Models of Computation

1. **Models of Computation: What's that?**
2. **Synchronous FSMs and GALS models**
3. **Dataflow Models**

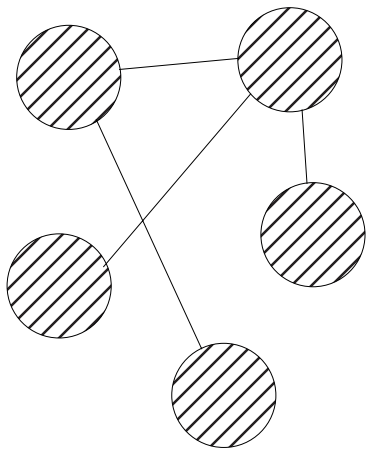
Models of Computation

☞ The *model of computation* deals with the set of theoretical choices that build the execution model of the language.

- A design is represented as a set of components, which can be considered as isolated monolithic modules (often called *processes* or *tasks*), interacting with each other and with the environment.

  The *model of computation* defines the behavior and interaction mechanisms of these modules.

Models of Computation (cont'd)



- Models of computation usually refer to:
  - how each module (process or task) performs internal computation
  - how the modules transfer information between them
  - how they relate in terms of concurrency
- Some models of computation do not refer to aspects related to the internal computation of the modules, but only to module interaction and concurrency.

Models of Computation (cont'd)

The main aspects we are interested in:

- Concurrency
- Communication&Synchronization
- Time
- Hierarchy

Common Models of Computation

Some of the models of computation commonly used to describe embedded systems:

- **(Synchronous) Finite State Machines**
- **GALS Models**
- **Dataflow Models**
- Petri Nets
- Discrete Event
- Timed Automata

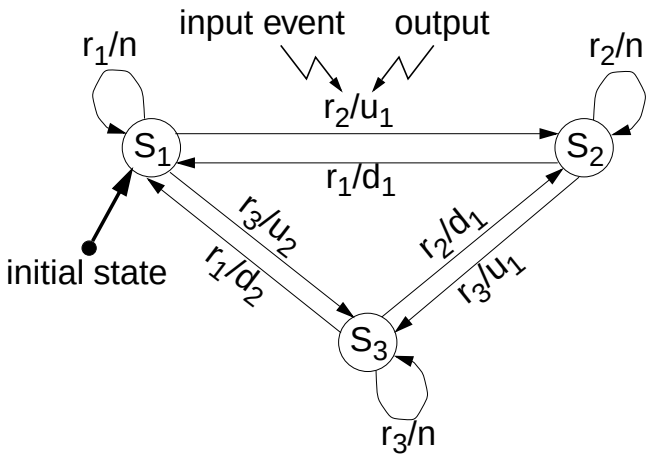
Finite State Machines

- The system is characterised by *explicitly* depicting its states as well as the transitions from one state to another.
- One particular state is specified as the initial one
- States and transitions are in a finite number.
- Transitions are triggered by input events.
- Transitions generate outputs.
- FSMs are suitable for modeling control dominated reactive systems (react on inputs with specific outputs; not much computation)

FSM Example

Elevator controller

- Input events {r_1, r_2, r_3}
  - r_i: request from floor i.
- Outputs {d_2, d_1, n, u_1, u_2}
  - d_i: go down *i* floors
  - u_i: go up *i* floors
  - *n*: stay idle
- States {S_1, S_2, S_3}
  - S_i: elevator is at floor *i*.



State Explosion

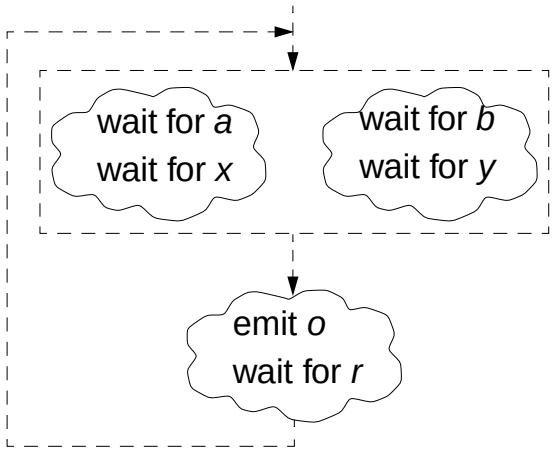
- Complex systems tend to have very large number of states. This particularly is the case in the presence of concurrency.
  The fenomen is called *state explosion*.
- Every global state of a concurrent system must be represented individually ⇒ interleaving of independent actions leads to an exponential number of states.
- Expressing such a system as a FSM (or extended FSM) is very difficult.

