



Universiteit Leiden

# Embedded Systems: Specification and Modeling (part II)

**Todor Stefanov**

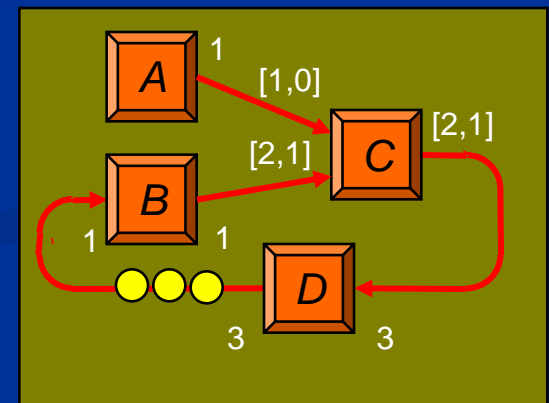
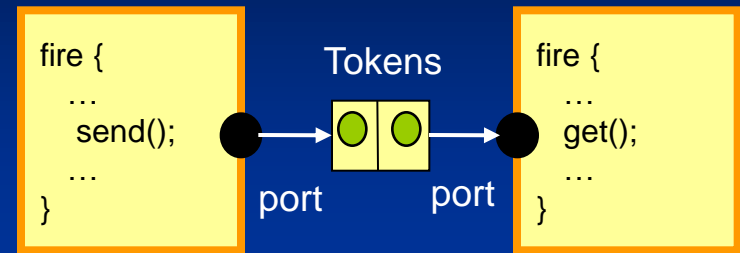
Leiden Embedded Research Center,  
Leiden Institute of Advanced Computer Science  
Leiden University, The Netherlands

# Outline

- Why considering modeling and specification?
- Requirements for Specification Techniques
- Models of Computation
  - State-based models (not considered in this course!)
    - FSM (classical automata)
    - Timed automata
    - StateCharts
  - Petri Nets (not considered in this course!)
    - Condition/Event Nets
    - Predicate/Transition Nets
    - Place/Transition Nets
  - Actor-based Dataflow models
    - SDF, CSDF, PPN, PSDF, PCSDF, PPPN, KPN
- Specification Languages
  - VHDL, SystemC, Others

# Cyclo-Static Dataflow (CSDF)

- Introduced by Lauwereins et al., KU Leuven, 1994
- Network of concurrent actors
  - Passive actors
  - Communication is buffered
- Useful generalization of SDF
  - Variable production/consumption
  - Variations form periodic pattern
- Characteristics of CSDF
  - Compile time analyzable
    - Static schedule
    - Buffer sizes
  - Optimization for memory/speed
  - Usually uses less buffer memory compared to SDF

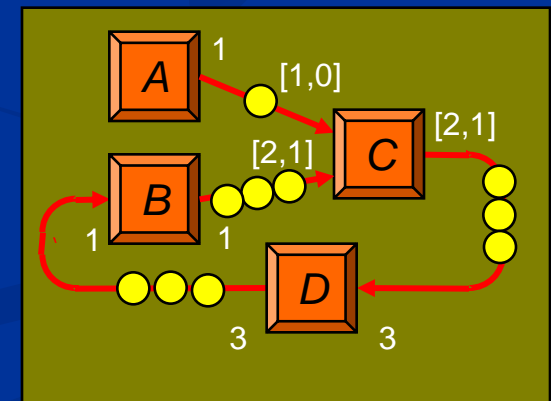


Iteration: **ABBCBCD**

**Actor C has variable production/consumption rate with period of 2**

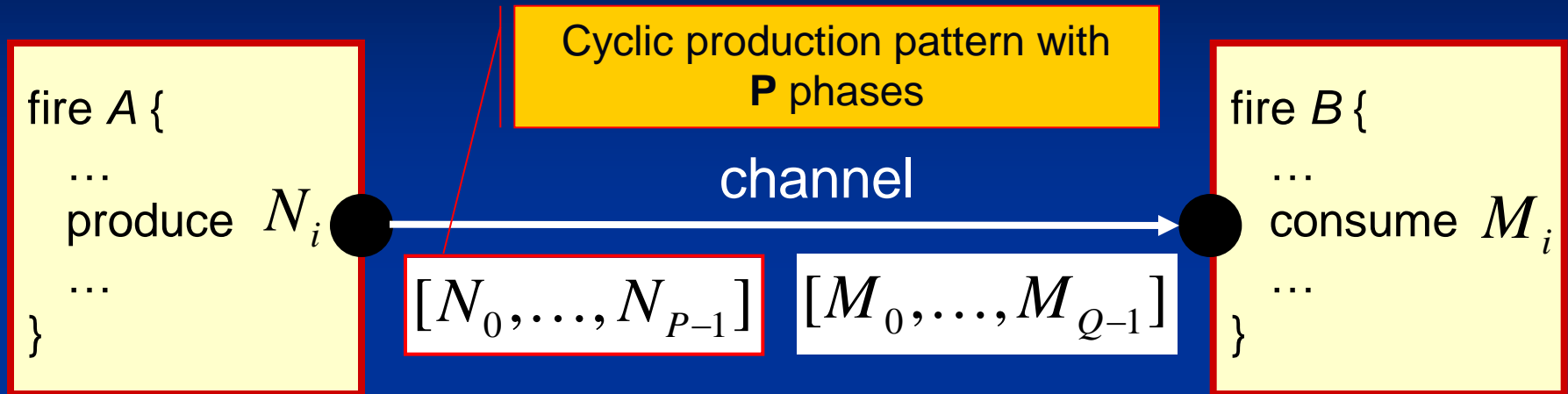
# CSDF Operational Semantics: Firing Rule

- CSDF actor is *enabled* if there is a certain number of tokens on each of its input channels
- *Enabled actor is fired* by removing
  - number of tokens from each of its input channels
  - placing tokens on each of its output channels
- **Iteration:** sequence of actors firings that brings CSDF to its initial state
  - many possible sequences as long as firing rules are obeyed
  - actors can fire in parallel!



Iteration: **ABBBCBCD**

# CSDF: Variable Production and Consumption rate



- How can we exploit cyclic production/consumption for analysis?
- Define a Balance equations for each channel:

$$r_A * \sum_{i=0}^{P-1} N_i = r_B * \sum_{i=0}^{Q-1} M_i$$

$$f_A = P * r_A; f_B = Q * r_B$$

number of tokens consumed per phase

**aggregated** number of firings per iteration

**actual** number of firings per iteration

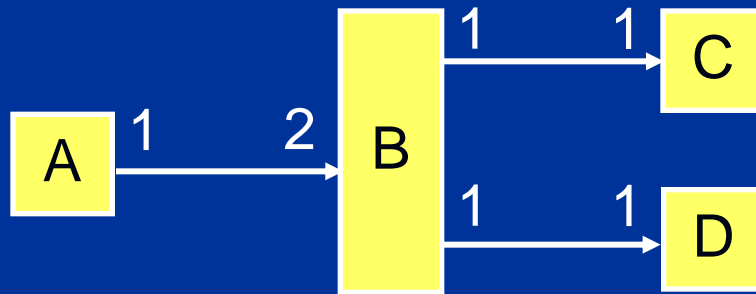
# CSDF: Scheduling

- Scheduling is much like SDF
  - Balance equations establish relative firing rates as for SDF
  - Any scheduling algorithm that avoids buffer underflow will produce a valid schedule if one exists
- Advantage: even more schedule flexibility
- Makes it easier to avoid large buffers

