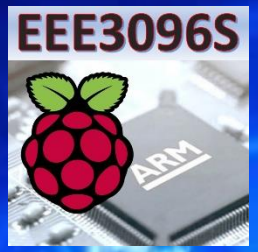


Specifications and Modeling (5)

Section 2.5 – Data Flow

L10

Embedded Systems II

Dr Simon Winberg

Electrical Engineering
University of Cape Town

Outline of Lecture

- Synchronous Data Flow (SDF)
- Homogeneous Synchronous Data Flow (HSDF)
- Parallel Scheduling of SDF Models
- Expressiveness of data flow MoCs
- ‘Analyzability burr’
- Actor Languages
- SDF Periodic Schedule
- SDF → C Code Example

Models of computation in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text ✓, use cases ✓ | Sequence Charts ✓, ICD		
Communicating finite state machines✓	StateCharts ✓		SDL
Data flow	Scoreboarding + Tomasulo Algorithm (☞ Comp.Archict.)		Kahn networks✓, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy (Ptolemy only discussed briefly)	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures
SystemC will not be delved into detail. Only brief flavour of VHDL and Verilog given

Synchronous Data Flow (SDF)

- Less computationally powerful, but easier to analyze:

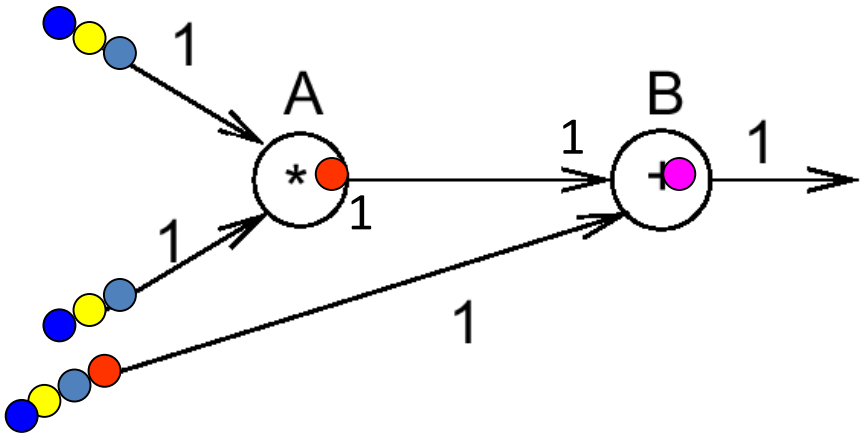
- **Synchronous data flow (SDF).**

  - **Synchronous**
    = global clock controlling “firing” of nodes

  - Again using ***a***synchronous message passing
    = tasks do not have to wait until output is accepted.

(Homogeneous-) Synchronous data flow (SDF)

- Nodes are called ***actors.***
- Actors are ***ready***, if the necessary number of input tokens exists and if enough buffer space at the output exists
- Ready actors ***can fire*** (be executed).

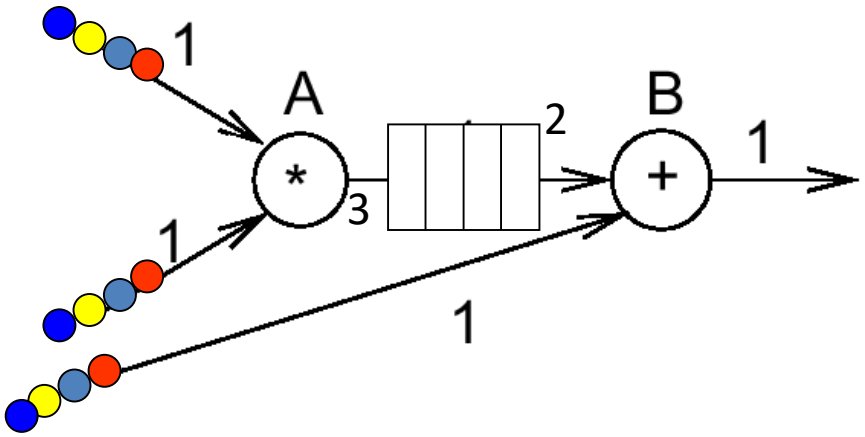


Execution takes a ***fixed, known time***.

Actually, this is a case of ***homogeneous*** synchronous data flow models (HSDF): # of tokens per firing the same. ☞

(Non-homogeneous-) Synchronous data flow (SDF) (1)

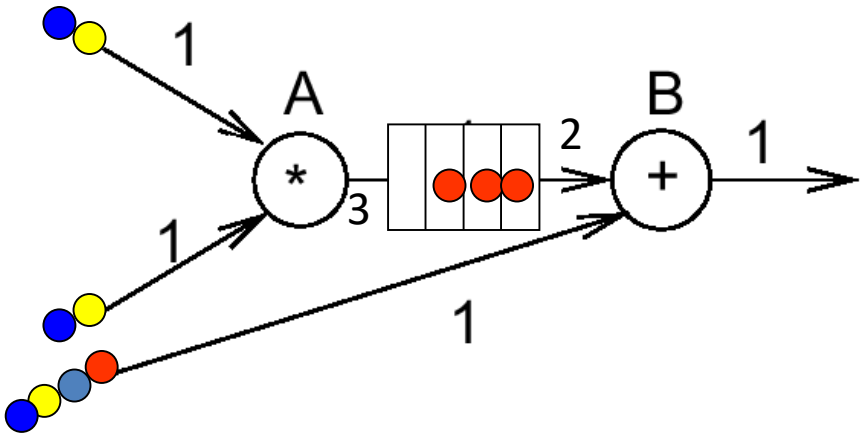
- In the general case, a number of tokens can be produced/ consumed per firing



A ready, can fire (does not have to)

(Non-homogeneous-) Synchronous data flow (SDF) (2)

