Static Scheduling for Embedded Systems

Luciano Lavagno

University of Udine and Cadence Berkeley Labs

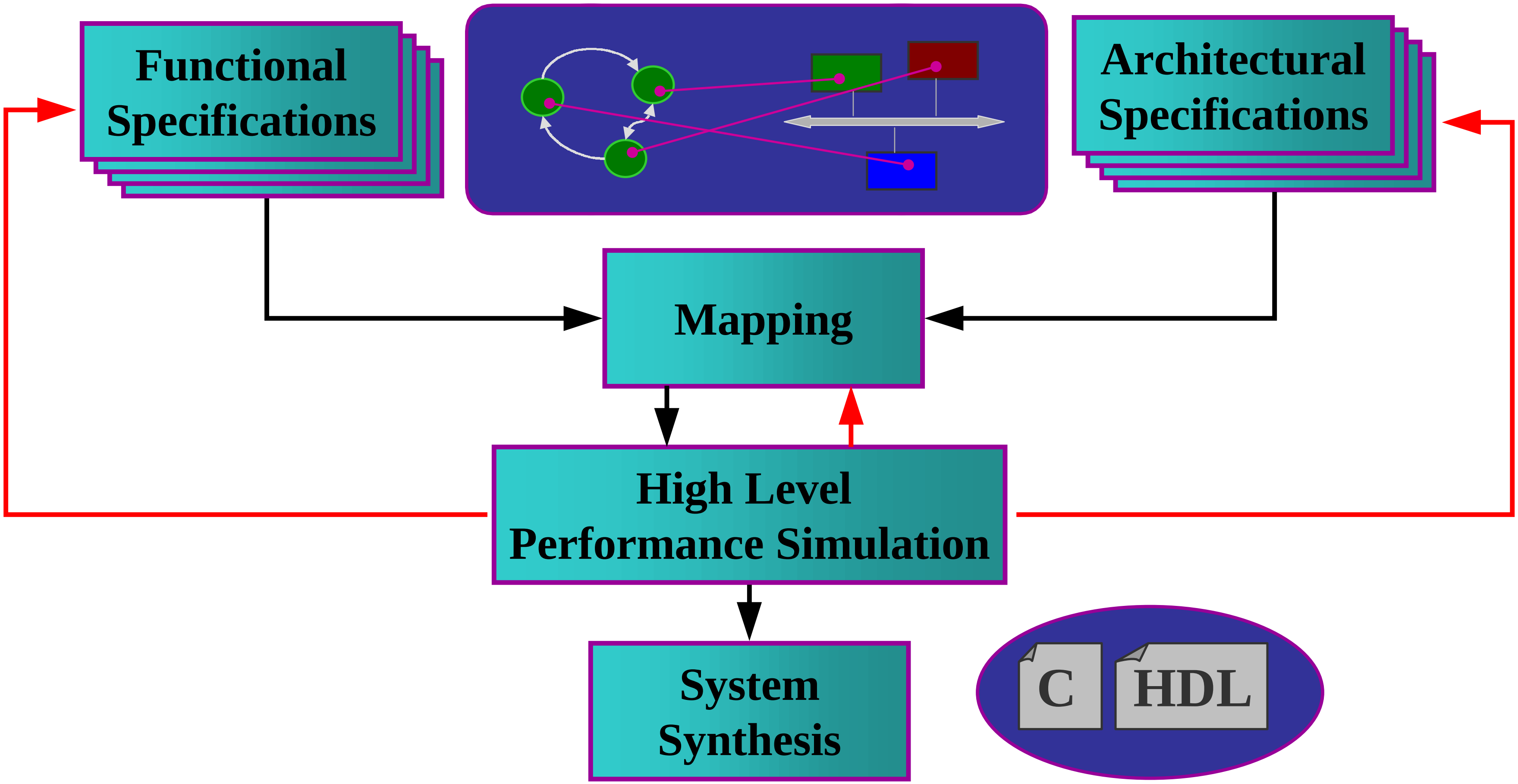
Joint work with:

Jordi Cortadella, Alex Kondratyev, Marc Massot,
Sandra Moral, Claudio Passerone,
Alberto Sangiovanni-Vincentelli, Marco Sgroi,
Yosinori Watanabe

Outline

- Motivation
- Static Scheduling of dataflow networks
  - schedulability
  - code and data size optimization
- Quasi-Static Scheduling of process networks using Petri nets
  - Free Choice nets
  - Non-Free-Choice nets
- Conclusions

Function-architecture co-design



Embedded Software Synthesis

- Specification: concurrent functional netlist
(Kahn processes, dataflow actors, SDL processes, …)
- Software implementation:
(smaller) set of concurrent software tasks
- Two sub-problems:
  - Generate code for each task
(from code fragments of functional blocks)
  - Schedule tasks dynamically
(to satisfy real-time constraints)
- Goals:
  - minimize real-time scheduling overhead
  - maximize effectiveness of compilation

Dataflow networks

- A little history
- Syntax and semantics
  - actors, tokens and firings
- Scheduling of Static Dataflow
  - static scheduling
  - code generation
  - buffer sizing
- Other Dataflow models
  - Boolean Dataflow
  - Dynamic Dataflow

Dataflow networks

- Powerful formalism for data-dominated system specification
- Partially-ordered model (no over-specification)
- Deterministic execution independent of scheduling
- Used for
  - simulation
  - code generation (scheduling and memory allocation)

  for Digital Signal Processors (HW and SW)

A bit of history

- Kahn process networks (‘58): formal model
- Karp computation graphs (‘66): seminal work
- Dennis Dataflow networks (‘75): programming language for MIT DF machine
- Lee’s Static Data Flow networks (‘86): efficient static scheduling
- Several recent implementations (Ptolemy, Khoros, Grape, SPW, COSSAP, SystemStudio, DSPStation, Simulink, …)