# CSDF vs. SDF

- SDF actors consume/produce the same number of tokens at each firing!
- Usually this lead to larger buffer requirements in SDF compared to CSDF
- Example: Model a distributor actor (i.e., actor B)

SDF model of B

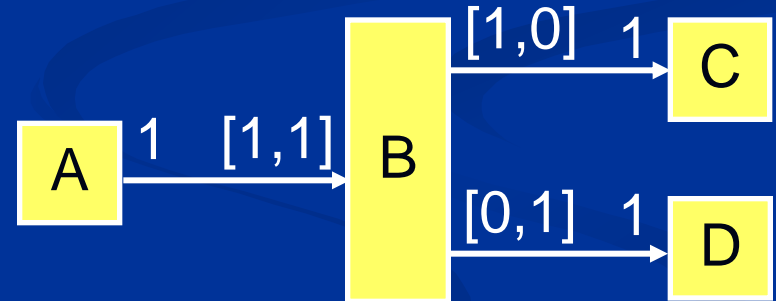


**Schedule: AABCD**

**Requires: 4 units of buffer memory**

**2 for edge (AB) and 1 for (BC) and (BD)**

CSDF model of B



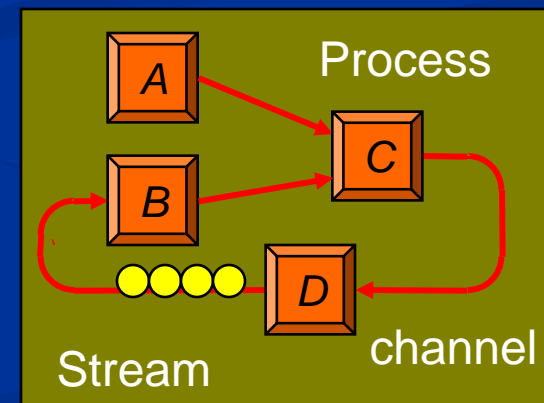
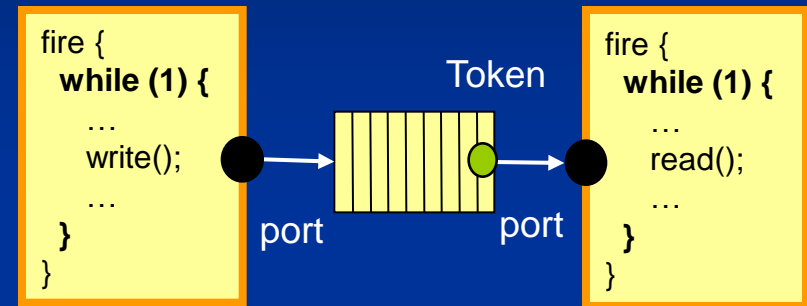
**Schedule: ABCABD**

**Requires: 3 units of buffer memory**

**1 for each edge (AB), (BC), and (BD)**

# Polyhedral Process Network (PPN)

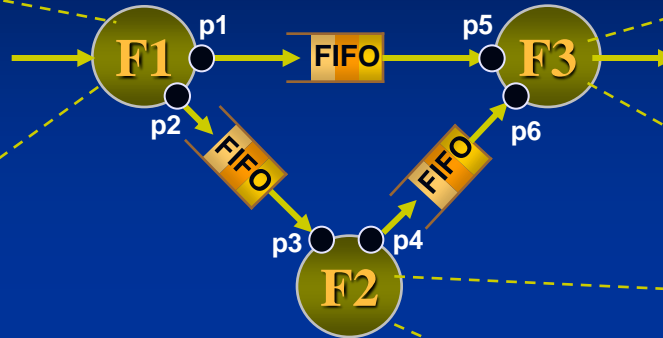
- Introduced at LIACS in 2000
- Network of concurrent processes
  - Active actors (processes)
  - Communicate over bounded FIFOs
- Processes:
  - Perform some computation
  - Communicate data (read/write)
    - blocking read
    - blocking write
- Process behaviour expressed as *parameterized polyhedral descriptions*
- Characteristics of PPNs
  - Compile time analyzable
  - Deterministic execution
  - Do not impose a particular schedule





# PPN: Example

```
int M = 10, P = 3;
for( i=1; i <= M; i++) {
  out = F1( );
  if( i <= P)
    write( p2, out );
  else
    write( p1, out );
}
```



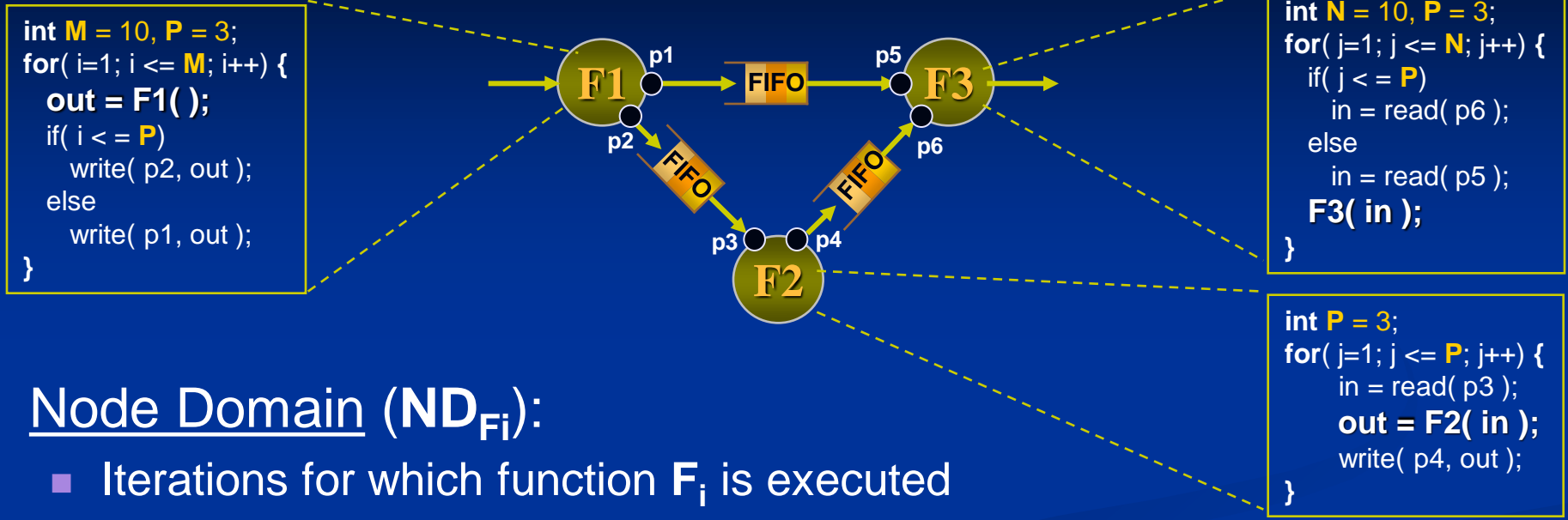
```
int N = 10, P = 3;
for( j=1; j <= N; j++) {
  if( j <= P)
    in = read( p6 );
  else
    in = read( p5 );
  F3( in );
}
```

```
int P = 3;
for( j=1; j <= P; j++) {
  in = read( p3 );
  out = F2( in );
  write( p4, out );
}
```

## ■ Polyhedral Process Networks (PPN)

- Equivalent to Static Affine Nested-loop Programs
- Can be derived automatically
- Well defined structure of a process
  - READ - EXECUTE - WRITE code sections
  - Parameterized, static, and affine control in
    - for-loop bounds
    - if-conditions
  - Parameters cannot change values at run-time!