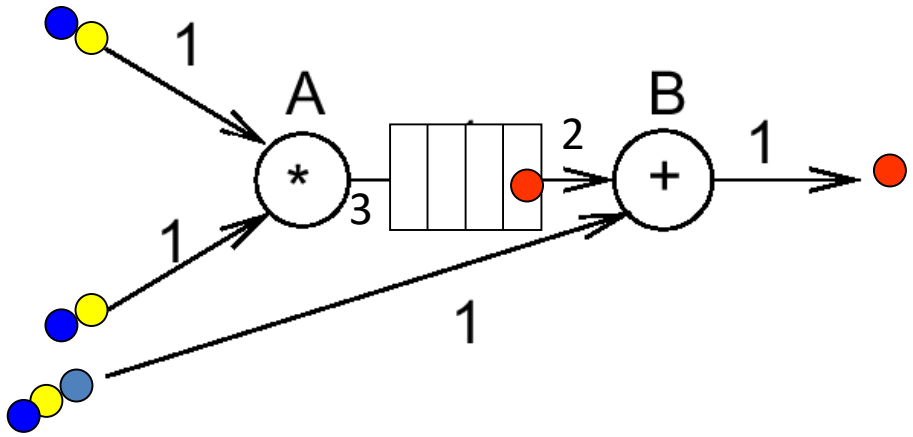
- In the general case, a number of tokens can be produced/ consumed per firing



B ready, can fire

(Non-homogeneous-) Synchronous data flow (SDF) (3)

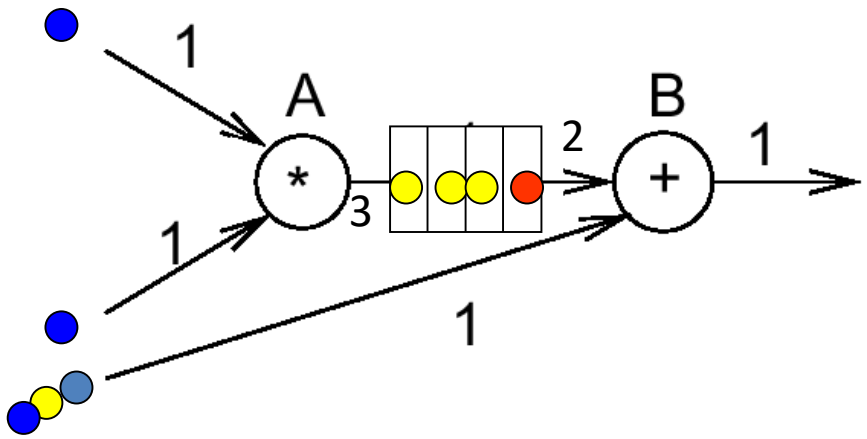
- In the general case, a number of tokens can be produced/ consumed per firing



A ready, can fire

(Non-homogeneous-) Synchronous data flow (SDF) (4)

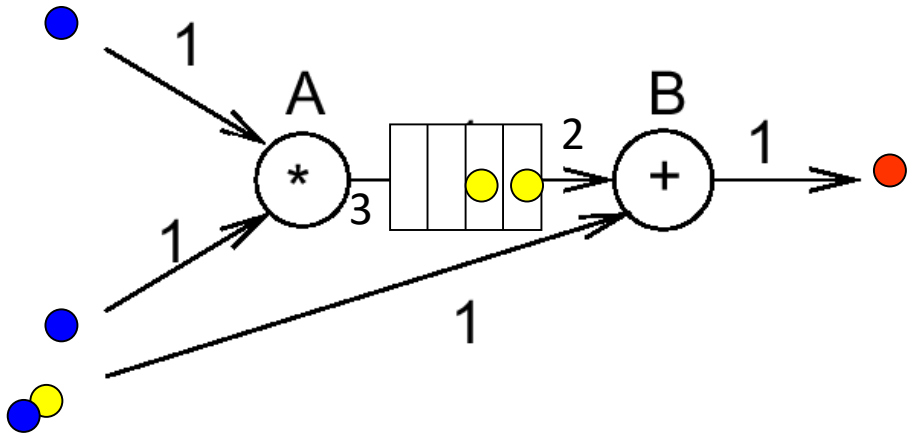
- In the general case, a number of tokens can be produced/ consumed per firing



B ready, can fire

(Non-homogeneous-) Synchronous data flow (SDF) (5)

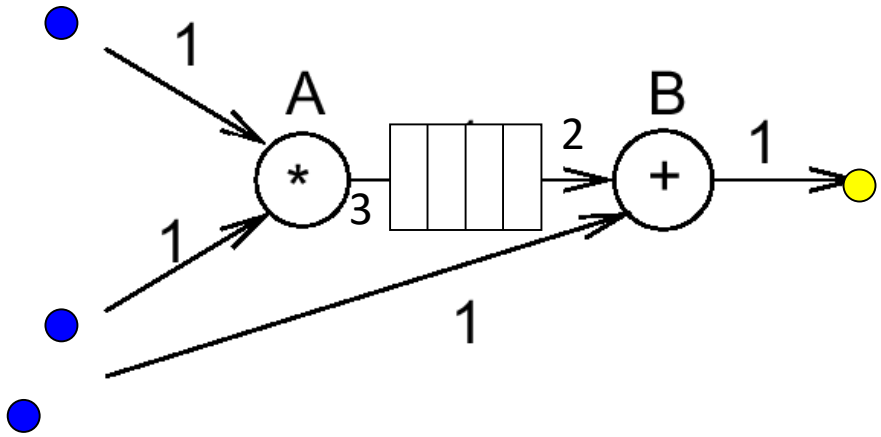
In the general case, a number of tokens can be produced/ consumed per firing



B ready, can fire

(Non-homogeneous-) Synchronous data flow (SDF) (6)

