Chapter 3-3: MoC and Programming

Soo-Ik Chae

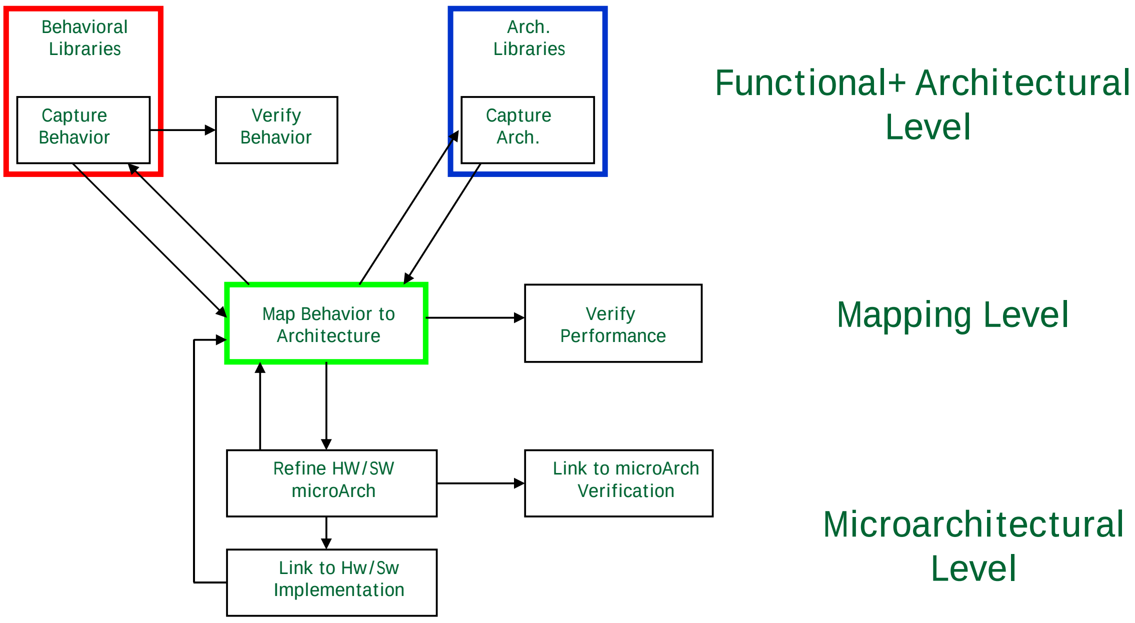
Topics

- Models of computation and programming.

Design Approach

- Start design process *before* hw-sw partitioning
- Sequence of steps are vital
  - system specification unbiased to implementation
    - describe system behavior at high level
  - Initial functional design
  - verification
  - mapping to target architecture
- Thus, function-architecture codesign is key approach

System design flow



Design conception to design description

- At functional level, behavior of a system to be implemented is selected and analyzed against a set of specifications
  - Specifications vs.. behavior?
    - Specs: I/O relation, set of constraints, system goals
    - Behavior: algorithm to realize the function
- Specs vs algorithm
  - Algorithm is the result of implementation decision (purists view)
  - Specs: algorithm itself! (another view)

Algorithm Design and Analysis

- Algorithm development: key aspect of system design at functional level
  - Relatively little work has been done on selection of algorithm based on specifications
  - Must have strong correctness properties in critical operations
- Algorithm analysis is more general concept than simulation
- It is important to decide on mathematical model for designer that will support algorithm analysis

Algorithm Implementation

- Need of intermediate steps that transform an algorithm to a set of tractable functional components
- The functional components are to be formally defined to capture the algorithm's properties
- MoC is a key answer to the above!
- Selection of MoC is to be done carefully.

Introduction to MoC

- System design goals: time-to-market, safety, low cost, reliability
- Specification - need for an unambiguous formalism to represent design choices and specifications
- Need for heterogeneity, unification for optimization and verification purposes

Basic Concepts

- A MOC is composed of:
  - Syntax: a description mechanism
  - Semantics: rules for computation of the behavior given syntax
    - Operational semantics
    - Denotational semantics
- It is chosen for its suitability:
  - Compactness of description,
  - Fidelity to design style,
  - Ability to synthesize and optimize the behavior to an implementation

Most MOCs

- permit distributed system description (with a collection of modules)
- Gives rules dictating how each module compute (function)
- Gives rules specifying how the modules communicate