# PPN: Some Definitions



- Node Domain ( $ND_{F_i}$ ):

- Iterations for which function  $F_i$  is executed
- Example:  $ND_{F_3}$  is  $1 \leq j \leq N$

- Input Port Domain ( $IPD_{P_i}$ ):

- Iterations for which port  $P_i$  is read
- Example:  $IPD_{P_5}$  is  $P < j \leq N$

- Output Port Domain ( $OPD_{P_i}$ ):

- Iterations for which port  $P_i$  is written
- Example:  $OPD_{P_2}$  is  $1 \leq i \leq P$

- Mapping ( $M_{P_j P_i}$ ):

- Relation between  $IPD_{P_j}$  and  $OPD_{P_i}$  corresponding to channel  $(P_j P_i)$
- Example:  $M_{P_5 P_1} : i = 1 * j$ , where  $j \in IPD_{P_5}$  and  $i \in OPD_{P_1}$

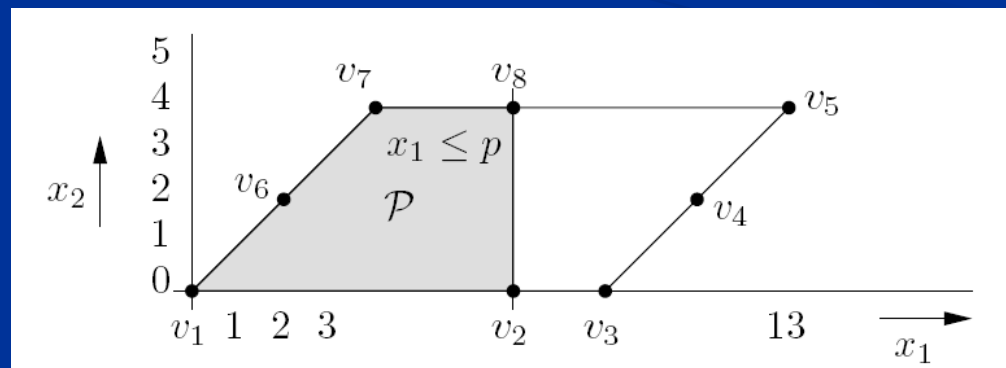
# PPN: Polyhedral Description (1)

- Process behavior expressed as parameterized polyhedrons
- What is a parameterized polyhedron?

$$\mathcal{P}(\mathbf{p}) = \{\mathbf{x} \in \mathbb{Q}^n \mid \mathbf{Ax} = \mathbf{Bp} + \mathbf{b} \wedge \mathbf{Cx} \geq \mathbf{Dp} + \mathbf{d}\}$$

- Set of points  $\mathbf{x}$  in the  $n$ -dimensional space satisfying some *constraints* where
  - $\mathbf{p} \in \mathbb{Q}^m$  is a vector of parameters
  - $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$  are integral matrixes
  - $\mathbf{b}$  and  $\mathbf{d}$  are an integral vectors
- Example

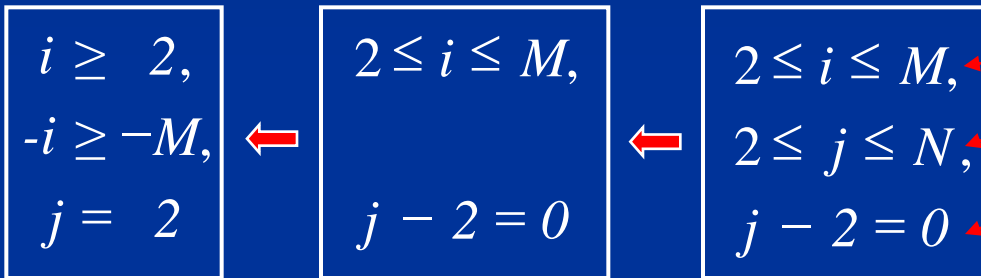
$$\mathcal{P}(p) = \{(x_1, x_2) \in \mathbb{Q}^2 \mid 0 \leq x_2 \leq 4 \wedge x_2 \leq x_1 \leq x_2 + 9 \wedge x_1 \leq p \wedge p \leq 40\}$$



# PPN: Polyhedral Description (2)

Every Node, Input and Output Port Domain can be represented as Parameterized Polyhedron

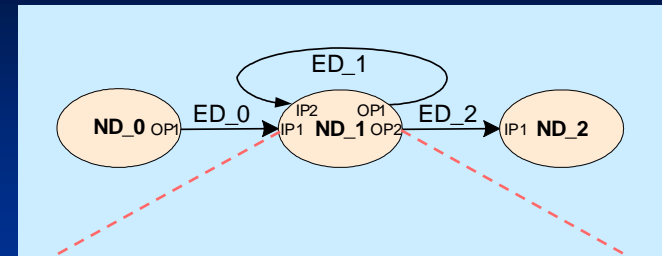
- Example:  $IPD_{IP1}$  as polyhedron



$$\begin{aligned}
 &1*i + 0*j \geq 0*M + 0*N + 2, \\
 &-1*i + 0*j \geq -1*M + 0*N + 0, \\
 &0*i + 1*j = 0*M + 0*N + 2
 \end{aligned}$$

$$P(M, N) = \left\{ (i, j) \in \mathbb{Z}^2 \mid \left[ 0 \ 1 \right] * \begin{pmatrix} i \\ j \end{pmatrix} = \left[ 0 \ 0 \right] * \begin{pmatrix} M \\ N \end{pmatrix} + 2 \cap \left[ \begin{matrix} 1 & 0 \\ -1 & 0 \end{matrix} \right] * \begin{pmatrix} i \\ j \end{pmatrix} \geq \left[ \begin{matrix} 0 & 0 \\ -1 & 0 \end{matrix} \right] * \begin{pmatrix} M \\ N \end{pmatrix} + \begin{pmatrix} 2 \\ 0 \end{pmatrix} \right\}$$

$$\mathcal{P}(\mathbf{p}) = \{ \mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} = B\mathbf{p} + \mathbf{b} \wedge C\mathbf{x} \geq D\mathbf{p} + \mathbf{d} \}$$



```

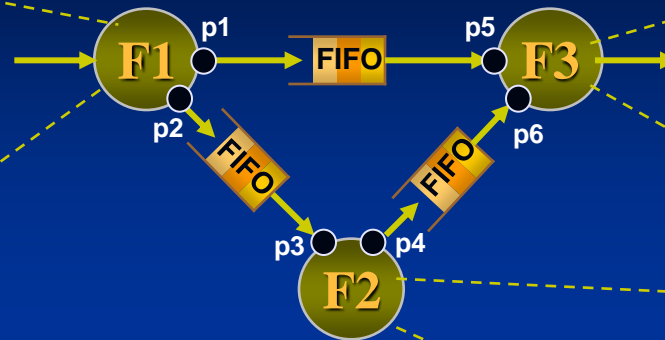
1 // process ND_1
2 void main() {
3   for( int i=2; i<=M; i++)           CONTROL
4   for( int j=2; j<=N; j++) {
5     if( j-2 == 0 )
6       read( IP1, in_0 );             READ
7     if( j-3 >= 0 )
8       read( IP2, in_0 );
9     Transformer( in_0, out_0 );     EXECUTE
10    if( -j+N-1 >= 0 )
11      write( OP1, out_0 );           WRITE
12    if( j-N == 0 ) {
13      write( OP2, out_0 );
14    } // for j
15 } // main