- In the general case, a number of tokens can be produced/ consumed per firing

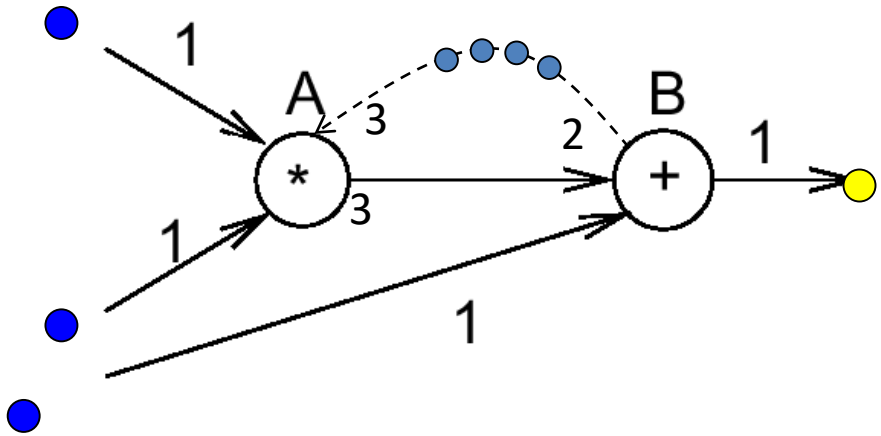


1 period complete, A ready, can fire

Replace FIFO by back-edge

Actual modeling of buffer capacity

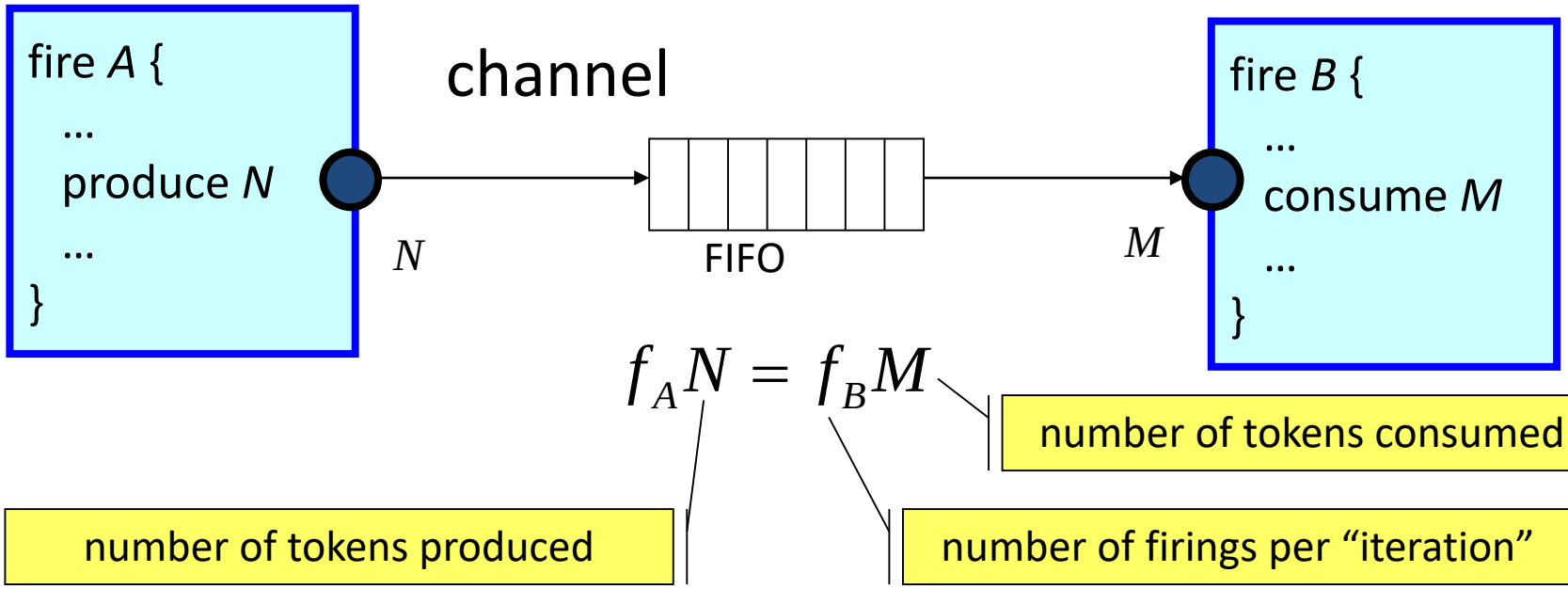
The capacity of buffers can be modeled easier: as a ***backward*** edge where (initial number of tokens = buffer capacity).



Firing rate depends on # of tokens …

Note there does not need to be tokens on the feedback look (dotted line) for A to file. The actor A may only consume these when available or as needed

Multi-rate models & balance equations (one for each channel)



Decidable:

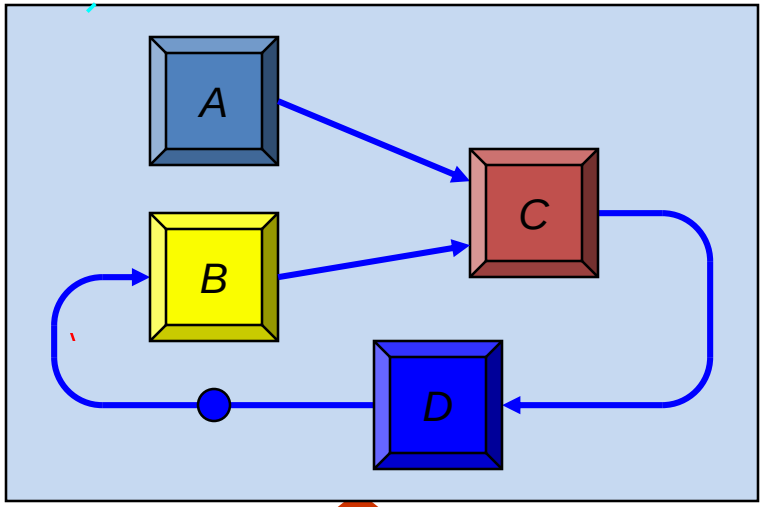
- buffer memory requirements
- deadlock

Schedulable statically

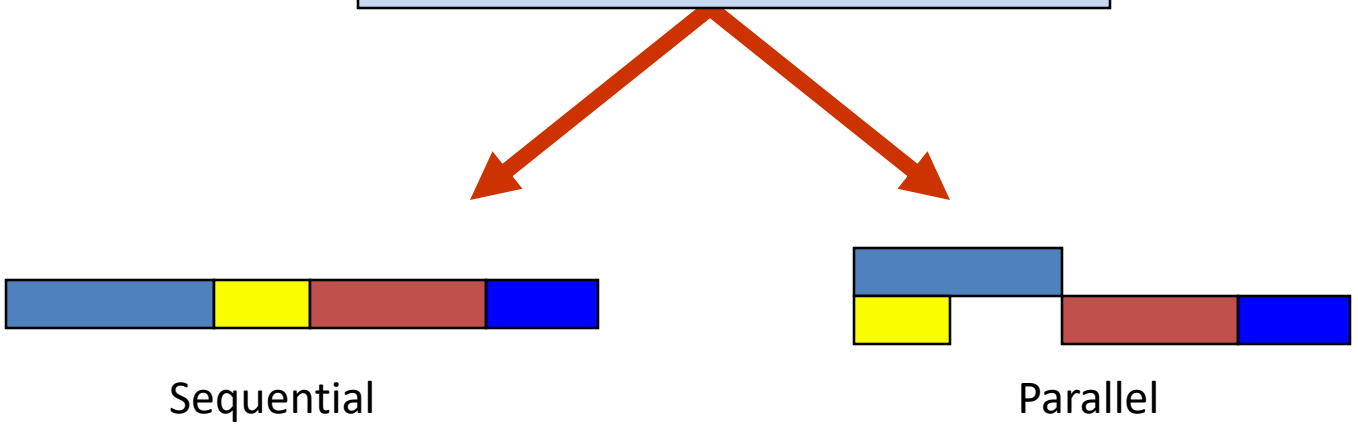
Adopted from: ptolemy.eecs.berkeley.edu/presentations/03/streamingEAL.ppt

Parallel Scheduling of SDF Models

SDF is suitable for automated mapping onto parallel processors and synthesis of parallel circuits.

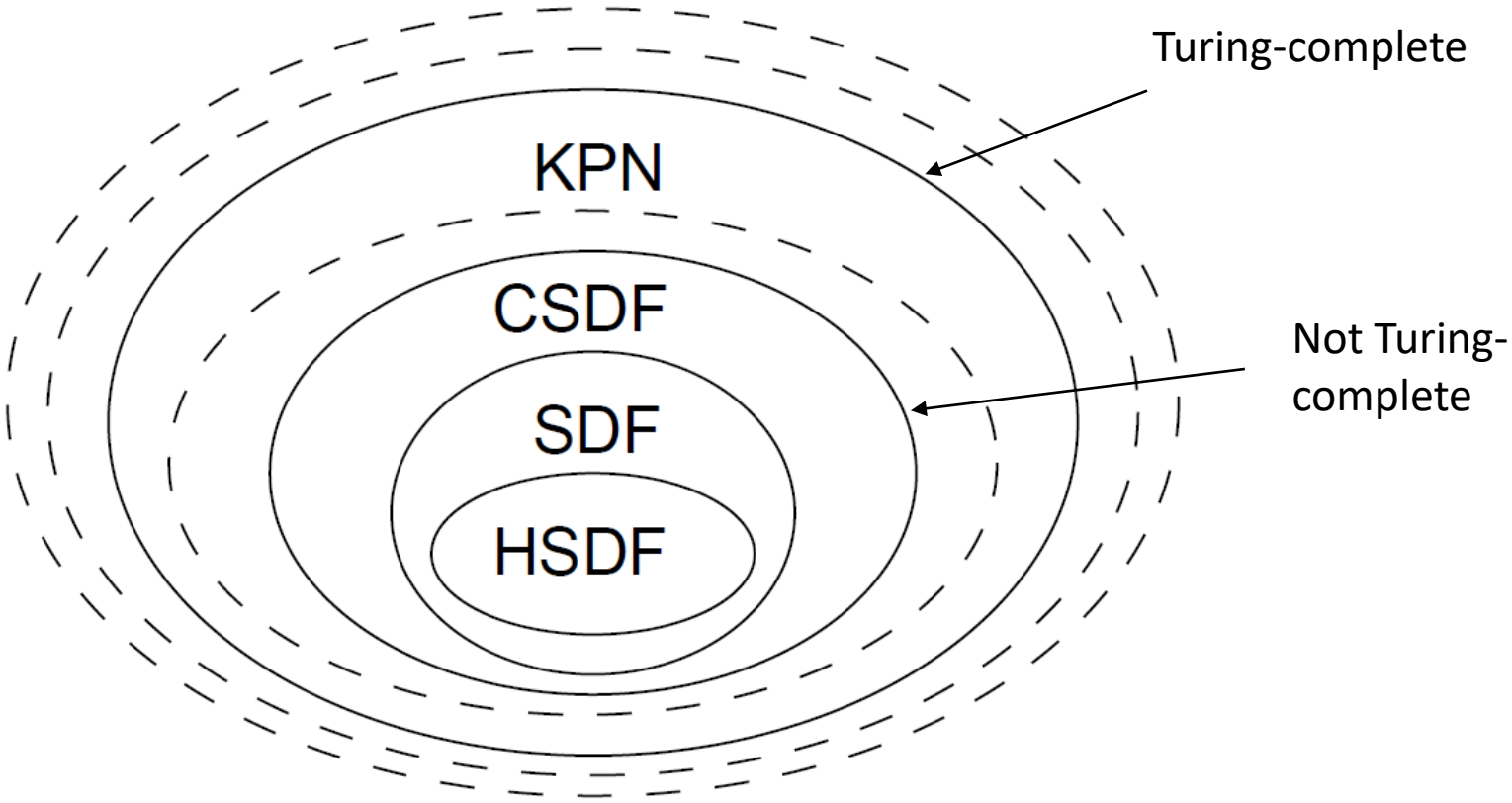


Many scheduling optimization problems can be formulated using these models.