- Function and communication may not be described completely separately

Modeling with MOCs (cont)

- MOCs are implemented by a language and its semantics (operational or denotational)
- MOCs describe:
  - the kinds of relations that are possible in a denotational semantics
  - how the abstract machine behaves in an operational semantics
  - how individual behavior is specified and composed
  - how hierarchy abstracts the composition
  - communication style

Modeling with MOCs (cont)

- Two levels of abstraction
  - High level: processes and their interaction using signals
    - denotational view
    - processes describe both functional and communication behavior
  - Lower level: general primitives for function and timing
    - operational view

MoC Primitives

- *Functions*
  - combination of Boolean functions
  - synchronous state machines
- *Communications*
  - queues,
  - buffers,
  - schedulers

Tagged-Signal Model (TSM)

- Formalism for describing aspects of MOCs for embedded system specification
- A semantic framework for comparing and studying and comparing of MOCs
- Very abstract: describing a particular model involves imposing further constraints that make more concrete
  - A high level abstraction model: Defines processes and their interaction using signals
  - Denotational view without any language

Signals, Events, Tags

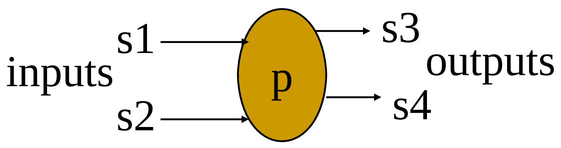
- Event: a value/tag pair -> the fundamental entity;
  - Tags: model time, precedence relationship, synchronization points, and etc
  - Values: represent the operands and results of computation.
  - Bottom ($\perp$): indicate the absence of a value
- Signal: a set of events (abstract aggregation)
  - Empty signal (λ),

TSM Processes

- Process P with n signals is a subset of the set of all n-tuples of signals S^n
- A signal $s \in S^n$ satisfies a process P if $s \in P$
  - *an* s *is also called a behavior of the process*
- A process is a set of possible behaviors or constraints on the set of legal signals.
- Processes in a systems operate concurrently and the constraints are on *communication or synchronization*
- The environment can be modeled as a process

TSM Processes

- Signals associated with a process may be divided as *input* and *output*
  - process does not determine its input
  - process does determine its output
- Process defines a relation between *input* and *output* signals
  - Ex. p = {s1,s2,s3,s4}



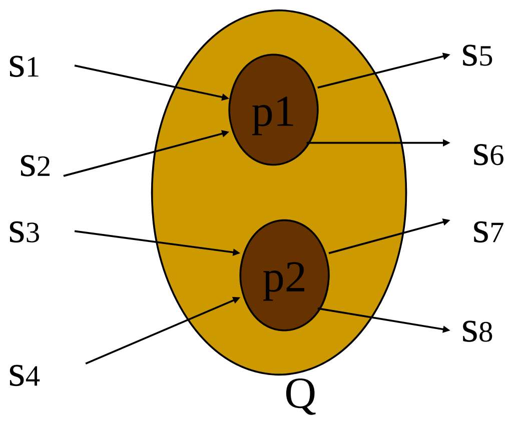
TSM Processes (cont)

- Functional process - given input signals, output is determined
  - injective
- Completely specified process: for all the inputs there is a unique behavior
  - bijective

TSM Process Composition

- Definition: Process composition in TSM is defined by the intersection of the constraints each process imposes on each signal
- Properties of process preserved by composition:
  - Functionality: unique output n-tuples for every input n-tuple
  - Complete specification: for every input n-tuple, there exists a unique output n-tuple

TSM Process composition



Q = {s1, s2, s3, …,s8)

TSM Process Composition

- Given a formal model of functional specification and of the properties, three situations may arise:
  - property is *inherent* for model of specification
  - property can be verified *syntactically* for given specification
  - property must be verified *semantically*, for given specification

Functional property examples

- Any design described by Dataflow Network is functional and hence this property need not be checked for this MoC. (Inherent)
- If above design is in FSM, even if the components are functional and completely specified, the result of composition may be either incompletely specified or nonfunctional.
  - This is due to feed-back loop in the composition
  - A syntactical check can find the feed-back paths
- With Petrinets, functionality is difficult to prove.
  - Exhaustive simulation required for checking functionality

Comparing MoCs

- System behavior
  - functional behavior and *communication behavior*, each as of TSM processes
- Process
  - functional behavior and *timing behavior*
- Function ⇒ how inputs are used for computing output
- Time ⇒ the order in which things happen (assignment of Tags to each event)