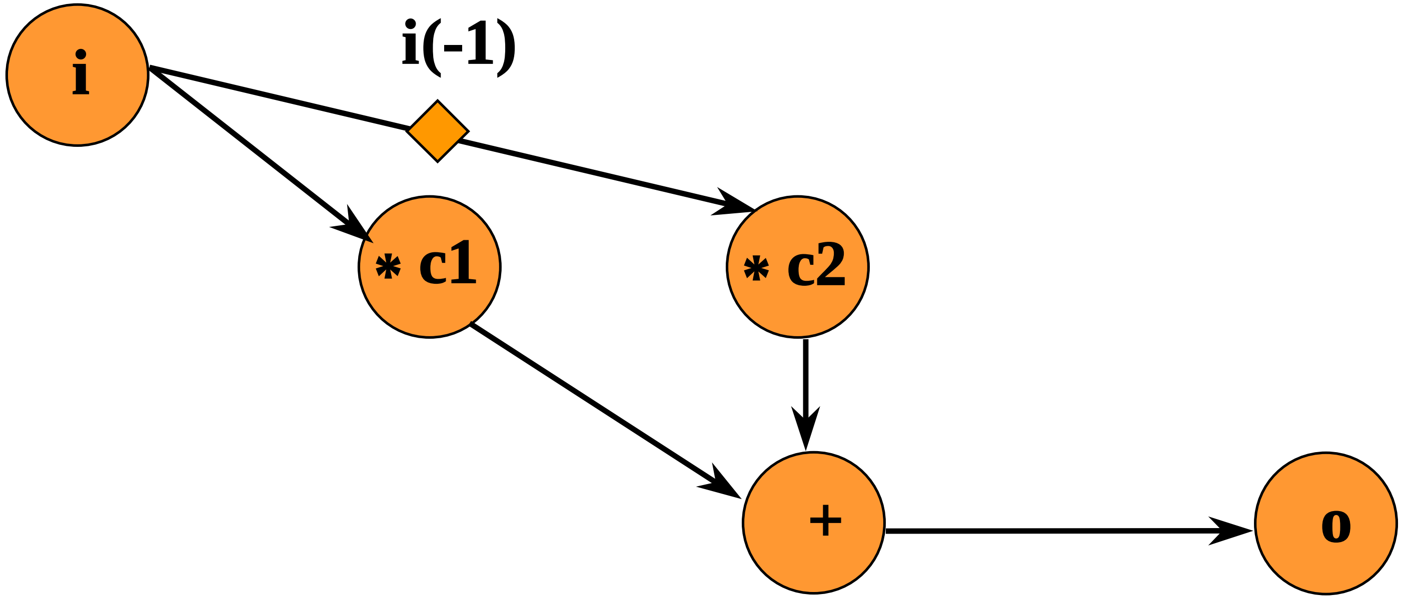
Intuitive semantics

- (Often stateless) actors perform computation
- Unbounded FIFOs perform communication via *sequences of tokens* carrying values
  - (matrix of) integer, float, fixed point
  - image of pixels, …..
- State implemented as self-loop
- Determinacy:
  - unique output sequences given unique input sequences
  - Sufficient condition: *blocking read*

    (process cannot test input queues for emptiness)

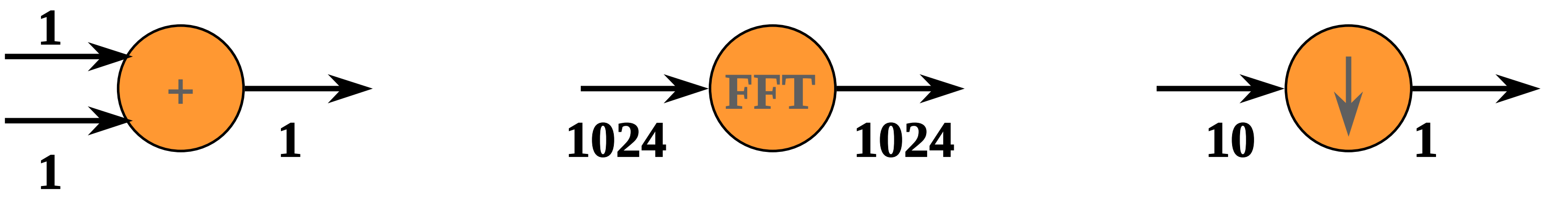
Intuitive semantics

- Example: FIR filter
  - single input sequence i(n)
  - single output sequence o(n)
  - o(n) = c1 * i(n) + c2 * i(n-1)

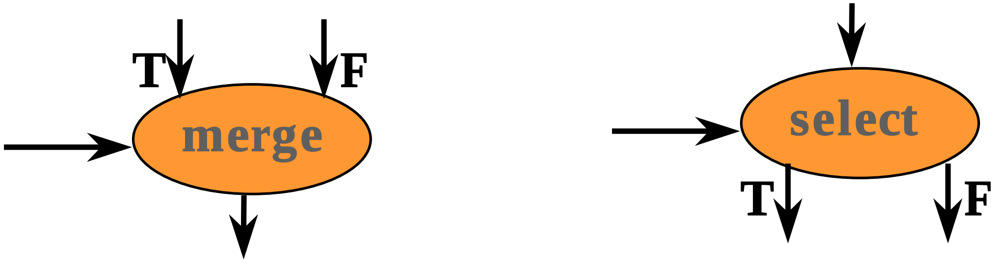


Examples of Dataflow actors

- SDF: Static Dataflow: fixed number of input and output tokens



- BDF: Boolean Dataflow control token determines number of consumed and produced tokens



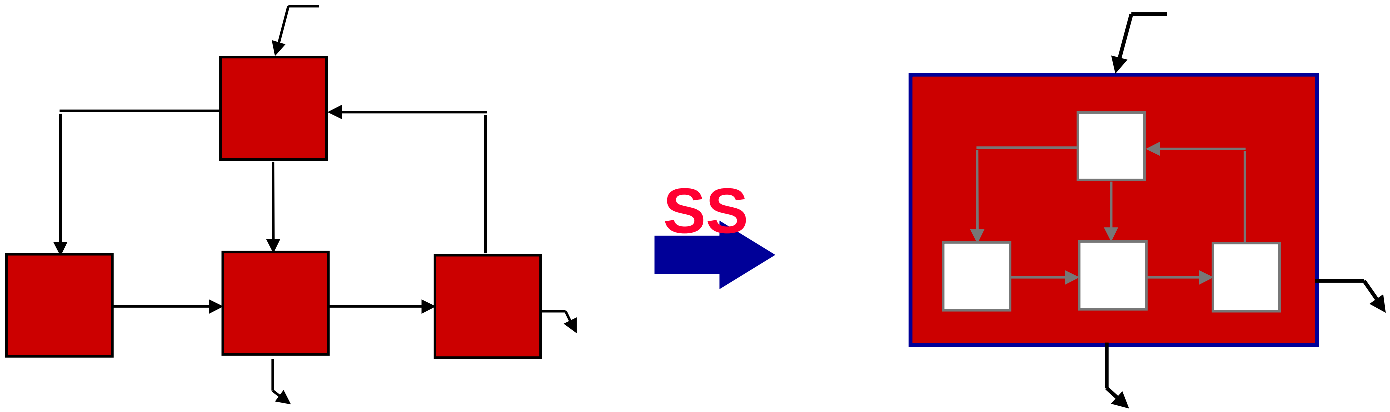
Outline

- Motivation
- Static Scheduling of dataflow networks
  - schedulability
  - code and data size optimization
- Quasi-Static Scheduling of process networks using Petri nets
  - Free Choice nets
  - Non-Free-Choice nets
- Conclusions

Static scheduling of DF

- Key property of DF networks: output sequences do not depend on *firing sequence* of actors
- SDF networks can be *statically scheduled* at compile-time
  - execute an actor when it is *known* to be fireable
  - no overhead due to sequencing of concurrency
  - static buffer sizing
- Different schedules yield different
  - code size
  - buffer size
  - pipeline utilization

Static Scheduling



- Sequentialize concurrent operations as much as possible
  - less communication overhead
    (run-time task generation)
  - better starting point for compilation
    (straight-line code from function blocks)

$\Rightarrow$ Must handle

  - multi-rate communication

Static scheduling of SDF

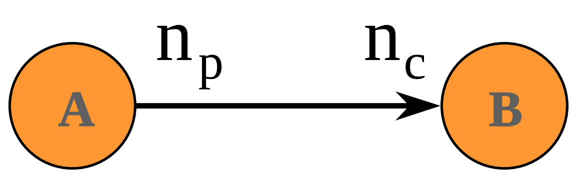
- Based only on *process graph* (no functionality)
- Network state: number of tokens in FIFOs
- Objective: find schedule that is *valid*, i.e.:
  - admissible

    (only fires actors when fireable)
  - periodic

    (brings network back to initial state firing each actor at least once)
- Optimize cost function over admissible schedules