Source: ptolemy.eecs.berkeley.edu/presentations/03/streamingEAL.ppt

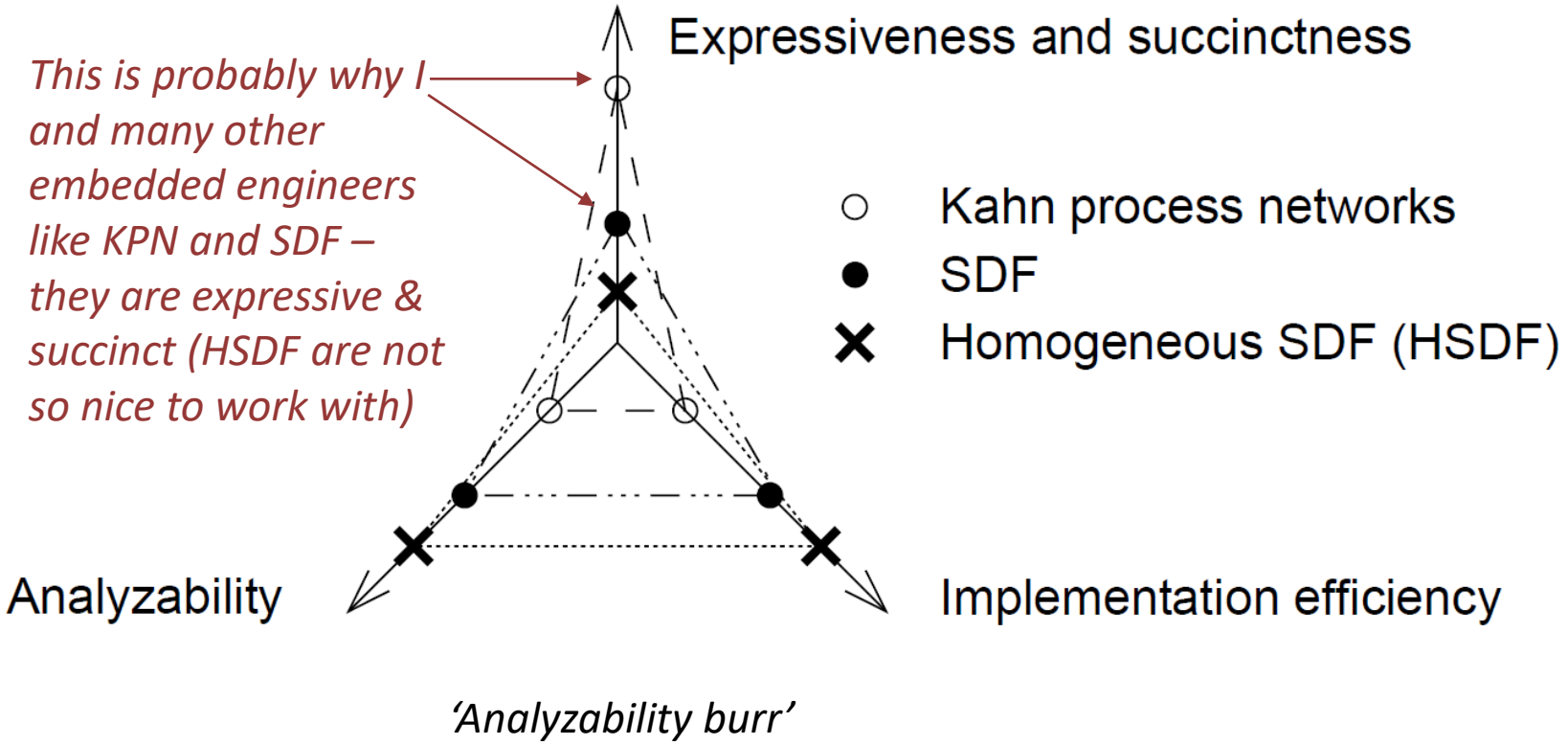
Expressiveness of data flow MoCs



CSDF=Cyclo static data flow (rates vary in a cyclic way)

[S. Stuijk, 2007]