- Distinction between Function and Time is not clear in every context as in FSM

Concurrency and Communication

- ES has several coordinated *concurrent* processes with *communication* among them
- Communication can be:
  - explicit – sender forces an order on the events (sender and receiver processes)
  - implicit - sharing of tags (common time scale, or clock), which forces a common partial order of the events
- Time - a larger role in the embedded systems
  - two events are synchronous if they have the same tag

Communication primitives

- Unsynchronized: no coordination, no guarantee of valid read or not overwrite
- Read-modify-write: locks data structure during data access (In TSM, events are totally ordered, R-M-W action is one event)
- Unbounded FIFO buffered: point-to-point, produced token is consumed only after generated. (TSM context: simple connection where signal is constrained to have totally ordered. If consumer process has unbounded FIFOs on all inputs, all inputs have a total order imposed upon them.

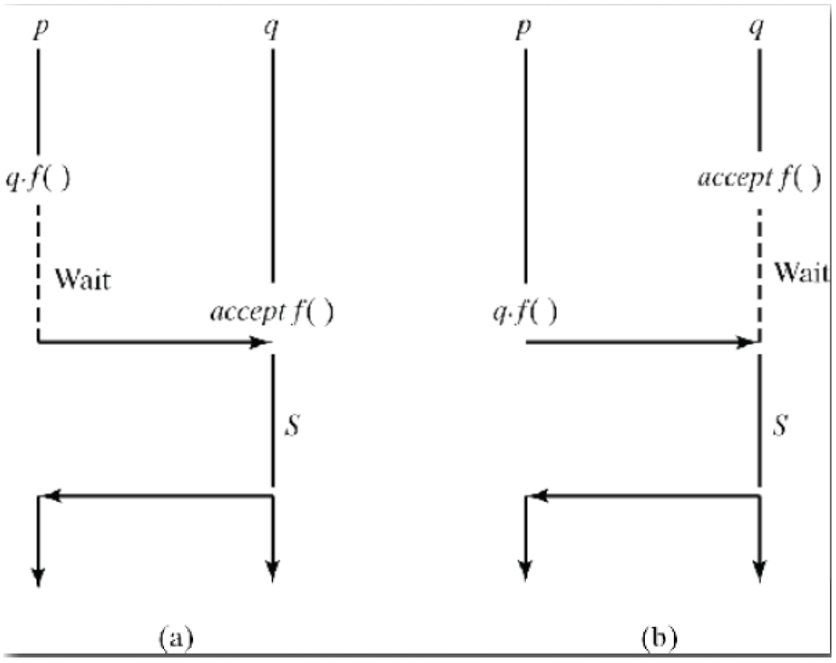
Communication primitives

- Bounded FIFO buffered: Each input and output signals are internally totally ordered. For buffer size = 1, input and output events must interleaved. For larger size, impose the maximum difference between input or output events occuring in succession.
- Petri net places
- Randezvous

Randezvous

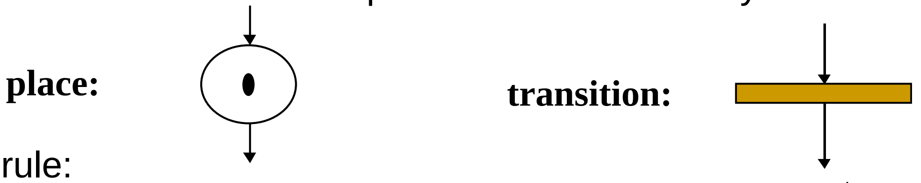
- Called process P is
  - part of an arbitrary process
  - Maintains state between calls
  - P may accet/delay/reject call
- Set up is symmetrical
  - Any process may be a client or a server.
- Caller: q.f(param)
  - Similar syntax/semantics to RPC
  - Q is the called process (server)
- Server: accept f(param) S
  - Must indicate willingness to accept
- Rendezvous
  - Caller (calling process) or server (called process) waits for the other, then they execute in parallel

Randezvous



Petri Net

- A PN is a mathematical formalism and a Graph tool to model and analyze discrete event dynamic systems. It is directed graphs with two types of nodes: places and transitions. Places represent states which may be 'held' and transitions represent events that may 'occur'

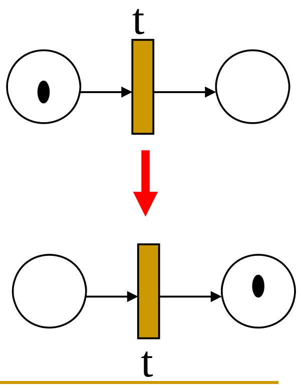


- Enabling rule:

  A transition t is enabled if and only if all the input places of the transition t have a token.