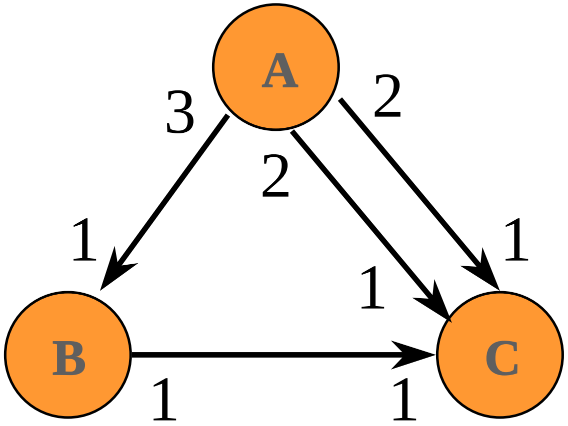
Balance equations

- Number of produced tokens must equal number of consumed tokens on every edge



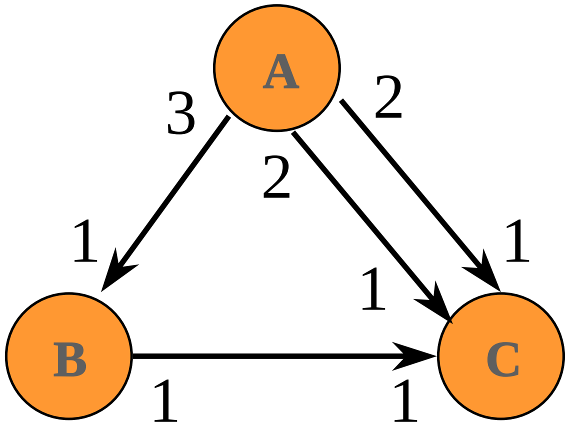
- Repetitions (or firing) vector v_S of schedule S: number of firings of each actor in S
- $v_S(A)\ n_p = v_S(B)\ n_c$
  must be satisfied for each edge

Balance equations



- Balance for each edge:
  - $3\ v_S(A) - v_S(B) = 0$
  - $v_S(B) - v_S(C) = 0$
  - $2\ v_S(A) - v_S(C) = 0$
  - $2\ v_S(A) - v_S(C) = 0$

Balance equations

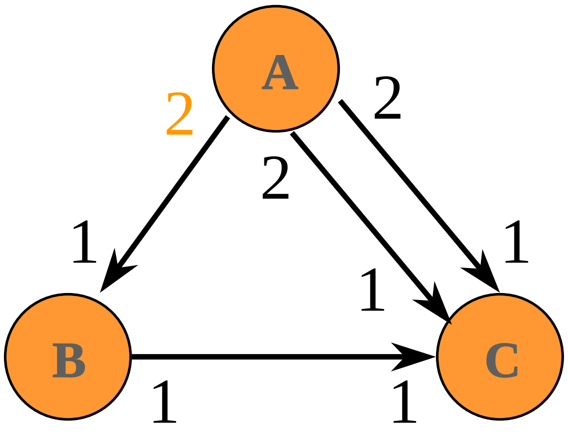


$$M = \begin{vmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{vmatrix}$$

- $M\ v_S = 0$
  iff S is periodic
- Full rank (as in this case)
  - no non-zero solution
  - no periodic schedule

(too many tokens accumulate on A->B or B->C)

Balance equations



$$M = \begin{vmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{vmatrix}$$

- Non-full rank
  - infinite solutions exist (linear space of dimension 1)
- Any multiple of $q = |1 \quad 2 \quad 2|^T$ satisfies the balance equations
- ABCBC and ABBCC are minimal valid schedules
- ABABBCBCCC is non-minimal valid schedule

Static SDF scheduling

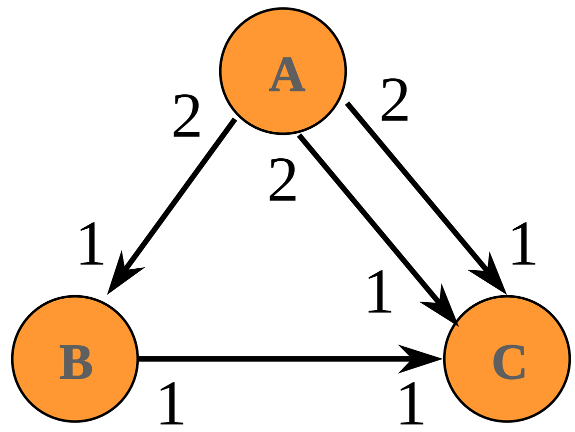
- Main SDF scheduling theorem (Lee ‘86):
  - A connected SDF graph with *n* actors has a periodic schedule iff its topology matrix M has rank *n-1*
  - If M has rank *n-1* then there exists a unique smallest integer solution q to

    $$M\ q = 0$$

From repetition vector to schedule

- Repeatedly schedule fireable actors up to number of times in repetition vector

$q = |1 \quad 2 \quad 2|^T$



- Can find either ABCBC or ABBCC
- If deadlock before original state, no valid schedule exists (Lee '86)

From schedule to implementation

- Static scheduling used for:
  - behavioral simulation of DF code generation for DSP
  - HW synthesis (Cathedral, Lager, …)
- Issues in code generation
  - execution speed (pipelining, vectorization)
  - code size minimization
  - data memory size minimization (allocation to FIFOs)
  - processor or functional unit allocation

Outline

- Motivation
- Static Scheduling of dataflow networks
  - schedulability
  - code and data size optimization
- Quasi-Static Scheduling of process networks using Petri nets
  - Free Choice nets
  - Non-Free-Choice nets
- Conclusions

Compilation optimization

- Assumption: *code stitching*
  (chaining custom code for each actor)
- More efficient than C compiler for DSP
- Comparable to hand-coding in some cases
- Explicit parallelism, no artificial control dependencies
- Main problem: memory and processor/FU allocation depends on scheduling, and vice-versa

Code size minimization

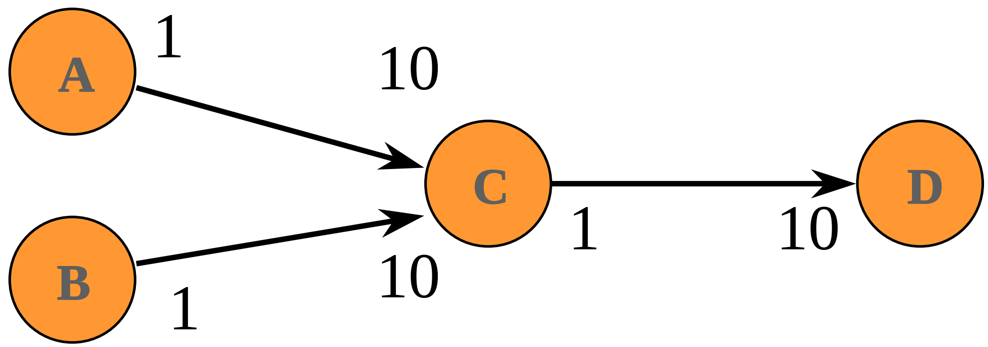
- Assumptions (based on DSP architecture):
  - subroutine calls expensive
  - fixed iteration loops are cheap ("zero-overhead loops")
- Global optimum: *single appearance schedule*

  e.g. ABCBC -> A (2BC), ABBCC -> A (2B) (2C)
  - may or may not exist for an SDF graph…
  - buffer minimization relative to single appearance schedules

    (Bhattacharyya '94, Lauwereins '96, Murthy '97)

Buffer size minimization

- Assumption: no buffer sharing
- Example:



$q = | 100 \;\; 100 \;\; 10 \;\; 1|^T$

- Valid SAS: (100 A) (100 B) (10 C) D
  - requires 210 units of buffer area
- Better (factored) SAS: (10 (10 A) (10 B) C) D
  - requires 30 units of buffer areas, but…
  - requires 21 loop initiations per period (instead of 3)

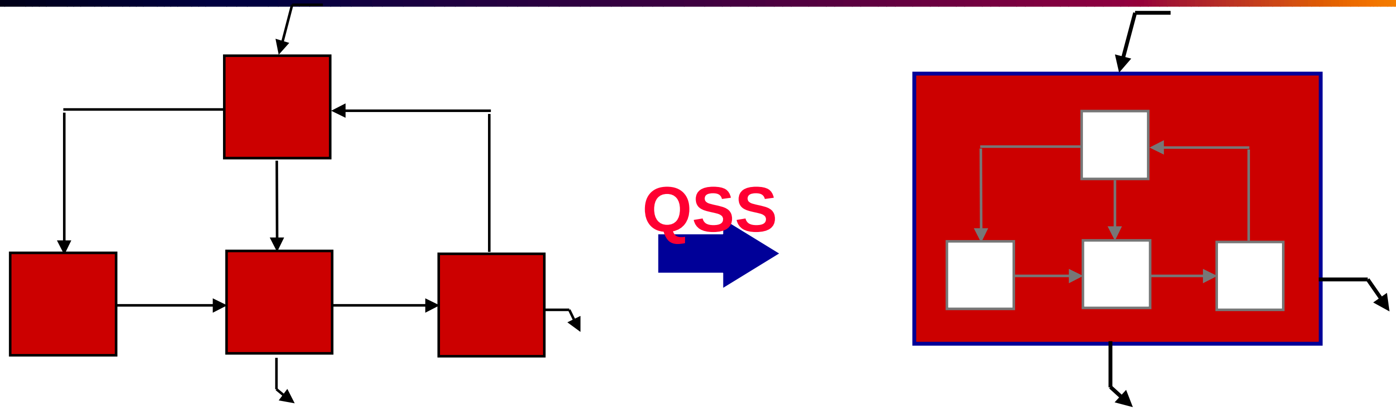
Scheduling more powerful DF

- SDF is limited in modeling power
- More general DF is too powerful
  - non-Static DF is Turing-complete (Buck ‘93)
  - bounded-memory scheduling is not always possible
- Boolean Data Flow: Quasi-Static Scheduling of special “patterns”
  - if-then-else, repeat-until, do-while
- Dynamic Data Flow: run-time scheduling
  - may run out of memory or deadlock at run time
- Kahn Process Networks: quasi-static scheduling using Petri nets
  - conservative: schedulable network may be declared unschedulable

Outline

- Motivation
- Static Scheduling of dataflow networks
  - schedulability
  - code and data size optimization
- Quasi-Static Scheduling of process networks using Petri nets
  - Free Choice nets
  - Non-Free-Choice nets
- Conclusions

Quasi-Static Scheduling



- Sequentialize concurrent operations as much as possible
  - less communication overhead

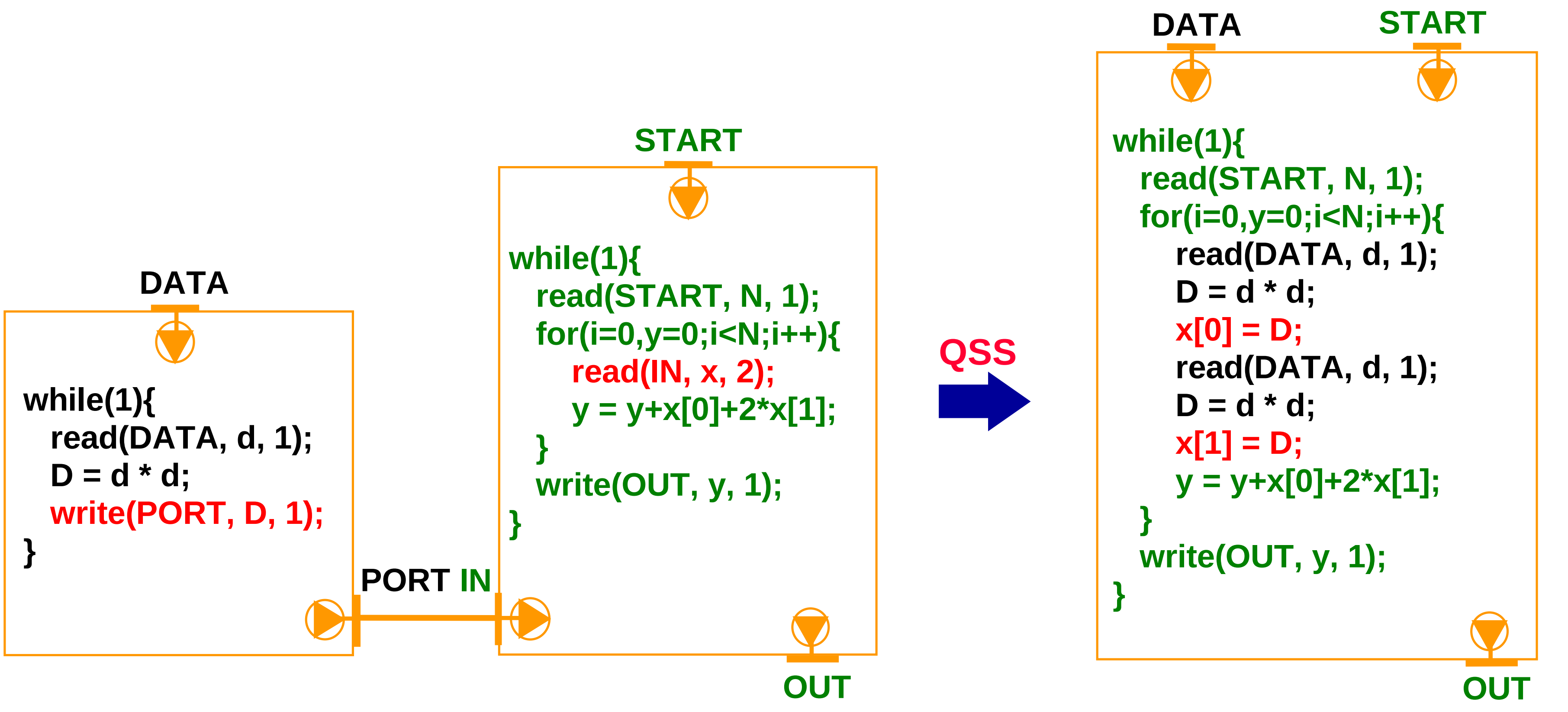
    (run-time task generation)
  - better starting point for compilation

    (straight-line code from function blocks)

⇒ Must handle

  - data-dependent control
  - multi-rate communication

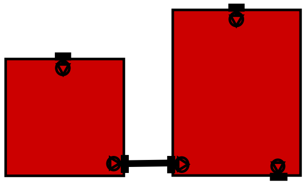
Quasi-Static Scheduling



The problem

- Given:

  a network of Kahn processes

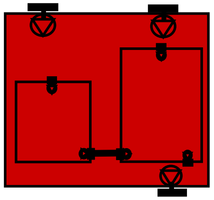


  – Kahn process: sequential function + ports

  – communication: port-based, point-to-point, uni-directional, multi-rate

- Find:

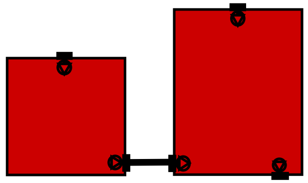
  a single task



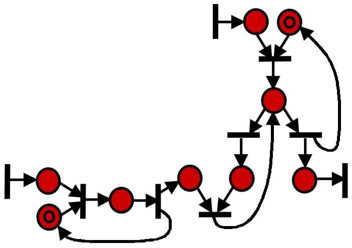
  – functionally equivalent to the original network (modulo concurrency)

The scheduling procedure

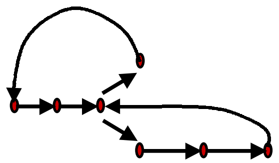
1. Specify a network of processes
   - process: C + communication operations
   - netlist: connection between ports



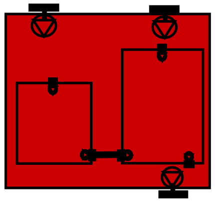
2. Translate to the computational model: Petri nets



3. Find a "schedule" on the Petri net



4. Translate the schedule to a task

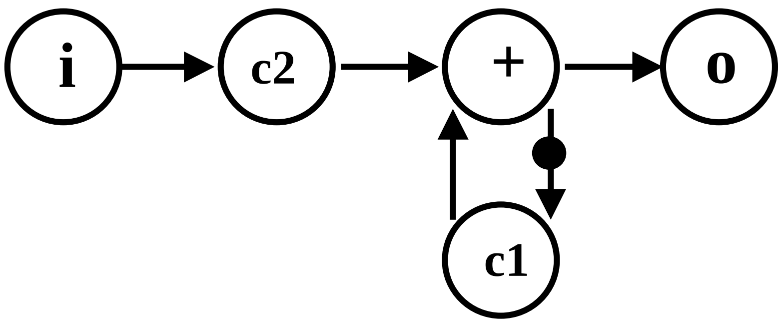


Scheduling Petri Nets

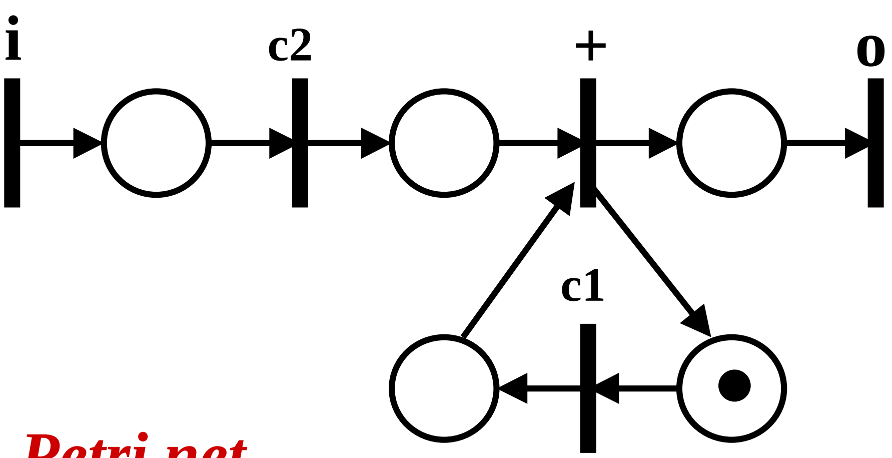
- Unified model for mixed control and dataflow
- Most properties are decidable
  (possibly scheduling is not ☹)
- A lot of theory is available

Infinite Impulse Response filter specification:

o[i] = c2 * i[i] + c1 * o[i-1]

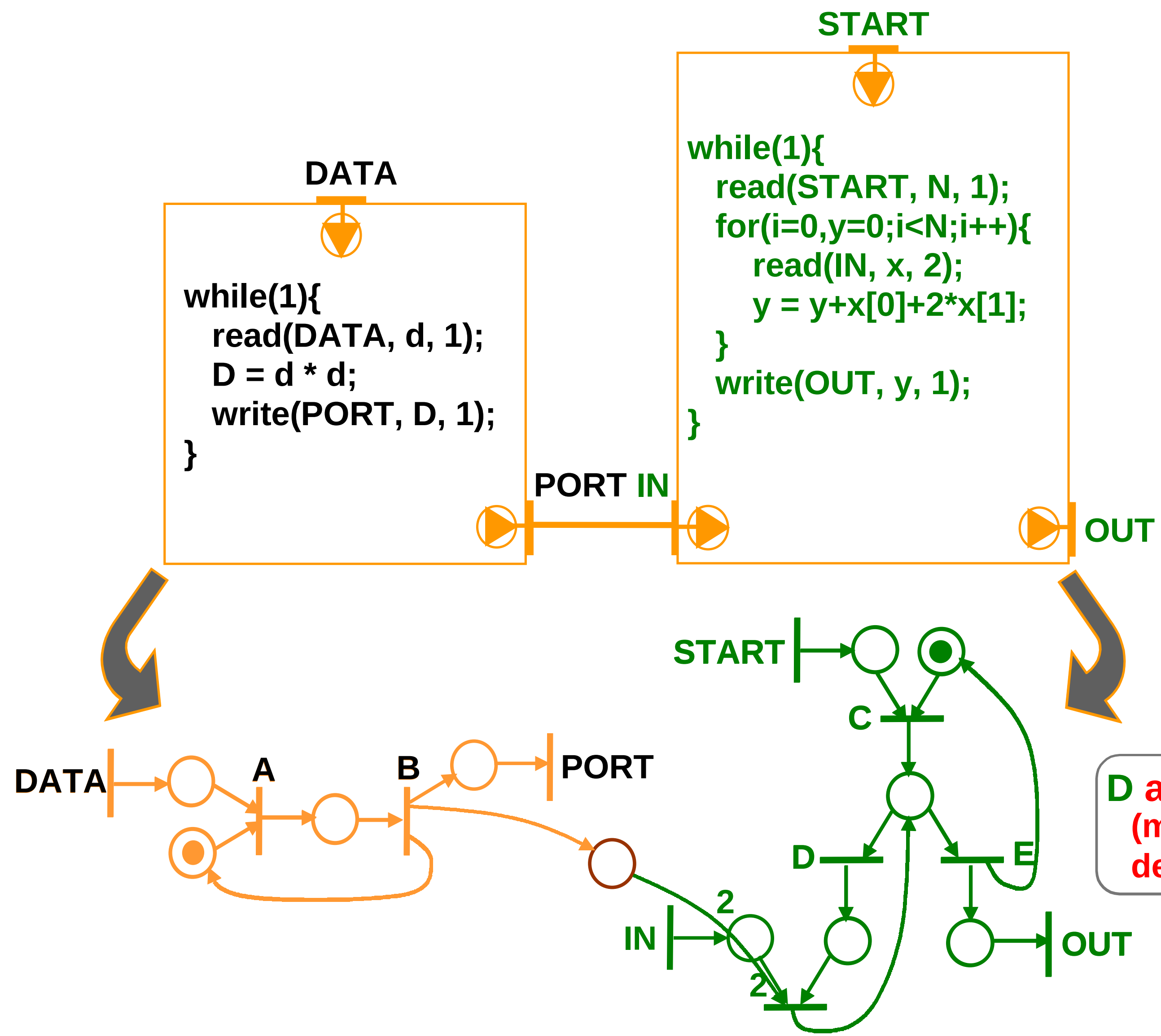


Static Data Flow network



Petri net

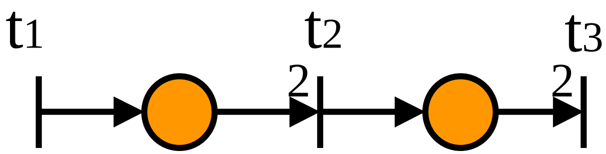
From process network to Petri Net



D and E in conflict
(modeling data-dependent control)