State Explosion (cont'd)

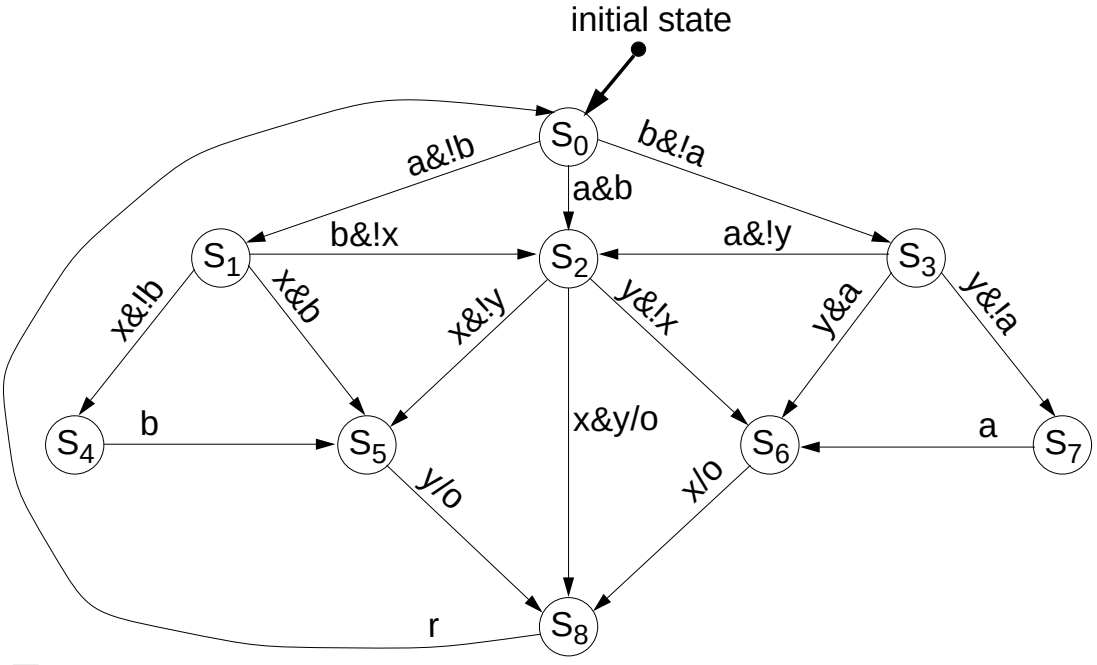
Example

After starting the system, it waits simultaneously for event *a* followed by *x*, and event *b* followed by *y*. Events can arrive in any order, except that *x* follows *a* and *y* follows *b*. Once the events are received, output *o* is emitted. Then the system waits for the reset signal *r* to return into the initial state.

- Input events {a, b, x, y, r}
- Output {o}
- States {S_0, S_1, ..., S_8}

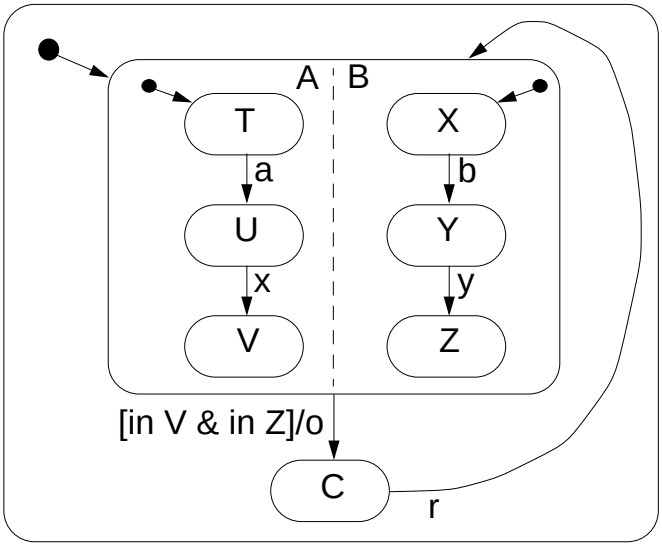


State Explosion (cont'd)



Hierarchical Concurrent Finite State Machines

The same example using concurrent FSMs (in Statecharts):