- Firing rule:

  An enabled transition t may fire at marking Mc. Firing a transition t will remove a token from each of its input places and will add a token to each of its output places.



Time and tags

- Different models of time -> different order relations on the set of tags
- Ordering relation
  - Reflexive
  - Transitive
  - Anti-symmetric
- If the ordering is partial, then T is called a partially ordered set or poset.
- Timed systems
  - Tags are totally ordered
- Untimed systems
  - Tags are partially ordered rather than totally ordered.

Basic Time

- ES are usually real-time systems
- Two *events* are synchronous if they have the same *tag*.
- Two *signals* are synchronous if each *event* in one *signal* is synchronous with an *event* in other *signal* and vice versa

Causality

- A casual process has a non-negative time delay from inputs to outputs.
- Strictly casual process has a positive time delay from inputs to outputs.

Treatment of Time

- Discrete-event system: a timed system where tags in each signal are *order isomorphic* with the natural numbers (Verilog, VHDL)
- Synchronous system: every signal in system is synchronous with every other signal in the system
- Discrete-time system: a synchronous discrete-event system
- Asynchronous system: no two events can have the same tag
  - asynchronous interleaved - tags are totally ordered
  - asynchronous concurrent - tags are partially ordered

Discrete-Event MOC

- Global event queue, totally ordered time
- Verilog, VHDL languages
- Simultaneous events present a challenge for discrete-event MOCs

t

MoC for reactive systems

- Main MoCs:
  - Finite State Machines (FSM)
  - Data Flow Process Networks
  - Petri Nets
  - Discrete Event
  - Codesign Finite State Machines
- Some main Languages:
  - Esterel, StateCharts, Dataflow Networks

Synchronous/Reactive

- Synchronous
  - All events are synchronous (all signals have identical tags)
  - Tags are totally ordered and globally available
  - All signals have events at all clock ticks (unlike discrete event model)
  - At each cycle the order of event processing may be determined by data precedences
  - Inefficient for systems where events do not occur at the same rate in all signals

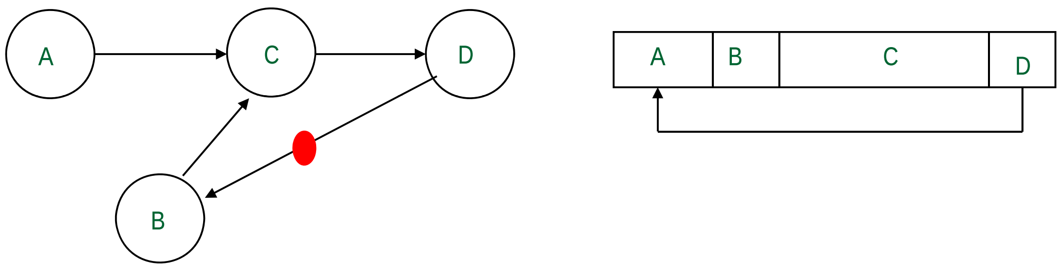
Synchronous/Reactive (cont)

- Synchronous/Reactive
  - set of concurrently-executing synchronized modules
  - modules communicate through signals which are either present or absent in each clock tick
- Computation is delay-free, arbitrary interconnection of processes is possible
- Verifying causality (non-contradictory and deterministic) is a fundamental challenge (a program may have no or multiple interpretations)
- Can be translated into finite state descriptions or compiled directly in hardware

Dataflow Process Networks

- Directed graph where the nodes (*actors*) represent computations and arcs represent totally ordered sequences of events (*streams*)
- Nodes can be language primitives specified in another language (C)
- A process can use partial information about its input streams to produce partial information at output -> causality without time
- Each process is decomposed into an indivisible sequence of firings, each firing consumes and produces tokens

Dataflow Process Networks (cont)



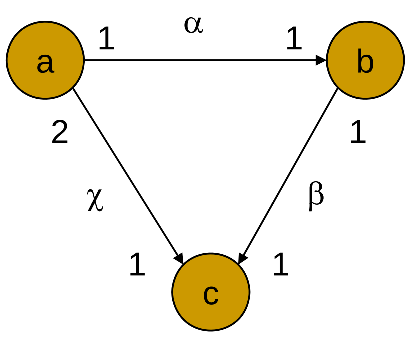
- A cycle in the schedule returns the graph in the original state
- Synchronous dataflow: processes consume and produce a finite number of tokens for each firing
- Tagged token model: partial order of tokens is explicitly carried in them

Synchronous data flow scheduling

- Determine schedule for data flow network (PAPS):
  - Periodic.
  - Admissible---blocks run only when data is available, finite buffers.
  - Parallel.
- Sequential schedule is PAPS.

