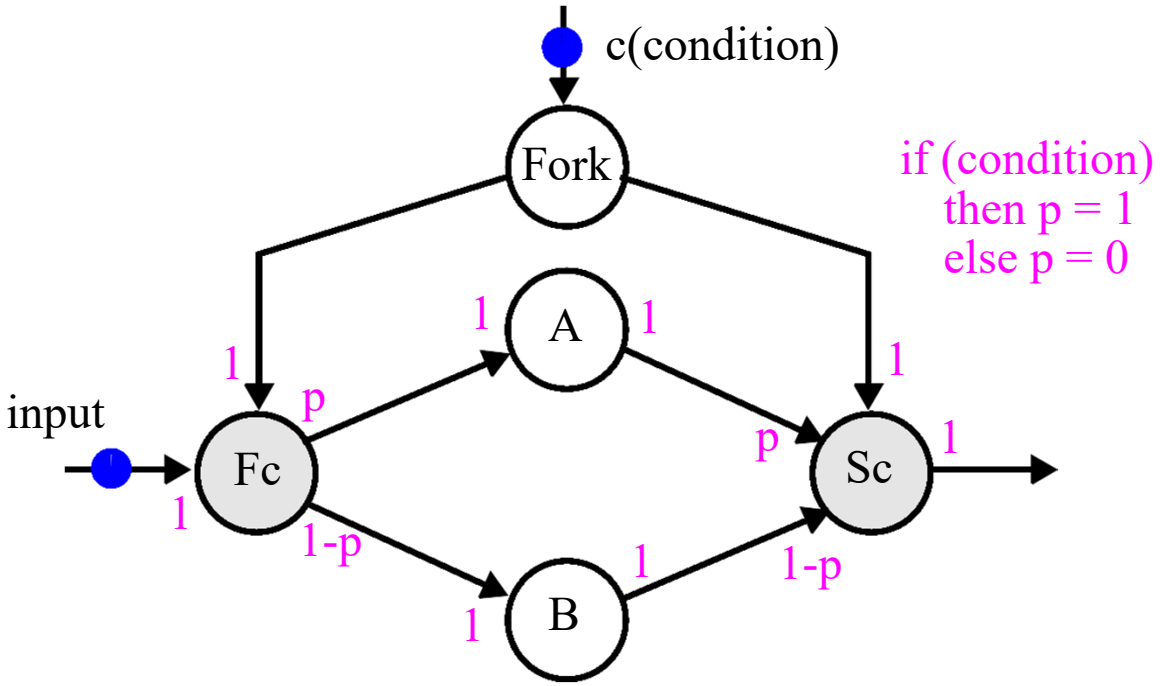
The *selector-actor* on the right routes either *A* or *B* to the output

Note that this is not an exact match to the *if-then-else* in C because BOTH the *if* branch (*A*) and the *else* (*B*) must execute and produce tokens

However, it is a good match to hardware, which uses a *multiplexer* to select among one of several input results

Limits of SDF Models

Solution 2: *extend* the SDF semantics using *Boolean Data Flow* (BDF)



BDFs 'tune' the *production* and *consumption* rate of a *actor* according to the value of an **external control token**

The *condition* token is distributed to two BDF **conditional fork** and **merge** nodes, *Fc* and *Sc*

Limits of SDF Models

Here, the *conditional fork* will fire when there is an *input* token AND a *condition* token

A token is produced on EITHER the upper or lower edge, dependent on the *condition* token

This is indicated by a dynamic variable *p*, which signifies a *conditional production rate*

The *conditional merge* works similarly, i.e., it fires when there is a *condition* token and will consume a token on EITHER the upper or lower edge

Unfortunately, BDFs detract from the usefulness of SDFs

For example, we now have Data Flow graphs that are *conditionally admissible*

Also, the topology matrix now includes symbolic values, *p*, which complicates the closed form math

For a SDF with 5 conditions, we would have a matrix with 5 symbols or would need to expand the single matrix into 32 variants (2^5)

Limits of SDF Models

Beyond BDF, other flavors of *control-oriented* Data Flow graphs have been proposed that have similar challenges, such as:

- Dynamic Data Flow (DDF) which allows variable production and consumption rates
- Cyclo-Static Data Flow (CSDF) which allows a fixed, iterative variation on produc- tion and consumption rates