```

# PPN: Some Remarks

```

int M = 5, P = 3;
for( i=1; i <= M; i++) {
  out = F1( );
  if( i <= P)
    write( p2, out );
  else
    write( p1, out );
}
    
```



```

int N = 5, P = 3;
for( j=1; j <= N; j++) {
  if( j <= P)
    in = read( p6 );
  else
    in = read( p5 );
  F3( in );
}
    
```

```

int P = 3;
for( j=1; j <= P; j++) {
  in = read( p3 );
  out = F2( in );
  write( p4, out );
}
    
```

- PPNs allow to perform formal algebraic transformations, i.e.,
  - Affine (linear) transformations on polyhedrons
- PPNs allow to set and solve optimization problems (such as FIFO size calculations, etc.)
  - Expressed as Integer Linear Programming (ILPs)
- PPNs can be converted to CSDFs
  - PPNs are very compact representation of some class of CSDF
  - Example:



# Decidable Dataflow Models

- SDF, CSDF, PPN are Decidable Models
  - have limited expressive power
  - they can model only applications with static behavior
- However, there are many applications that employ high-level dynamics in their behavior
  - User interface functionality
  - Mode changes
  - Adaptive algorithms
  - Behavior changes depending on available processing resources, etc...
- How to solve this problem?

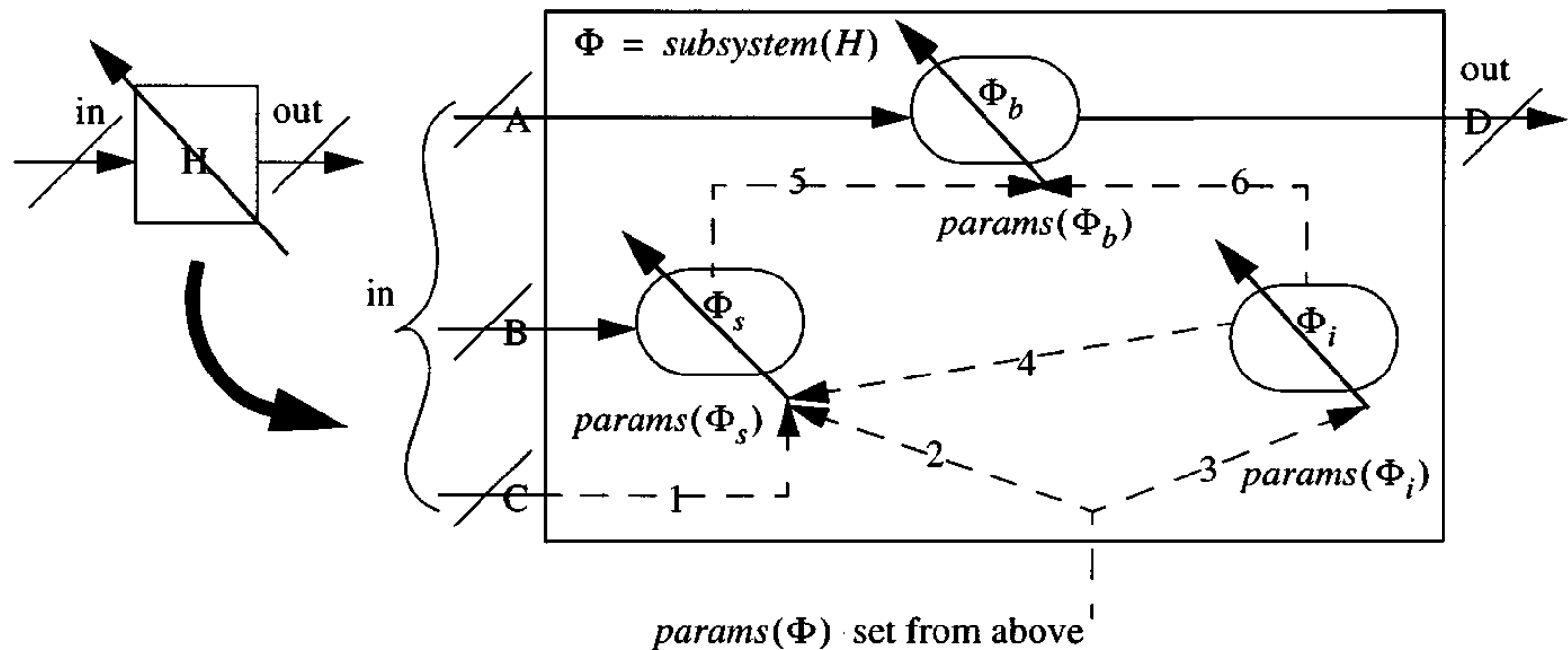
# Partly Decidable Dataflow Models

- Observation: Key *subsystems* of dynamic applications still
  - exhibit large amounts of “quasi-static” structure
  - stay fixed across significant windows of time
- Dynamic dataflow models have been proposed
  - address the limitation of decidable models by
  - abandoning most restrictions related to decidable dataflow
- However, these models are limited
  - in their ability to exploit the quasi-static structures
  - almost NO analysis can be done at design time
- Therefore, Partly Decidable Models are proposed!
- The Key is the Dynamic Parameterization of actors!

# Dynamic Parametrization of Actors

The Key concept is:

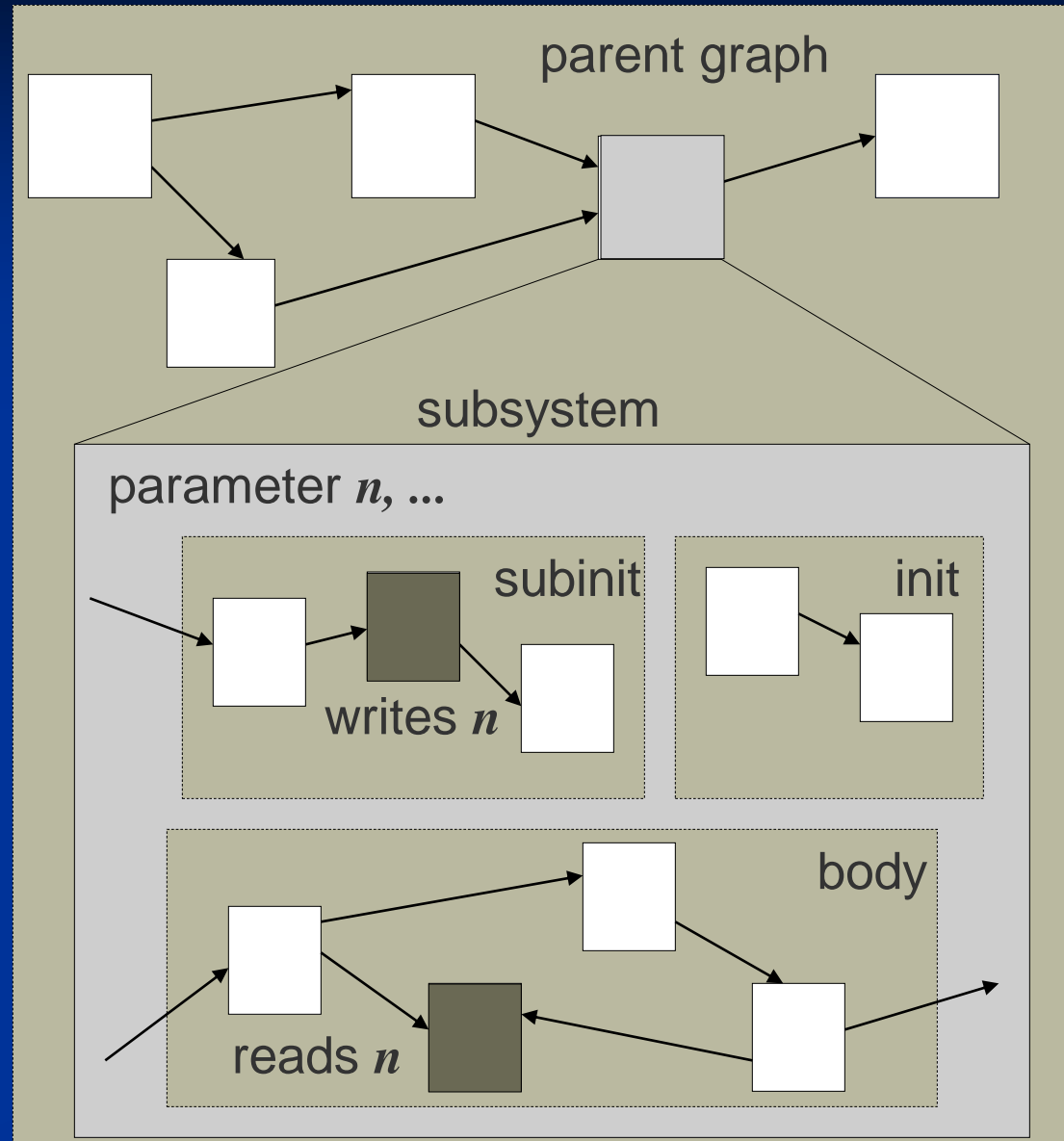
- Introduce Dynamic Parameters (global and/or local)
- Do Structured Control of Dynamic Parameters





# Parameterized Dataflow Concept

- Hierarchical modeling
- **Subsystem** is composed of 3 parameterized DF graphs:
  - **init**, **subinit**, **body**
- **Subsystem parameters**
  - configured in **init**/**subinit**
  - used in **body**
- Dynamically **reconfigurable**
  - **init** invoked at the beginning of each invocation of parent graph
  - **subinit** invoked at the beginning of each invocation of the associated subsystem
  - **body** invoked after each invocation of subinit

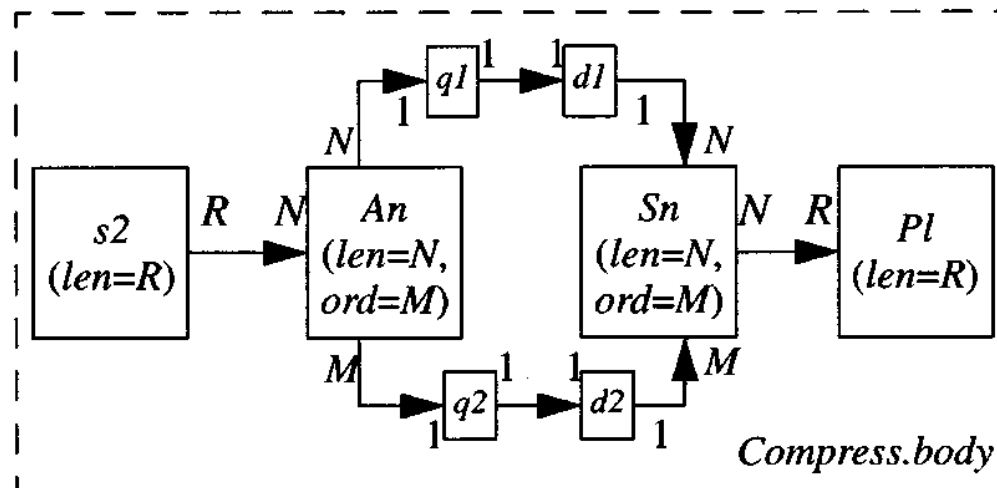
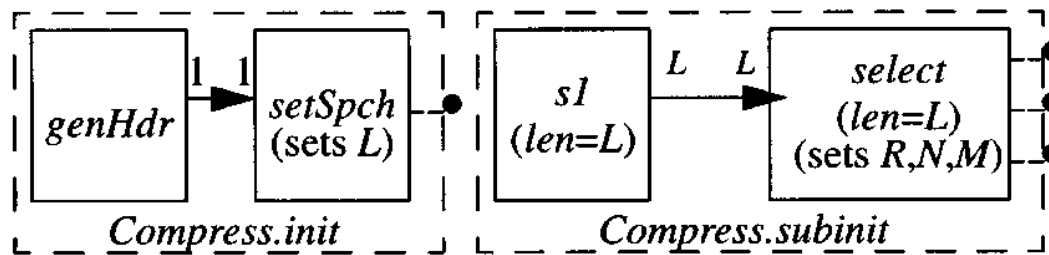


# Meta-modeling with parameterized dataflow concept

- Parameterized dataflow concept can be applied to any dataflow MoC denoted with  $X$
- Parameterized dataflow +  $X \rightarrow$  “Parameterized  $X$ ”
- Examples of parameterized dataflow MoC that we will look at are:
  - *Parameterized Synchronous Dataflow (PSDF)*
  - *Parameterized Cyclo-Static Dataflow (PCSDF)*
  - *Parameterized Polyhedral Process Network (PPPN)*

# PSDF Example: Speech Compression

subsystem *Compress*,  
 {params =  $L, R, N, M$ }  
 $\text{gcd}(R, N) = N$

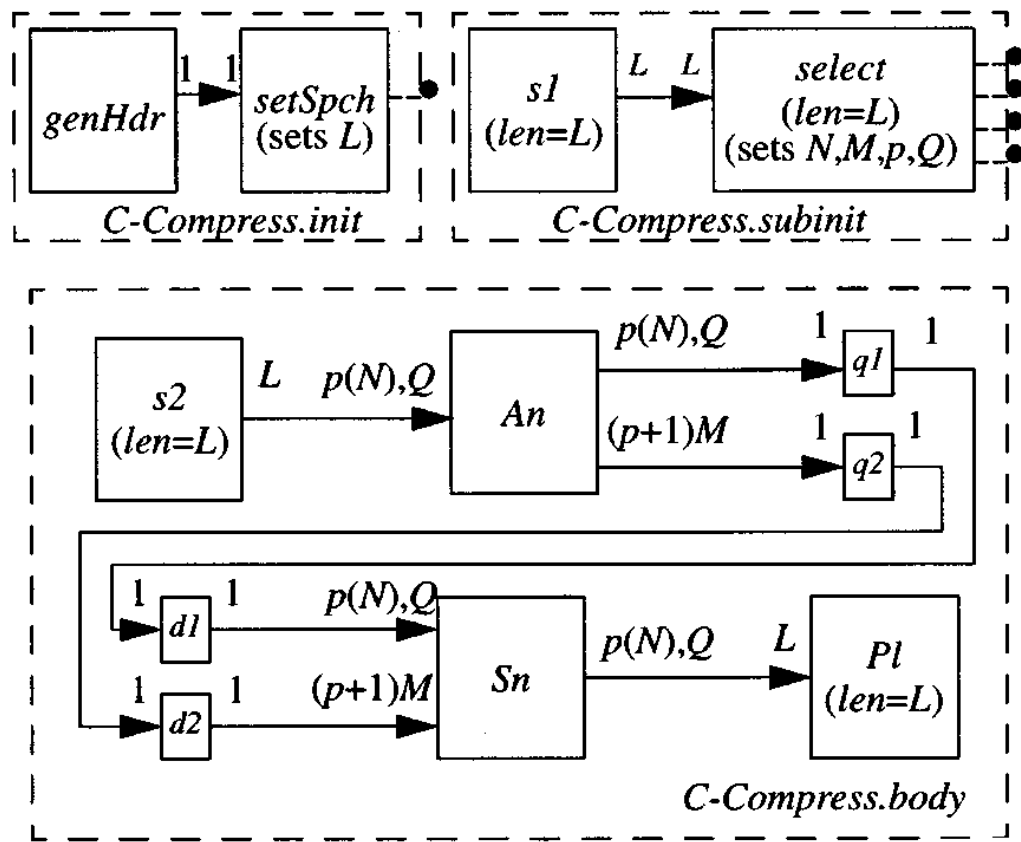


```

while (1) {
    /*fire Compress.init */
    fire genHdr;
    fire setSpch; /* sets L */
    /* fire Compress.subinit */
    fire s1;
    fire select /* sets R, N, M */
    /*fire Compress.body */
    fire s2
    repeat (R/N) times {
        fire An
        repeat (N) times {
            fire q1; fire d1
        }
        repeat (M) times {
            fire q2; fire d2
        }
        fire Sn
    }
    fire Pl
}
    
```