The expressiveness / analyzability conflict



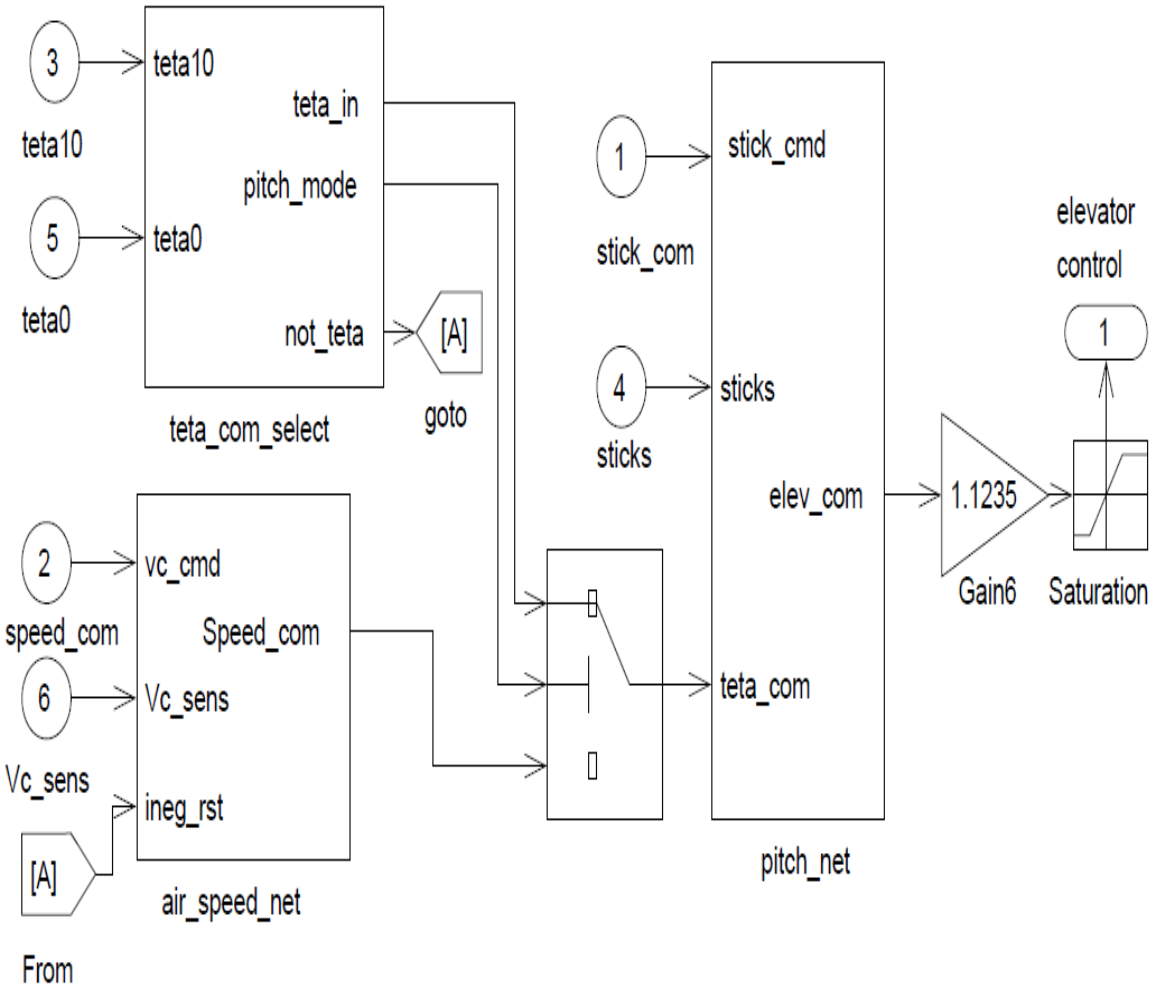
'Analyzability burr'

Expressiveness of a language *= the breadth of concepts that can be expressed in the langue. For example assembly is a weak, not very expressive language. It is hard to explain even slightly involved concepts (e.g. a loop) in aseembly. C is more expressive (e.g. has data types like structs) and C++ is a level beyond that in expressive power.*

[S. Stuijk, 2007]

Similar MoC: Simulink

- example -



Semantics? *"Simulink uses an idealized timing model for block execution and communication. Both happen infinitely fast at exact points in simulated time. Thereafter, simulated time is advanced by exact time steps. All values on edges are constant in between time steps."* [Nicolae Marian, Yue Ma]

[mathworks]

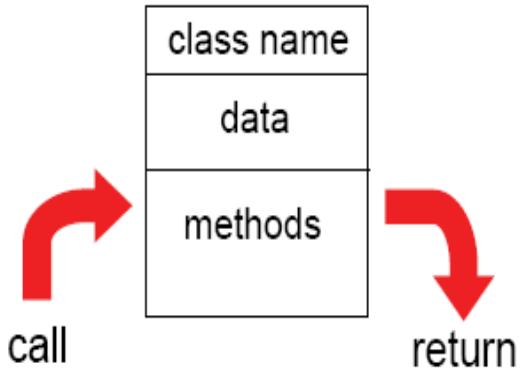
Not examined

Actor Languages

It's all very well describing (for SDF) the producing and consuming of tokens, but what processing is done by the actors? This is where the "actor language" comes in to describe the processing…

This provides ways to elaborate on the actors, generally they will have data and methods to apply (i.e. method(s) to consume and produce)

The established: Object-oriented:

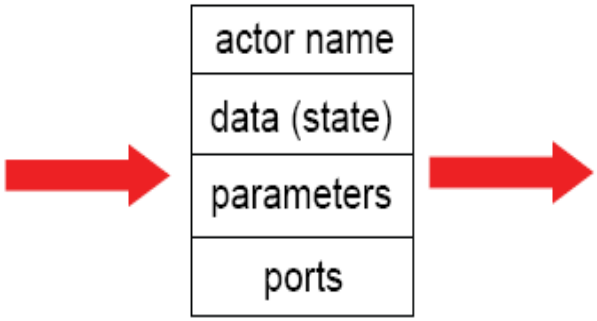


What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

Actors make things happen



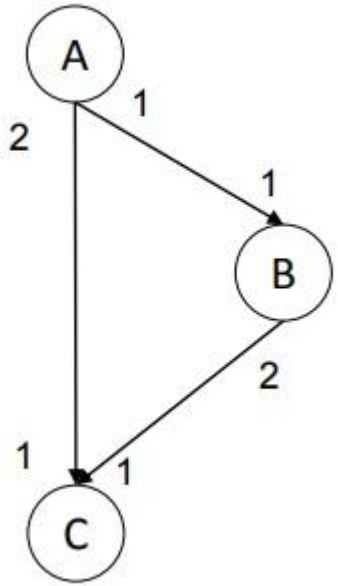
What flows through an object is streams of data

Example

Examples

- Describe the behaviour of this SDF model

… hint: Note that one doesn't usually indicate a starting actor, you may likely need to determine by working backwards to the node without dependencies.