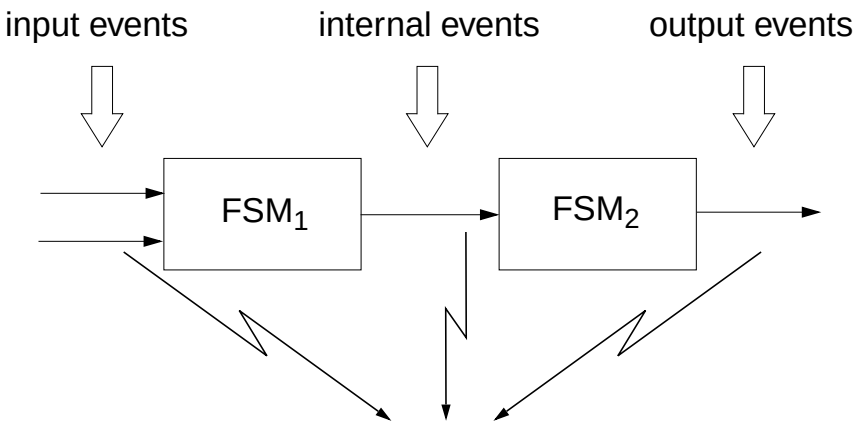


FSMs: Time and Synchrony

☞ FSMs are *synchronous models*.

- The synchrony hypothesis:
  The time is a sequence of instants (clock ticks) between which nothing interesting occurs. In each instant, some events (inputs) occur in the environment, and a reaction (output) is computed *instantly* by the modelled design.
  - Computation and internal communication take no time.
  - Events are either simultaneous (occur at the same clock tick) or one strictly precedes the other (as opposed to dataflow and Petri Nets where we only have a partial order of events).

FSMs: Time and Synchrony (cont'd)



synchronized: input events are at the same time with the internal and output events generated as response.

FSMs: Time and Synchrony (cont'd)

Question

Is the synchronous model sufficiently realistic to be used in practice?

Answer

For some applications yes!

It is the case when the following assumption is true:

> *The reaction time of the system (including internal communication) is neglectable compared to the rate of external events*.

Why Do We Like Synchronous Models?

- A set of communicating, concurrent FSMs behaves exactly like one equivalent FSM.

Models are deterministic.

It is possible to formally reason about models and to formally check certain properties of the model. This is important for certain class of applications (e.g. safety critical systems)

- It is possible to efficiently synthesise (compile) synchronous models to hardware or software.

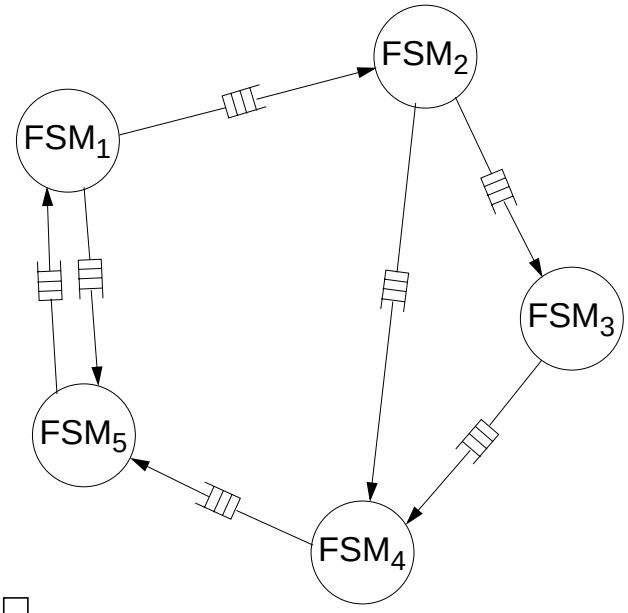
Why Do We Not Like Synchronous Models?

- Reasoning, verification and synthesis based on synchronous models are meaningful and correct only as long as:
  1. A completely synchronous implementation of the whole system is possible (the whole system acts similar to one single FSM).
  2. We are sure that for the *implemented system* the assumption on slide 14 is true.

- Implementing *large* models as synchronous systems is expensive and often technically impossible.

Globally Asynchronous Locally Synchronous Systems

Globally asynchronous and locally synchronous (GALS) models:



- Each FSM individually behaves like a synchronous systems ⇒ reacts instantaneously on a set of available inputs and generates output.
- The global system is asynchronous ⇒ communication time is finite and non-zero; reaction time of each FSM, as viewed by other FSMs is finite and non-zero.
- With global asynchrony, buffering of signals could be needed.