Describing the SDF network (Lee/Messerschmitt)

- Topology matrix Γ.
  - Rows are edges.
  - Columns are nodes.

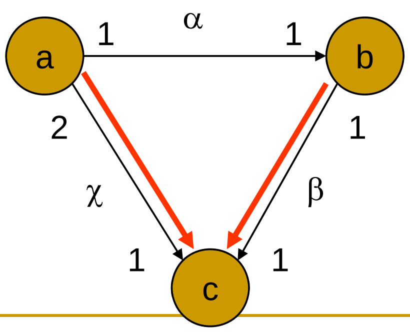


$$
\begin{array}{c|ccc}
 & a & b & c \\
\hline
\alpha & 1 & -1 & 0 \\
\beta & 0 & 1 & -1 \\
\chi & 2 & 0 & -1
\end{array}
$$

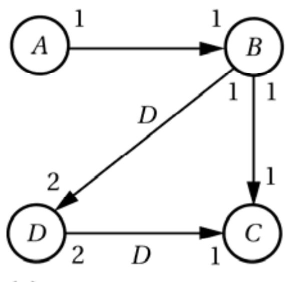
Feasibility tests

- Necessary condition for PASS schedule:
  - rank(Γ) = s-1 (s = number of blocks in graph).
- Rank of example is 3: no feasible schedule.

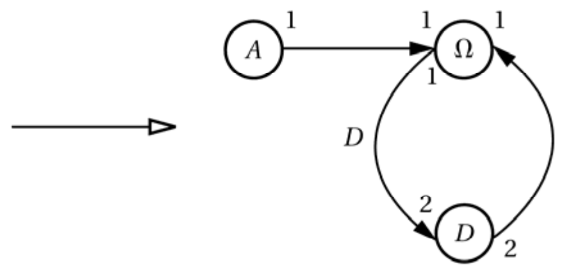
$$
\begin{array}{c|ccc}
 & a & b & c \\
\hline
\alpha & 1 & -1 & 0 \\
\beta & 0 & 1 & -1 \\
\chi & 2 & 0 & -1
\end{array}
$$



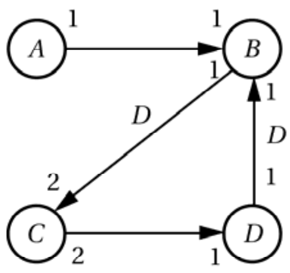
Clustering and single-appearance schedules



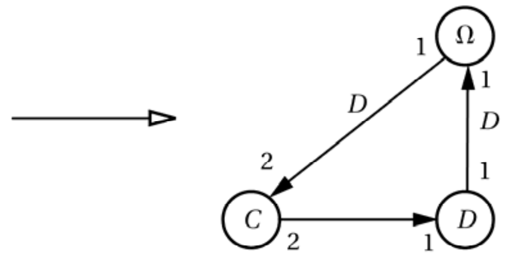
(a) 2(ab)d(2c)



(b) No single-appearance schedule



(c) 2(ab)c(2d)