Bounded scheduling of Petri Net

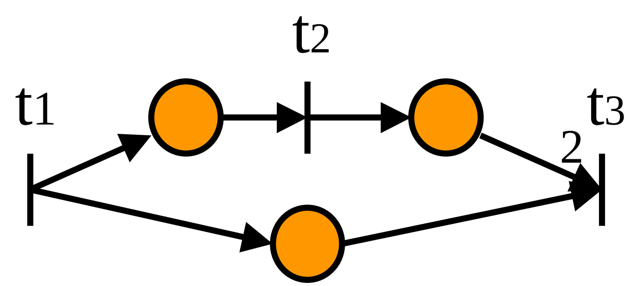
- A finite complete cycle is a finite sequence of transition firings that returns the net to its initial state:
  - infinite execution
  - bounded memory
- To find a finite complete cycle we must solve the ***balance (or characteristic) equation*** of the Petri net



$$D = \begin{bmatrix} 1 & 0 \\ -2 & 1 \\ 0 & -2 \end{bmatrix}$$

f * D = 0

f = (4,2,1)



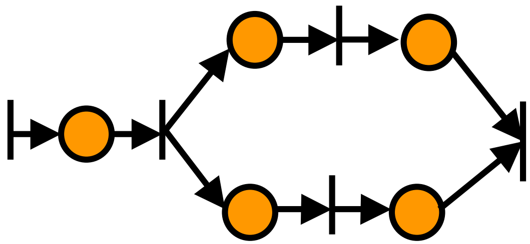
f * D = 0 has no solution

$\Rightarrow$ **No schedule**

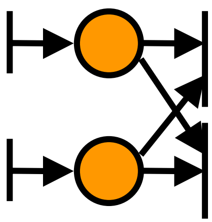
Outline

- Motivation
- Static Scheduling of dataflow networks
  - schedulability
  - code and data size optimization
- Quasi-Static Scheduling of process networks using Petri nets
  - Free Choice nets
  - Non-Free-Choice nets
- Conclusions

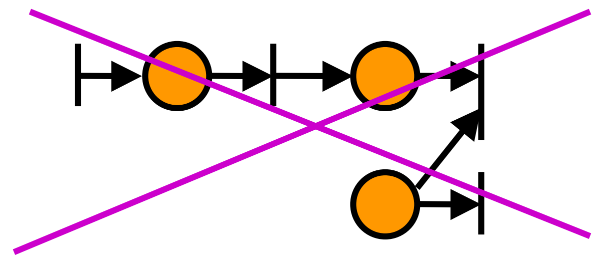
Free-Choice Petri Nets (FCPN)



Marked Graph (MG)



Free-Choice

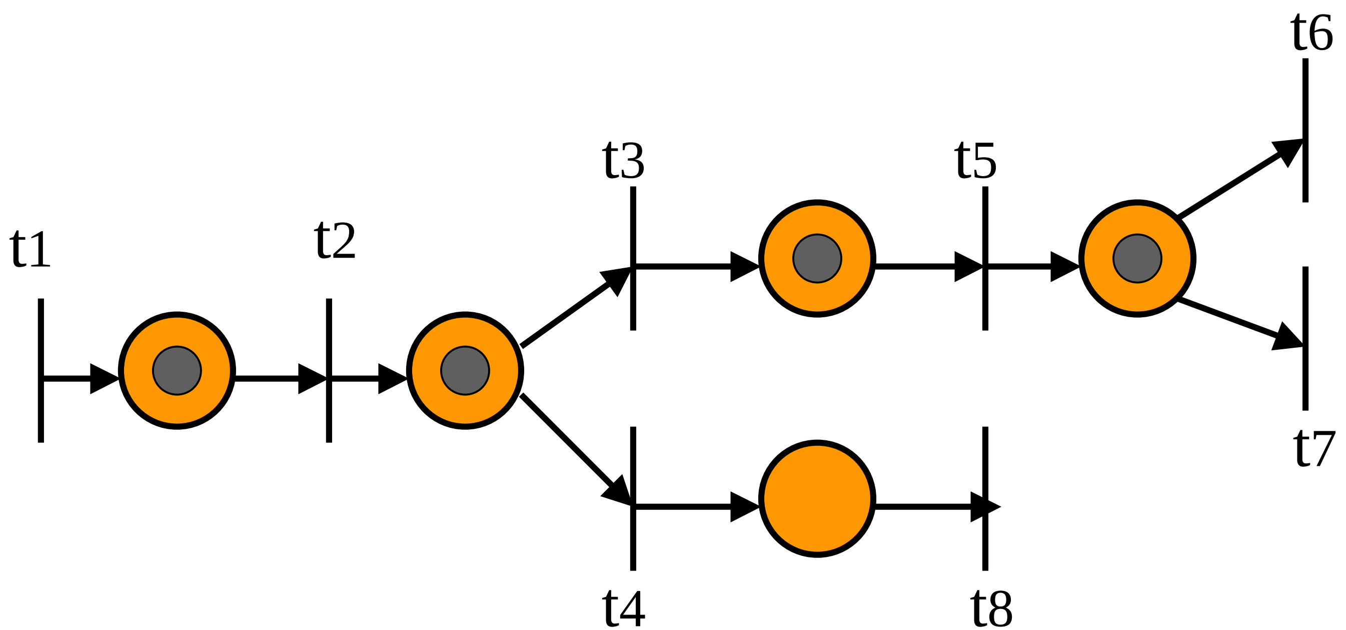


Confusion (not-Free-Choice)

- Free-Choice:
  - choice depends on token value (abstracted away) rather than arrival time
  - easy to analyze (using structural methods)