Globally Asynchronous Locally Synchronous Systems (cont'd)

☞ With a GALS model, the set of FSMs is not any more equivalent with a single FSM (as was the case for the synchronous model).

Several nice features are gone:

- With synchronous FSMs we had the nice theoretical background and the possibility of formal verification of the whole system. *Not the case with GALS*.
- Every implementation of a synchronous FSM model is guaranteed to be functionally equivalent to the initial model and behave exactly and deterministically like the model (in the case we are able to produce an implementation!). *Not the case with GALS*.

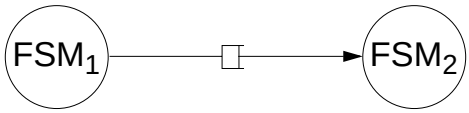
Globally Asynchronous Locally Synchronous Systems (cont'd)

☞ The GALS model is not deterministic, in the sense that its behavior depends on the amount of time taken for a certain communication or transition.

⇩

Two different implementations of the same GALS model can behave differently.

Globally Asynchronous Locally Synchronous Systems (cont'd)



- A GALS model in which FSM_1 and FSM_2 communicate through a single-slot buffer.
- FSM_1 outputs a signal (writes into the buffer) every 2 ms (we neglect communication time).
  1. If the reaction time of FSM_2 is 6ms, every third signal from FSM_1 will be reacted on.
  2. If we have a faster implementation of FSM_2, with reaction time 2ms, every signal from FSM_1 will be captured.

Globally Asynchronous Locally Synchronous Systems (cont'd)

- Each individual FSM can be still verified and even formal methods can be used.
- However, individual correctness of each FSM does not guarantee the correctness of the whole system. The system behaves correctly only if, in addition, certain assumptions regarding the timing of components and of communications are satisfied.

Example on previous slide:

- Each FSM can be functionally verified individually.
- The global system will be correct (no signal is lost) if FSM_2 has a reaction time which is smaller than the production rate of FSM_1.
- Estimation and simulation can be used in order to verify that a certain implementation (like FSM_1 as software on a certain μprocessor, and FSM_2 as an ASIC) satisfies this assumption.

GALS System Models

☞ A GALS system is modelled as a *network of FSMs*:

- Each FSM has a *locally synchronous* behavior: it executes a transition by producing a single output reaction based on a single, snap-shot input assignment in zero time.

- A System has a globally asynchronous behavior: each FSM reads inputs, executes a transition, and produces outputs in a finite amount of time as seen by the rest of the system.

Dataflow Models

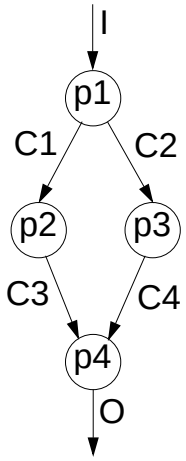
- Systems are specified as directed graphs where:
  - *nodes* represent computations (processes);
  - *arcs* represent totally ordered sequences (streams) of data (tokens).
- Depending on their particular semantics, several models of computation based on dataflow have been defined:
  - Kahn process networks
  - Dataflow process networks
  - Synchronous dataflow
  - - - - - - -

Dataflow Models (cont'd)

- Dataflow models are suitable for signal-processing algorithms.
  - Code/decode, filter, compression, etc.
  - Streams of periodic and regular data samples
  - Typically signal-processing algorithms are expressed as block diagrams; this naturally fits to dataflow semantics.

Dataflow Models (cont'd)

```
Process p1( in int a, out int x, out int y) {
……
}

Process p2( in int a, out int x) {
……
}

Process p3( in int a, out int x) {
……
}

Process p4( in int a, in int b, out int x) {
……
}

channel int I, O, C1, C2, C3, C4;

p1(I, C1, C2);
p2(C1, C3);
p3(C2, C4);
p4(C3, C4, O);
```



☞ The internal computation of a process can be specified in any programming language (e.g. C).

This is called the *host language*.

Kahn Process Networks

- Processes communicate by passing data tokens through unidirectional FIFO channels.
- Writes to the channel are non-blocking.
- Reads are blocking:
  - the process is blocked until there is sufficient data in the channel

} Non-blocking communication

⇩

A process that tries to read from an empty channel waits until data is available. It cannot ask whether data is available *before* reading and, for example, if there is no data, decide not to read that channel.