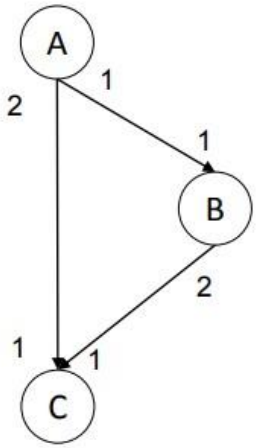


… Answer:

We see that C is the terminating actor for this model so we can work backwards from there. So accordingly the system described A will need to fire a single token to B (it only generates one token at a line on this edge), B will respond by firing two tokens to C. During this time A needs to fire two tokens to C. Once these tokens have been fired C will perform two iterations, each iteration consuming one token from the set fired by A and by B. (i.e. for each firing of A and B C will activate twice)

SDF Periodic Schedule

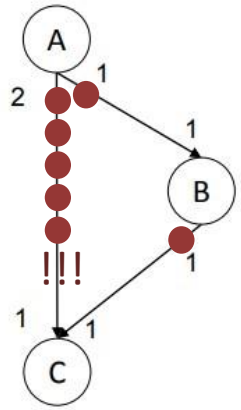
- The SDF model we saw previously can also be described in terms of its ‘periodic schedule’
- The ‘periodic schedule’ lists the pattern that nodes fire at, ie can be represented as follows:



Periodic schedule: ABCC

(or mode concisely): AB2C

Note schedules can be **a-periodic/chaotic** (not necessarily a bad thing) or **unbounded** (a bad thing, causes buffer overflows etc.)

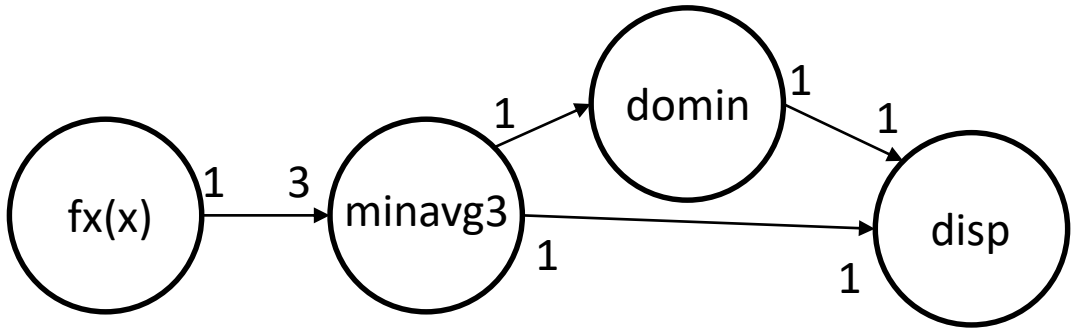


ABCABC…
(note A->C keeps filling)

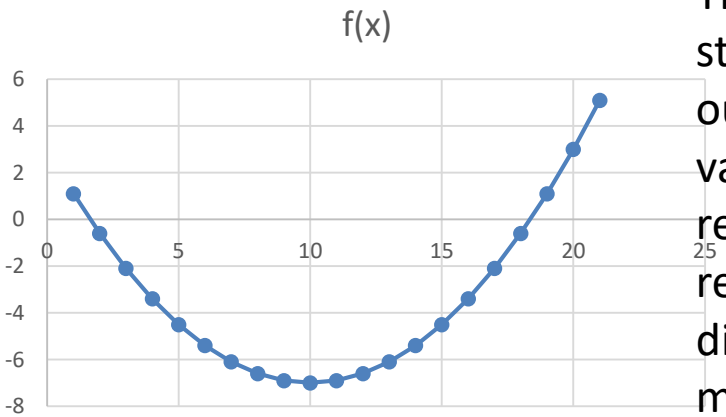
SDF
Coding

SDF → C Code Example

In order to implement a SDF model, you need both the model and a description of the actor(s) which would be given in an actor language.
We can use C++ for the actor language (or C if you prefer), with FIFOs defining the inputs and outputs. Here is a simple model to implement in C / C++ code…



$F(x) = x^2/10 - 2x + 3$



This program will output values of fx(x) from x starting at x=1. Minavg3 will collect 3 values, output the average to disp and the minimum value to min. Min will have a memory element, remembering the minimum value seen before and return the latest minimum value. Disp will just display the average of the last 3 values and the minimum value found so far.

SDF → C Code Example

Start by thinking up function prototypes for the functions you will use…

You can have a few minutes to brainstorm, then I'll show my sample solution.

Suggestion: you could consider defining a simple buffer struct

SDF → C Code Example

Suggested function prototypes:

```
#define MAXBUF 16
struct Buff  {
   int h,t;
   float data[MAXBUF];
};

void fx (float x, Buff& o);

void minavg3 (Buff& in, Buff& min, Buff& avg);

void domin (Buff& in,  Buff& globmin);

void disp (Buff& min,  Buff& avg);
```

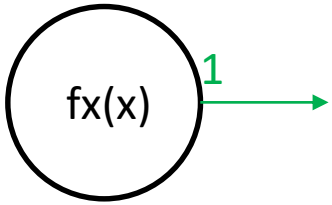
```
void buff_put ( Buff&b, float x )
{     b.data[b.t]=x; // add to buffer
      b.t = (b.t+1)%MAXBUF;
}

float buff_get ( Buff&b )
{     // read from buffer
      float r = b.data[b.h];
      b.h = (b.h+1)%MAXBUF;
      return r;
}
float buff_n ( Buff&b )
{  return
      (MAXBUF-b.h+b.t)
      %MAXBUF;
}   // num items in buffer
```

Now try fleshing out a function or two (no need to do all!)

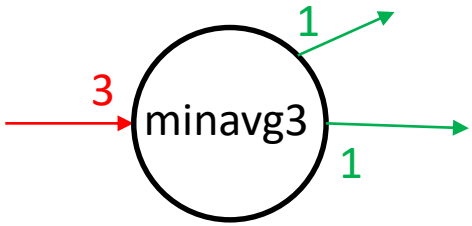
SDF → C Code Example

Fleshing out the functions:

```
void fx (float x, Buff& o)
{
  buff_put(o,x*x/10 – 2*x + 3);
)
```




```
void minavg3 (Buff& in, Buff& min, Buff& avg)
{
  if (buff_n(in)>3) {
       float a = 0;
       float m=99999.999; // or MAX value
       for (int i=0; i<3; i++) {
            float x = buff_get(in);
            if (x<m) m = x;
            a+=x;
            }
        buff_put(min,m); buff_put(avg,(float)a/3.0);
      }
}
```

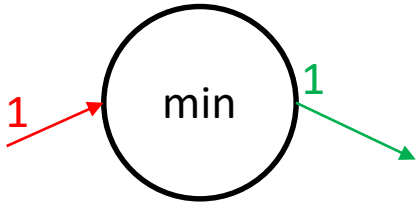


Note code to get tokens shown in red and code to put tokens (i.e. firing) shown in green.