# PCSDF Example: Speech Compression

subsystem *C-Compress*,  
{ params =  $L, N, M, p, Q$ }



```
/* fire C-Compress.body */
```

```
fire s2
```

```
repeat (p) times {
```

```
  fire An
```

```
  repeat (N) times {
```

```
    fire q1; fire d1
```

```
  }
```

```
  repeat (M) times {
```

```
    fire q2; fire d2
```

```
  }
```

```
  fire Sn
```

```
}
```

```
fire An
```

```
repeat (Q) times {
```

```
  fire q1; fire d1
```

```
}
```

```
repeat (M) times {
```

```
  fire q2; fire d2
```

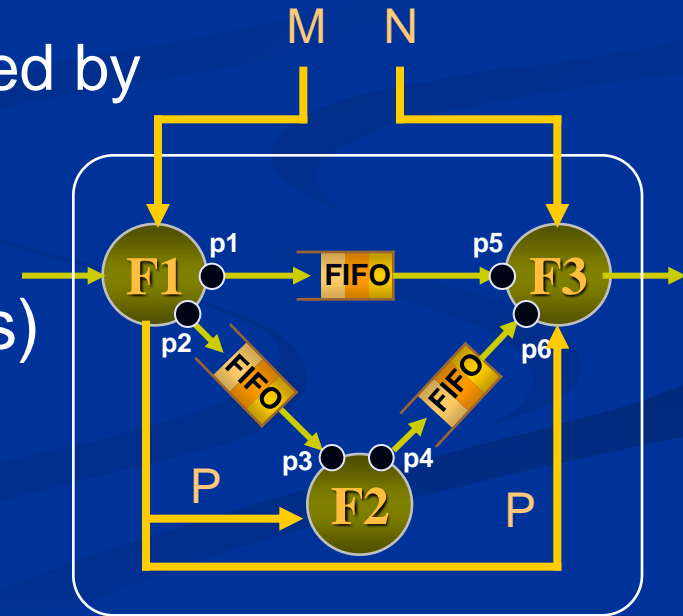
```
}
```

```
fire Sn
```

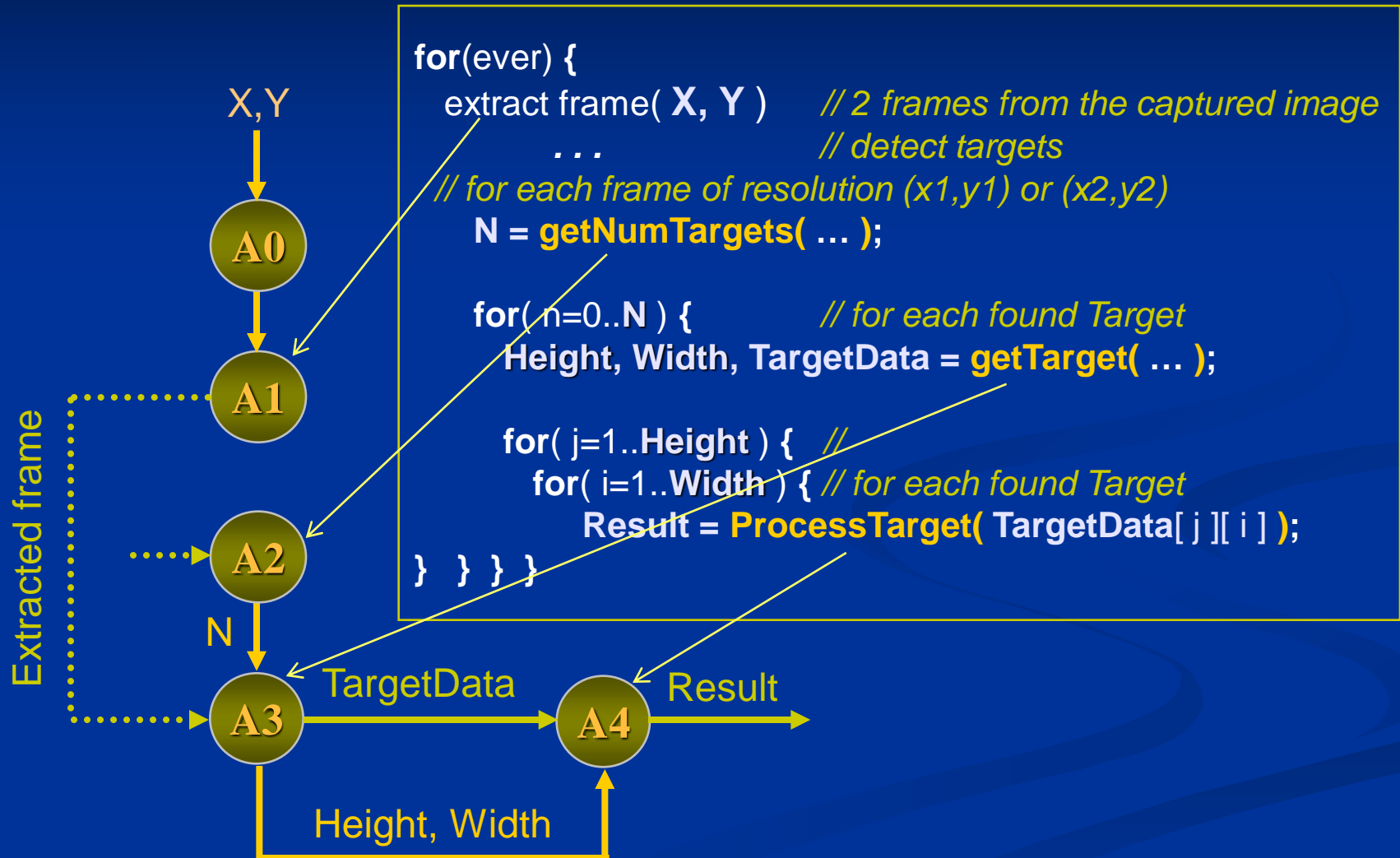
```
fire Pl
```

# Parameterized PPN (P<sup>3</sup>N)

- Extends the PPN model by allowing parameters to change values at run-time
- Special control channels are added to set the values of the parameters
  - Global parameters - values are changed by the environment
  - Local parameters - values are changed by nodes in the network
- Semantics defined to allow some **compile time analysis** (for buffer sizes)
- Parameter values are changed in a way that **preserves consistency** (exec. with bounded buff memory)