⇨ **DETERMINISM**

Kahn Process Networks (cont'd)

☞ Kahn process networks are deterministic:

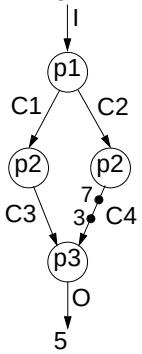
- For a certain sequence of inputs, there is only one possible sequence of outputs (regardless, for example, how long time it takes for a certain computation or communication to finish).

  Looking only at the specification (and not knowing anything about implementation) you can exactly derive the output sequence corresponding to a certain input sequence.

Kahn Process Networks (cont'd)

```
Process p1( in int a, out int x, out int y) {
int k;
loop
      k = a.receive();
      if k mod 2 = 0 then
            x.send(k);
      else
            y.send(k);
      end if;
end loop; }

Process p2( in int a, out int x) {
int k;
loop
      k = a.receive();
      x.send(k);
end loop; }

Process p3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
      if sw then
            k = a.receive();
      else
            k = b.receive();
      end if;
      x.send(k);
      sw = !sw;
end loop; }

channel int I, O, C1, C2, C3, C4;
p1(I, C1, C2);
p2(C1, C3);
p2(C2, C4);
p3(C3, C4, O);
```

7
3
0
16
8
5
11
21
8
5



5
0
11
16
21
8
5
8

```
Process q3(in int a, in int b, out int x) {
int k; bool sw = true;
loop
      if sw then
            k = a.receive() on timeout(d) do
                select on a,b
                    a: k = a.receive();
                or
                    b: k = b.receive();
                end select;
      else
            k = b.receive() on timeout(d) do
                select on a,b
                    a: k = a.receive();
                or
                    b: k = b.receive();
            end select;
      end if;
      x.send(k);
      sw = !sw;
end loop; }
```

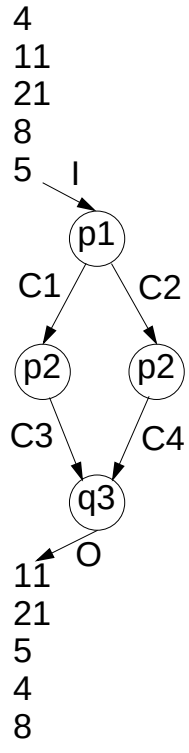
☞ Consider q3 instead of p3:

- Process q3 first tries channel *a* or *b*, depending on sw, like in the previous version.
- But**, instead of blocking**, if nothing comes after a timeout period *d*, q3 will read *f*rom *any* of the two channels, taking the token *which is available first*.

⇩

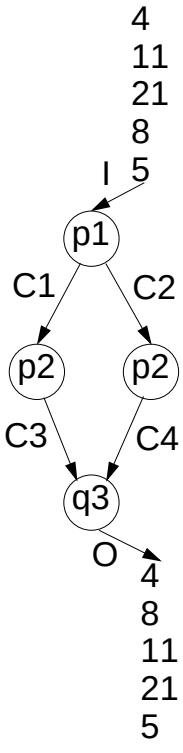
- With q3 we do not have a Kahn process network
- **The system is not deterministic**.

Kahn Process Networks (cont'd)



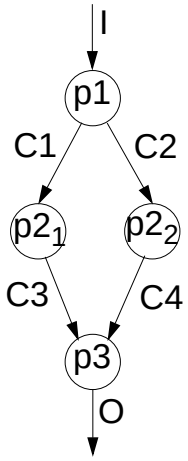
With an implementation such that channel C3 is very fast and C4 is very slow.

With an implementation such that channel C3 is very slow and C4 is very fast.



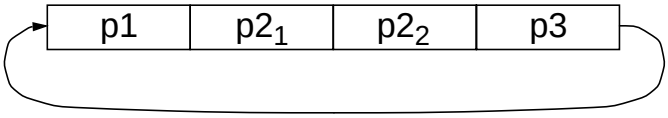
Kahn Process Networks (cont'd)

Let us come back to the Kahn process network (slide 28):



☞ Let us imagine we have to implement the system on a single processor architecture.