Bounded scheduling

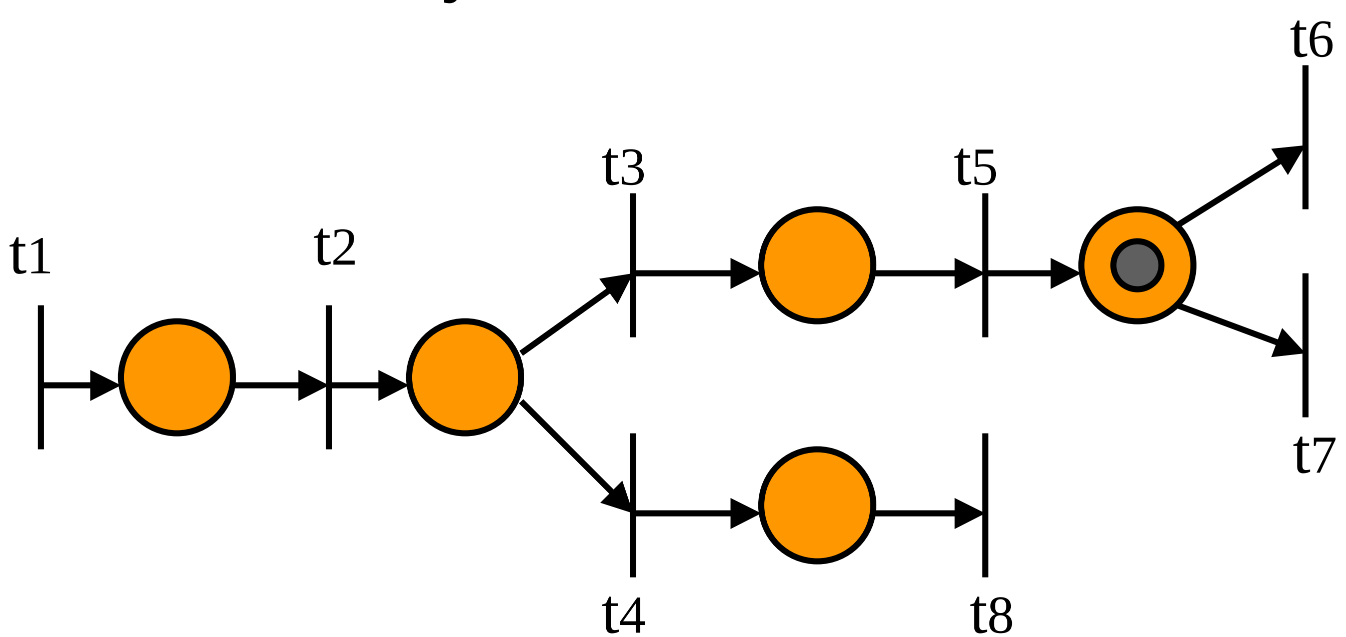
- Can the "adversary" ever force token overflow?



t1 t2 t3 t5 t6

Bounded scheduling

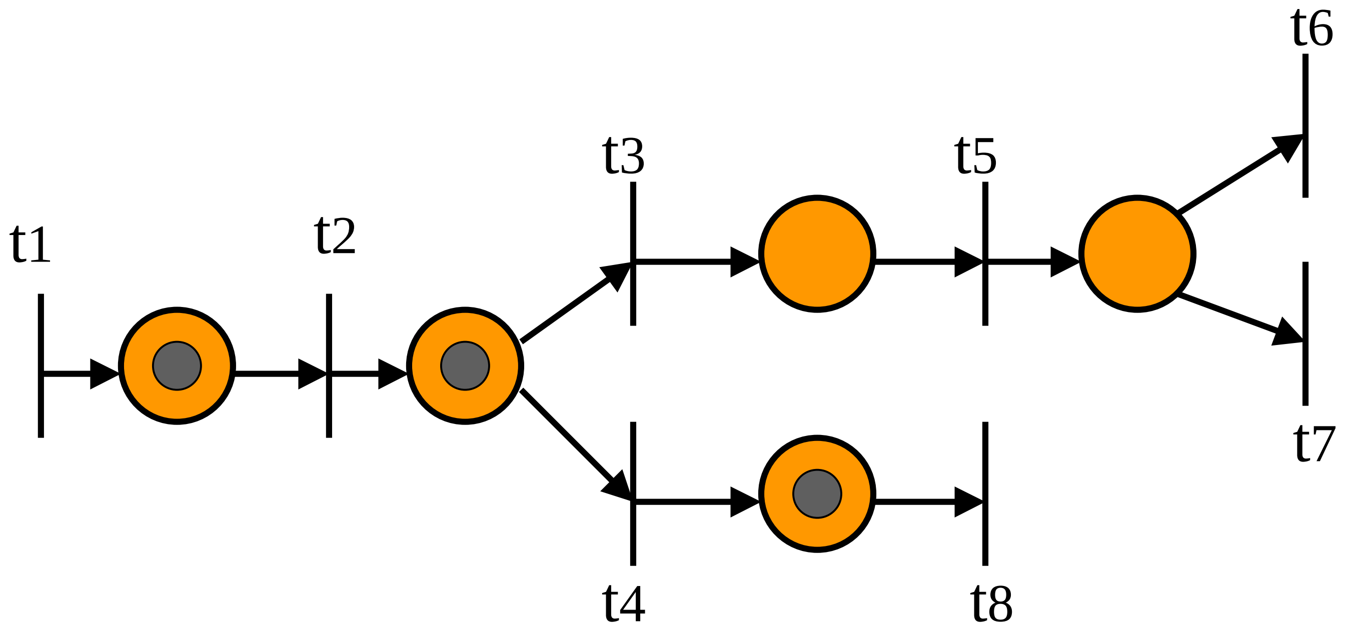
- Can the "adversary" ever force token overflow?



t1 t2 t3 t5 t7

Bounded scheduling

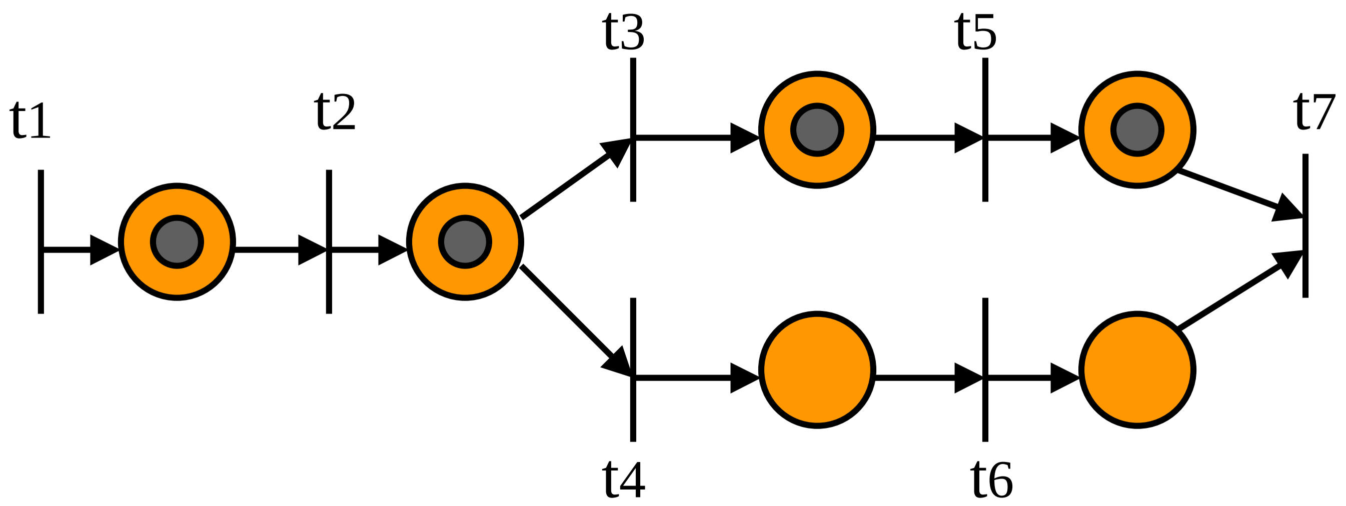
- Can the "adversary" ever force token overflow?