# P<sup>3</sup>N Example: Low Speed Obstacle Detection

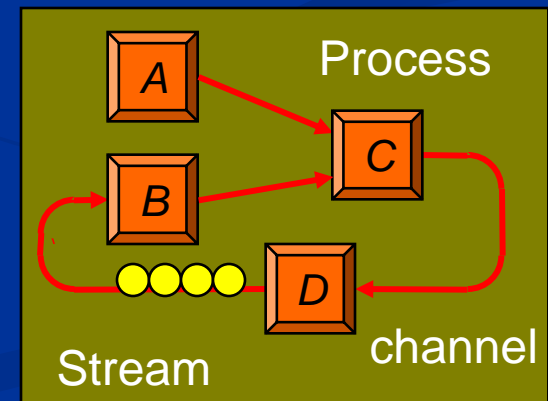
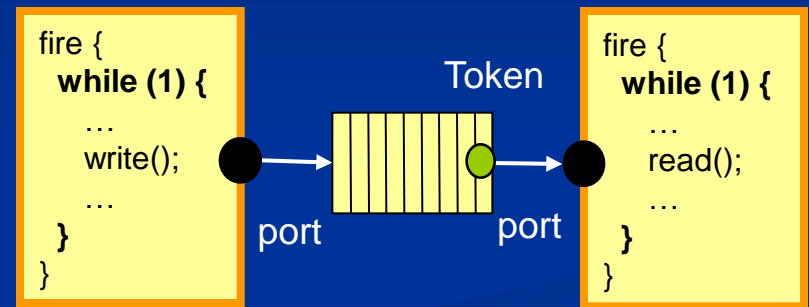


# Undecidable Dataflow Models

- Models for which the following questions cannot be answered at compile time:
  - Is the model deadlock free?
  - Can the model execute with bounded buffer memory?
  - Does a schedule exist?
- Undecidable models in this sense are
  - Boolean/Integer Data Flow (BDF, IDF)
  - Dynamic Data Flow (DDF)
  - **Kahn Process Network**

# Kahn Process Network (KPN)

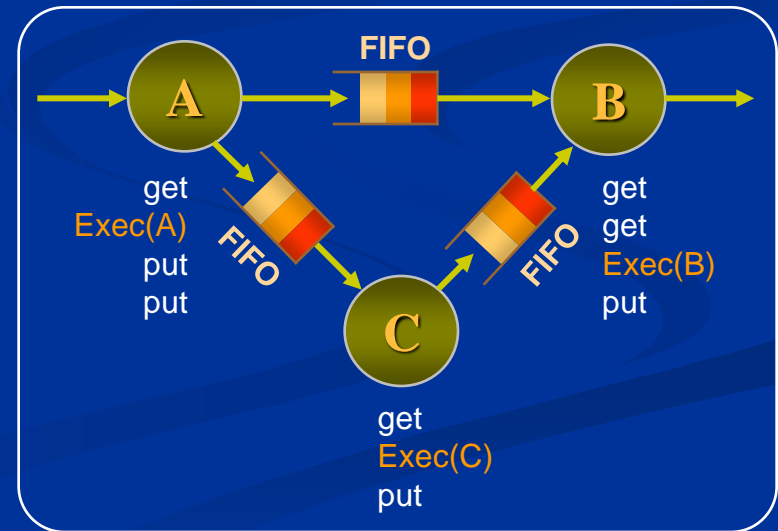
- Proposed by Kahn in 1974 as a scheme for parallel programming
  - Laid the theoretical foundation for dataflow
- Network of concurrent processes
  - Active actors
  - Communicate over *unbounded* FIFOs
- Synchronization
  - Blocking read on an empty channel





# KPN: Operational Semantics

- Processes either perform computation or communicate
- Reading an empty channel blocks until data is available
  - Process can not wait for data on multiple channels at the same time
- Writing to a channel is non-blocking
- There is only one producer and one consumer per channel
- Characteristics of KPN
  - **Deterministic**
  - **Distributed Control**
    - no global schedule needed
  - **Distributed Memory**
    - no shared memory used
    - no memory contention

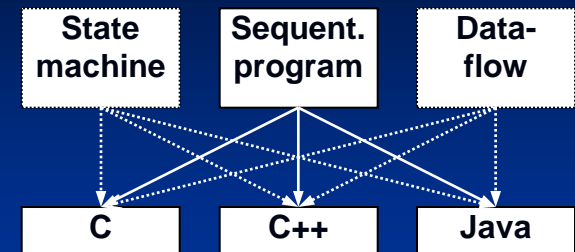


# KPN: Some Remarks

- Well suited for specifying streaming application
  - signal and image processing
- Whether a KPN can execute in bounded memory is undecidable
- In general, KPNs are difficult to impossible to analyze at compile time
- BUT KPNs are very useful because
  - they are deterministic
  - dynamic streaming application can be modeled efficiently

# Specification Languages

**Do not confuse  
Specification Languages with  
Models of Computation!!!**



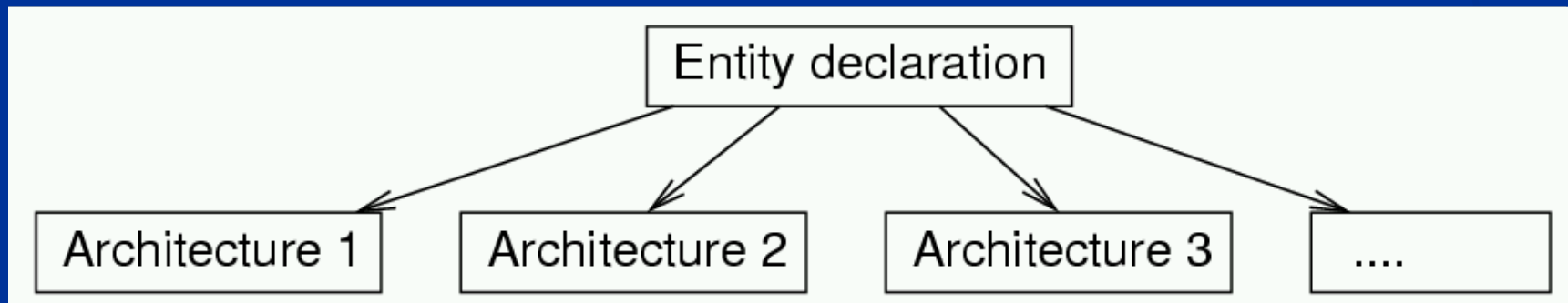
- Models of Computation describe system behavior
  - Conceptual notion, e.g., sequential execution, dataflow, FSM
- Specification Languages capture Models of Computation
  - Concrete syntax (textual or graphical) form, e.g., C, C++, Java
- Variety of languages can capture one model
  - E.g., C, C++, Java → sequential execution model
- One language can capture variety of models
  - E.g., C++ → sequential execution model, dataflow model, state machine model
- Certain languages are better at capturing certain model of computation