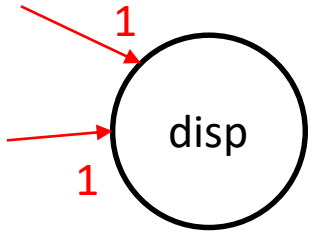
SDF → C Code Example

Fleshing out the functions:

```
void domin (Buff& in,  Buff& min)
{
  static float global_min = 99999.999;
  if (buff_n(in)>=1) {
     float x = buff_get(in);
     if (x<global_min) global_min = x;
     buff_put(min,global_min);
  }
}
```




```
void disp (Buff& avg,  Buff& min)
{
   if (buff_n(avg)>0 && buff_n(min)>0) {
      cout << std::setw(10) << buff_get(avg) << "\t"
           << buff_get(min) << endl;
   }
}
```

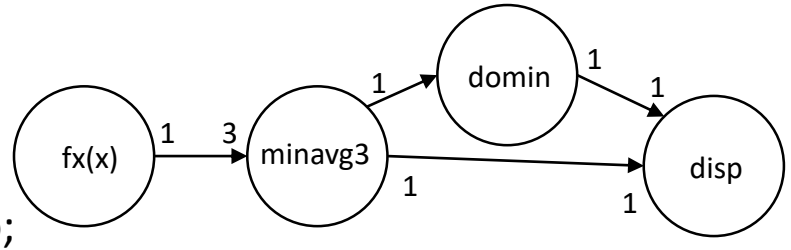


Note: setw just sets column with to 10 chars to make the formatting nicer

SDF → C Code Example

Connecting the pieces…

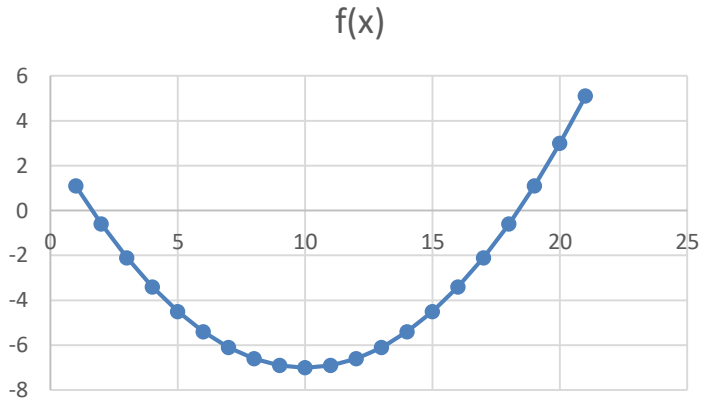
```
// Main function will exercise the SDF actors
int main () {
  // set up the buffers
  Buff fx_o, minavg3_o1, minavg3_o2, min_o;
  buff_init(fx_o); buff_init(minavg3_o1);
  buff_init(minavg3_o2); buff_init(min_o);
  for (int x = 1; x<=21; x++) {
   fx (x,fx_o);
   minavg3(fx_o,minavg3_o1, minavg3_o2);
   min (minavg3_o1, min_o);
   disp (minavg3_o2, min_o);
   }
}
```



Should see displayed:

```
-0.5333333      -2.1
-4.4333333      -5.4
-6.5333333      -6.9
-6.8333333      -7
-5.3333333      -7
-2.0333333      -7
3.06666667      -7
```

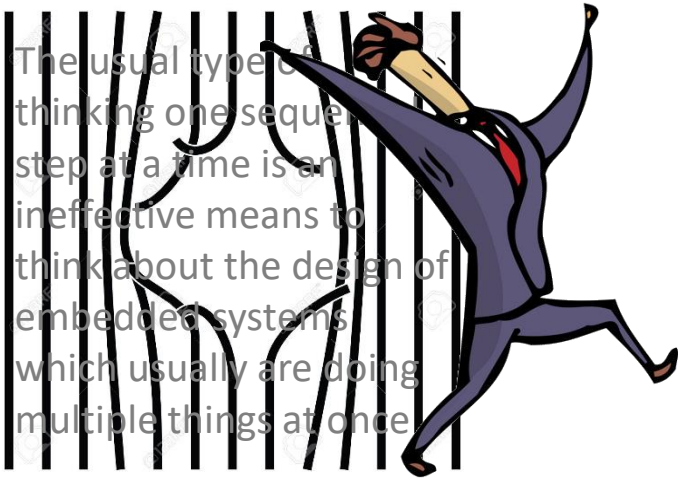
SDF_code.zip



Reflections

There are two main reasons why embedded systems text do so much modelling, namely:

BREAK FREE from traditional, de facto style of thinking that things must happen sequentially



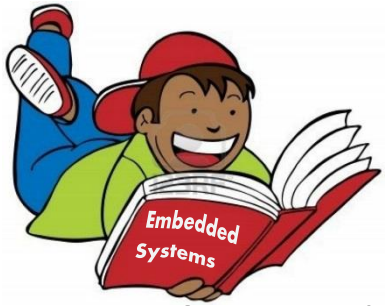
Good Design leads to Good Systems. Moreover, design and modelling features heavily at the engineering level of embedded systems

Next lecture finishing off the aspects of modelling and embedded software design… on towards hardware issues

The Next Episode…

Lecture P03, L11

P03: RPi GPIO
L11: Petri-nets, Levels of Hardware Modeling



Reminder: Read section 2.6, 2.7, 2.10
(section 2.7, 2.8 optional reading)