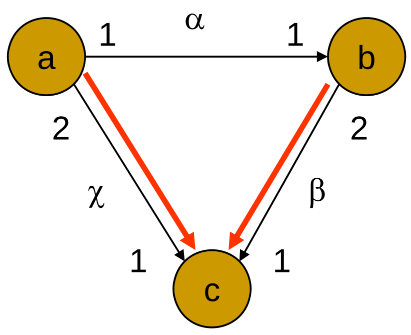


abc(2d)ab

(d) No single-appearance schedule

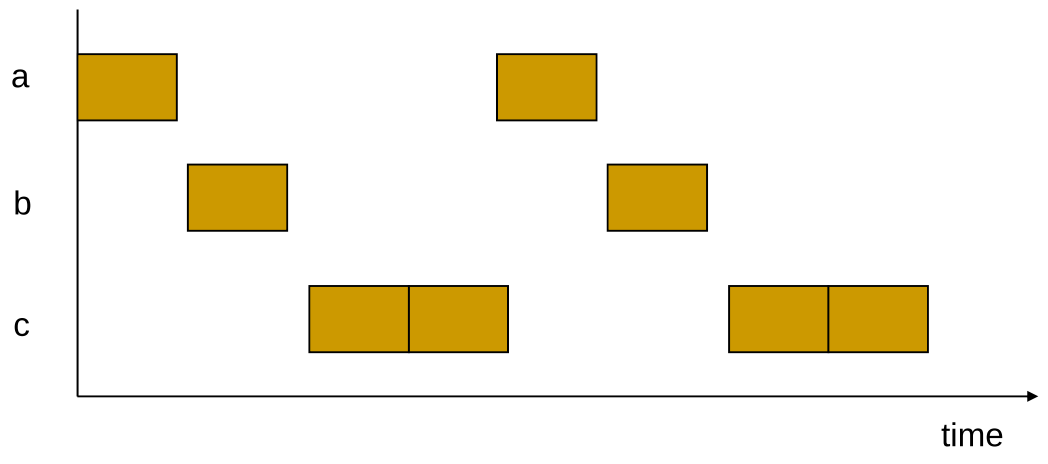
Fixing the problem

- New graph has rank 2.



	a	b	c
α	1	-1	0
β	0	2	-1
χ	2	0	-1

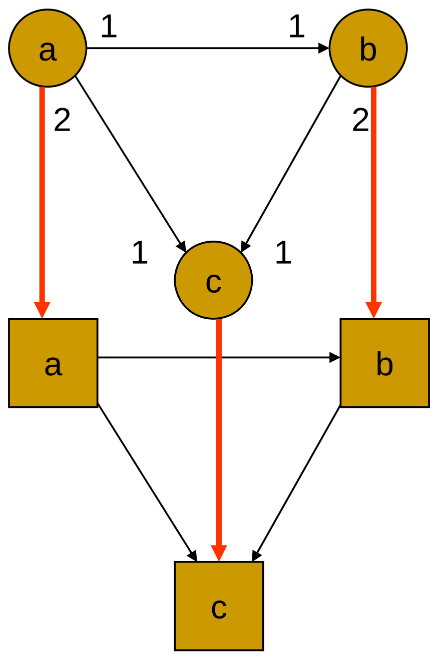
Serial system schedule



Schedule=ab(2c)

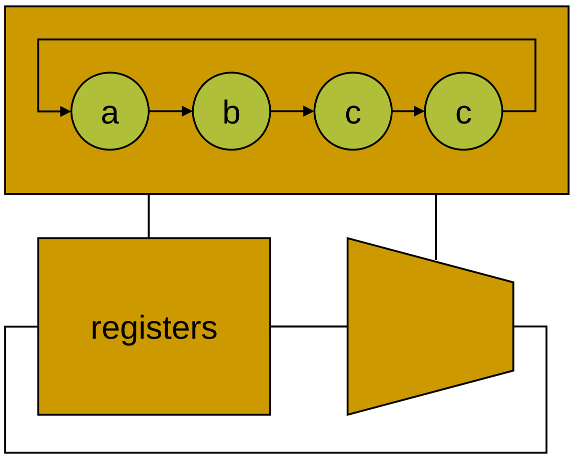
Allocating data flow graphs

- If we assume that function units operate at roughly the same rate as SDF graph, then allocation is 1:1.
- Higher data rates might allow multiplexing one function unit over several operators.



Fully sequential implementation

- Data path + sequencer perform operations in total ordering:



SDF scheduling

- Write schedules as strings: a(2bc)bc = abcbc.
- Lee and Messerschmitt: periodic admissible sequential schedule (PASS) is one type of schedule that can be guaranteed to be implementable.
  - Buffers are bounded.
- Necessary condition for PASS is that, given s blocks in graph, rank of topology matrix must be s-1.

Bhattacharyya et al. SDF scheduling algorithms

- One subgraph is subindependent of another if no samples from the second subgraph are consumed by the first one in the same schedule period in which they are produced.
- Loosely interdependent graph can be partitioned into two subgraphs, one subindependent of other.
- Single-appearance schedule has each SDF node only once.
  - We can use the recursive property of a single appearance schedule to schedule an SDF.