# Hardware Description Languages

- HDL = hardware description language
- Textual HDLs replaced graphical HDLs in the 1980s (better for complex behavior)
- Example of HDL is VHDL language:
  - VHDL = VHSIC hardware description language
  - VHSIC = very high speed integrated circuit
  - 1980: Definition started
  - 1984: first version of the language defined, based on ADA, PASCAL
  - 1987: IEEE standard 1076; 1992 revision;
  - Extension: VHDL-AMS models analog
- Another example is Verilog
  - Preferred in US

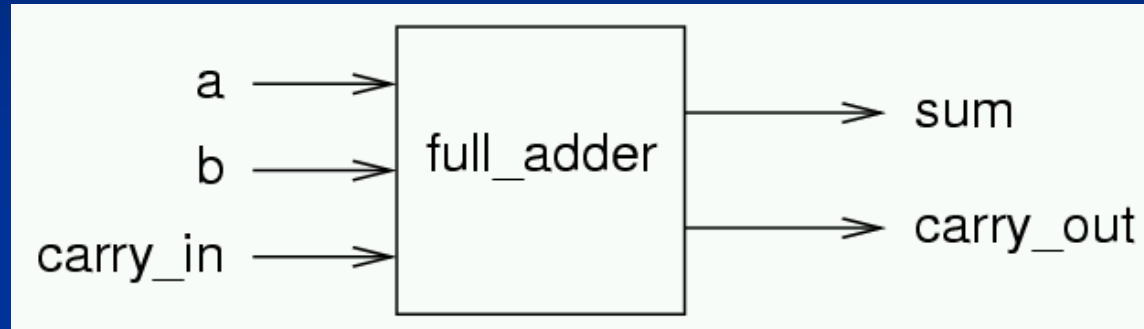
# VHDL: Entities and Architectures

- Each design unit is called an *entity*
- Entities are comprised of entity declarations and one or several *architectures*



- Each architecture includes a model of the entity
- The used architecture specified in a *configuration*

# VHDL: Entity Declaration



```
entity full_adder is  
  port(a, b, carry_in: in Bit; -- input ports  
        sum,carry_out: out Bit); --output ports  
end full_adder;
```

# VHDL: Architecture

**architecture** behavior of full\_adder is

**begin**

sum        <= (a xor b) xor carry\_in **after** 10 ns;

carry\_out <= (a and b) or (a and carry\_in) or  
                  (b and carry\_in)        **after** 10 ns;

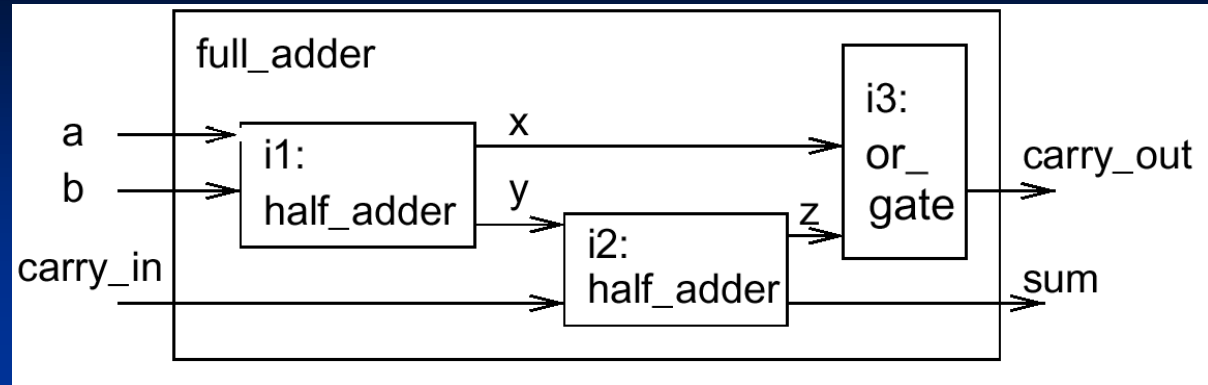
**end** behavior;

Architectural bodies can include:

- behavioral model
- structural model

Bodies not referring to hardware components are called behavioral bodies

# VHDL: Structural Body



```
architecture structure of full_adder is
component half_adder
    port (in1,in2:in Bit; carry:out Bit; sum:out Bit);
end component;
component or_gate
    port (in1, in2:in Bit; o:out Bit);
end component;
signal x, y, z: Bit;      -- local signals
begin                    -- port map section
    i1: half_adder port map (a, b, x, y);
    i2: half_adder port map (y, carry_in, z, sum);
    i3: or_gate    port map (x, z, carry_out);
end structure;
```



# VHDL: Processes

## Processes model parallelism in hardware

### General syntax:

```
label:           --optional  
process  
  declarations --optional  
begin  
  statements   --optional  
end process
```

### Example:

```
process  
begin  
  a <= b after 10 ns  
end process;
```

# VHDL: Wait Statements

## Processes synchronize via WAIT-statements

Four possible kinds of **wait**-statements:

- **wait on *signal list***;
  - wait until signal changes;
  - Example: **wait on a**;
- **wait until *condition***;
  - wait until condition is met;
  - Example: **wait until c='1'**;
- **wait for *duration***;
  - wait for specified amount of time;
  - Example: **wait for 10 ns**;
- **wait**;
  - suspend indefinitely

```
process
begin
    prod <= x and y ;
    wait on x,y;
end process;
```

# VHDL: Sensitivity List

Sensitivity lists are a shorthand for **wait on signal list** at the end of the process body:

```
process (x, y)
  begin
    prod <= x and y ;
  end process;
```

*is equivalent to*

```
process
  begin
    prod <= x and y ;
    wait on x,y;
  end process;
```

# VHDL Summary

- Behavioral hierarchy (procedures & functions)
- Structural hierarchy but no nested processes
- No object-orientation
- Static number of processes
- Complicated simulation semantics
- Too low level for initial specification
- Good as intermediate language for hardware generation

# SystemC language

- Why SystemC if we have VHDL or Verilog?
- Many standards (e.g. the GSM and MPEG-standards) are published in C
  - Using special HDLs require translation from C
- The functionalities of systems are provided by a mix of HW (in HDL) and SW (in C) components
- If different languages are used for the description of HW and SW
  - Simulations require an interface between HW and SW simulators
- Aims at describe SW and HW in same language
  - SW and HW developers are very familiar with C/C++

# SystemC: Features

- C++ class library: including required objects for modeling HW components in a SW language
- Concurrency: via processes, controlled by sensitivity lists and calls to wait primitives
- Time: Units ps, ns,  $\mu$ s, etc ...
- Support of bit-datatypes: *bitvectors* of different lengths; *multiple-valued logic* (2 and 4 resolution, i.e., '0', '1', 'u'-undefined, and 'z'-high impedance)
- Communication: plug-and-play channel models, allowing easy composition of IP components

# SystemC: Language Architecture

Channels for MoCs  
Kahn process networks, SDF, etc

Methodology-specific Channels  
Master/Slave library

Elementary Channels  
Signal, Timer, Mutex, Semaphore, FIFO, etc

## Core Language

Module

Ports

Processes

Events

Interfaces

Channels

**Event-driven simulation kernel**

## Data types

Bits and bit-vectors

Arbitrary precision integers

Fixed-point numbers

4-valued logic types, logic-vectors

C++ user defined types

C++ Language Standard