Let's try the following static schedule:

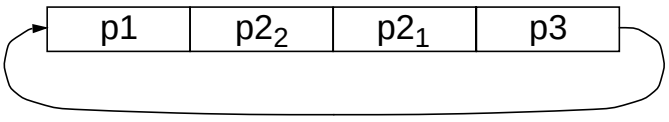


The system will block!
Example:

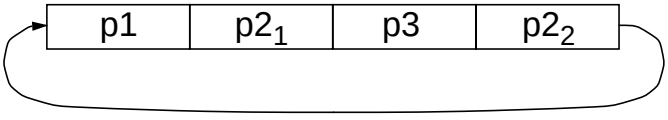
First token is 4; p1 passes to C1;
p2$_1$ passes to C3; p2$_2$ waits for ever.

Kahn Process Networks (cont'd)

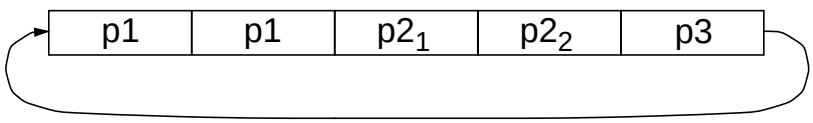
Let's see some other schedules:



The system will block!
Example: first token 4.



The system will block!
Example: first token 4.



The system will block!
Example: sequence starting with 4, 8.

Kahn Process Networks (cont'd)

- **Kahn process networks cannot be scheduled statically** ⇒ It is not possible to derive, at compile time, a sequence of process activations such that the system does not block under any circumstances.

⇩

Kahn process networks have to be scheduled dynamically ⇒ which process to activate at a certain moment has to be decided, during execution time, based on the current situation.

⇩

There is a (huge) overhead in implementing Kahn process networks.

Kahn Process Networks (cont'd)

- Another problem: memory overhead with buffers.
  Potentially, it is possible that the memory need for buffers grows unlimited (see channel C4 on slide 28).

- Kahn process networks are too general; they are strong in their expressive power but cannot be implemented efficiently.

⇩

Introduce limitations so that you can get efficient implementations.

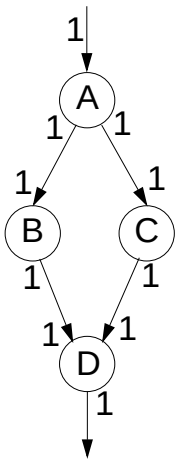
Synchronous dataflow

- *Dataflow process networks* are a particular case of Kahn process networks.

  A particular kind of dataflow process networks, which can be efficiently implemented, are *synchronous dataflow networks* (SDN).

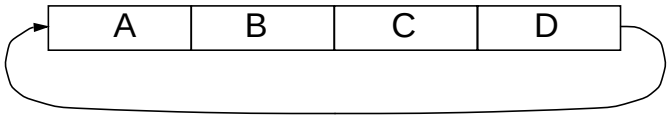
- *Synchronous dataflow networks* are Kahn process networks with additional restriction:
  - At each activation (firing) a process produces and consumes a fixed number of data tokens on each of its outgoing and incoming channels respectively.
  - For a process to fire, it must have at least as many tokens on its input channels as it has to consume.

Synchronous dataflow (cont'd)

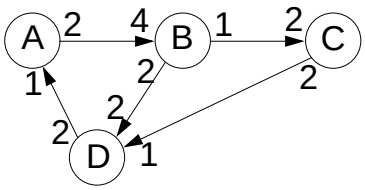


- Arcs are marked with the number of tokens produced or consumed.
- This is a simple "single-rate" system: every process is activated one single time before the system returns to its initial state.

Possible static schedule:



Synchronous dataflow (cont'd)



- For a correct synchronous dataflow network there exists a sequence of firings which returns the network in its original state.

  This sequence represents a static schedule which has to be repeated in a cycle.
- The schedule is such that a finite amount of memory is required (no infinite buffers)

Problem

How to derive such a cyclic schedule?

Deriving a static schedule for SDF

- Along the periodic sequence of firing, on each arc the same number of tokens has to be produced and consumed.

  a, b, c, d: the number of firings, during a period, for process A, B, C, D.

Balance equations:

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

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

Deriving a static schedule for SDF (cont'd)

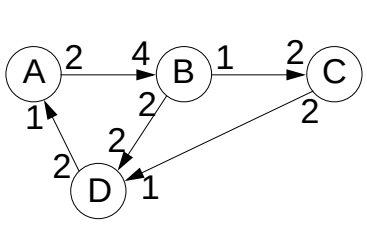
For a certain SDF graph we get an equation:

$$\Gamma \mathbf{q} = \mathbf{0}$$

- Γ: topology matrix of the graph
- $\mathbf{q}$: firing vector
- $\mathbf{0}$: vector of zeros

☞ If there is no $q \neq 0$ which satisfies the equation above $\Rightarrow$ there is no static schedule (there is a *rate inconsistency* between processes).

Deriving a static schedule for SDF (cont'd)

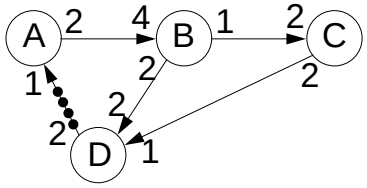


Among several possible solutions for vector *q*, we are interested in the smallest positive integer vector (smallest in the sense of the sum of the elements)

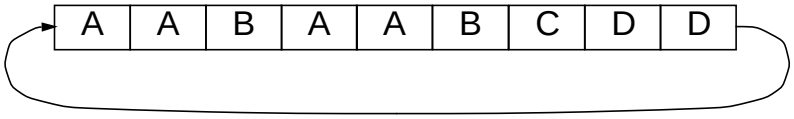
For our SDF graph, this solution is: a=4, b=2, c=1, d=2.

- The numbers above are telling us how often each task is activated during one period.
- Based on these numbers a periodic static schedule can be elaborated.

Deriving a static schedule for SDF (cont'd)



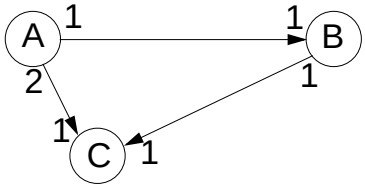
A possible schedule:



☞ The schedule is possible, without deadlock, only if 4 initial tokens are provided on the channel D $\rightarrow$ A.

Deriving a static schedule for SDF (cont'd)

With this example we have a rate inconsistency $\Rightarrow$ No static, periodic schedule with finite buffers is possible.



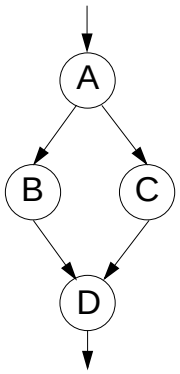
- There is no solution for the equation, different from a=b=c=0.
- It is easy to observe that on the arc A $\rightarrow$ C, tokens continuously accumulate.

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

Treatment of Time

☞ Dataflow systems are *asynchronous concurrent*.

- Events can happen at any time.
- There exists a a partial order of events:
  - Producing a token by A strictly precedes consuming a token by B.
  - There is no order between consuming a token by B and consuming a token by C.



Summary

- FSMs are the typical state based model. System states are explicitly depicted. As response to input events, an FSM reacts with a transition to a new state and also generates output events.
- FSMs are suitable for modeling control dominated reactive systems.
- Complex systems can have an extremely large number of states: state explosion. This is critical in the case of concurrent systems.
- FSMs are synchronous models. The whole system reacts instantaneously to a set of simultaneous input events. Events are either simultaneous or strictly ordered.
- FSM models are deterministic. Formal reasoning is possible, as well as efficient synthesis.

Summary (cont'd)

- Implementing synchronous systems can be done efficiently under certain circumstances. However, it is practically impossible for large systems and, in particular, distributed systems and, even more, hardware/software systems.
- For many systems and, in particular, larger distributed hardware/ software systems, only GALS is a realistic approach for implementation.
- Some of the nice features of synchronous FSMs are gone in this case. Formal reasoning about the global system is not possible any more.

Summary (cont'd)

- Dataflow models consist of nodes representing computation and arcs representing totally ordered sequences of data. They are particularly suitable for signal-processing applications and, in general, applications dealing with streams of periodic/regular data samples.
- Several models of computation based on dataflow have been defined. They represent different trade-offs between expressiveness, on the one side, and determinism or potential of efficient implementation, on the other side.
- Kahn Process Networks: FIFO channels and blocking read.
  They are deterministic: For a certain sequence of inputs, there is only one possible sequence of outputs.
  Kahn Process Networks, in general, cannot be scheduled statically.

Summary (cont'd)

- Synchronous Dataflow Networks introduce an additional restriction: at each activation a process produces and consumes a fixed number of tokens.
- For a correct Synchronous Dataflow Networks a static schedule can be derived.

☞ **Don't use the "strongest" modeling approach! Go for exactly that expressive power you need; not more.**