t1 t2 t4 t8

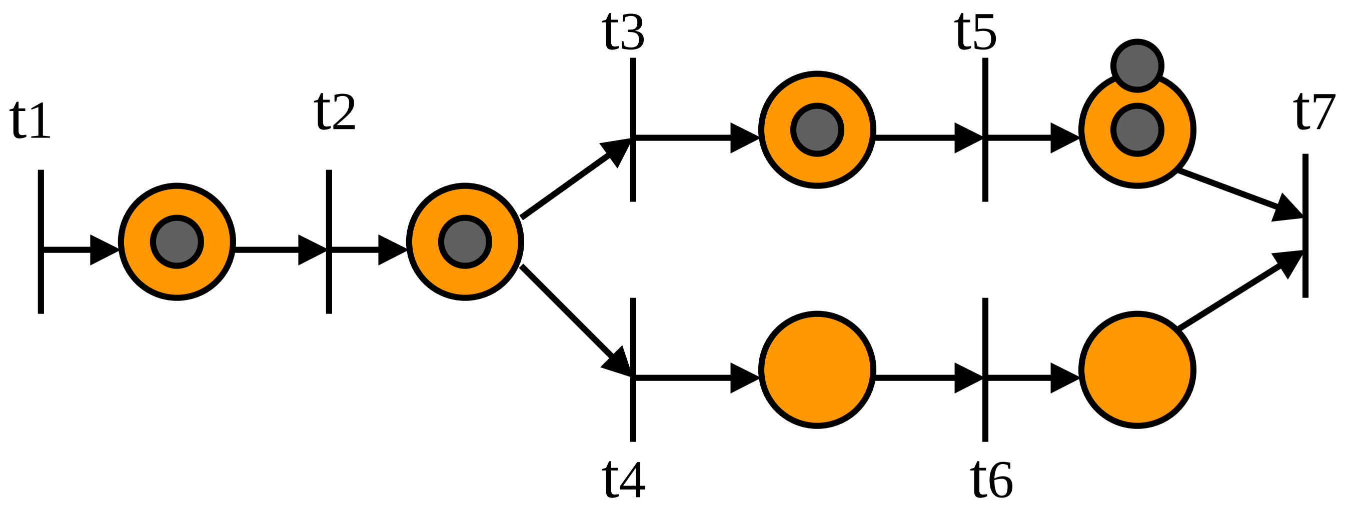
Bounded scheduling

- Can the "adversary" ever force token overflow?



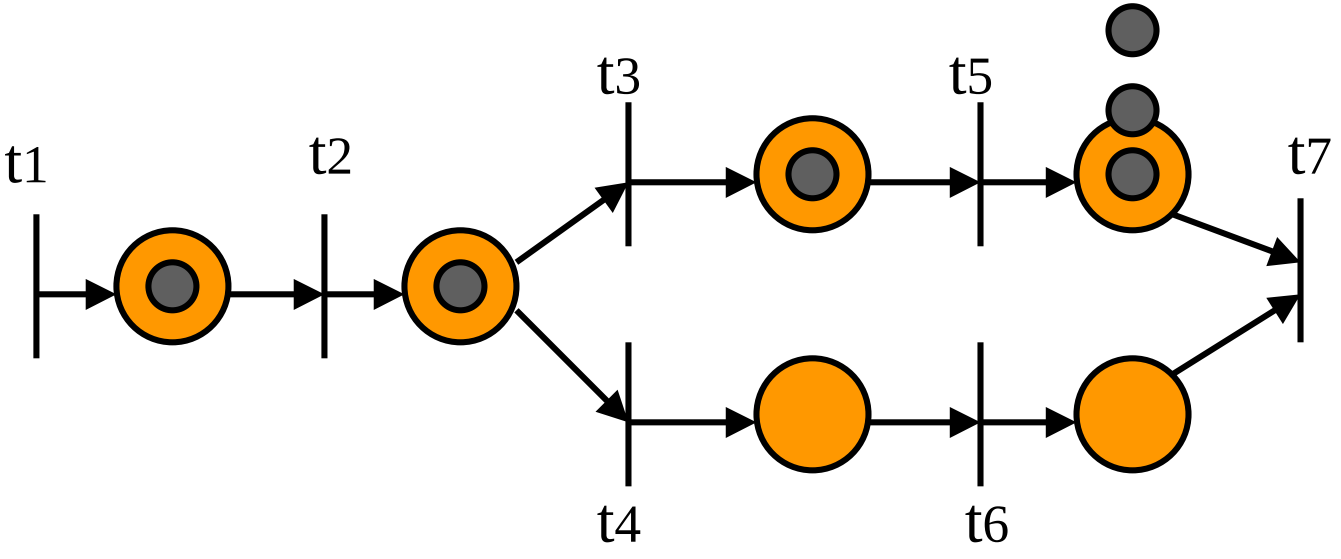
Bounded scheduling

- Can the “adversary” ever force token overflow?



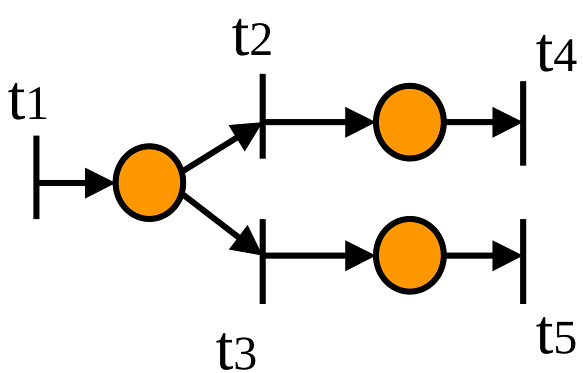
Bounded scheduling

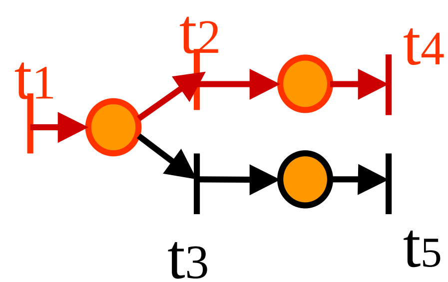
- Can the “adversary” ever force token overflow?



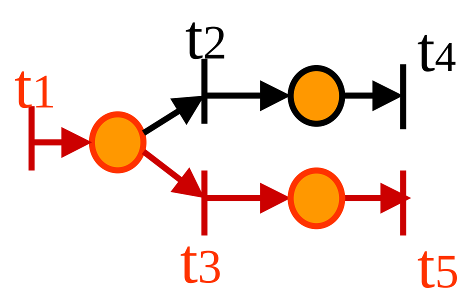
Schedulability of an FCPN

- Valid schedule Σ
  - is a set of finite firing sequences that return the net to its initial state
  - contains one firing sequence for every combination of outcomes of the free choices





(t_1 t_2 t_4)



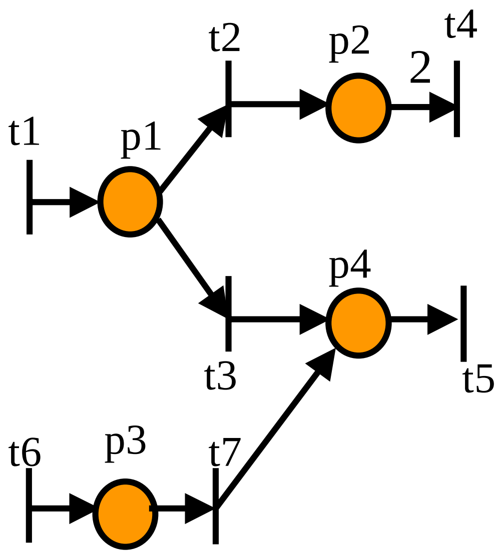
(t_1 t_3 t_5)

Σ={(t_1 t_2 t_4),(t_1 t_3 t_5)} → **Schedulable**

How to check schedulability

- Basic intuition: every resolution of data-dependent choices must be schedulable
- Algorithm:
  - Decompose the given Free-Choice Petri Net into as many Conflict-Free components (balance equation solutions) as the number of possible resolutions of the non-deterministic choices.
  - Check if every component is statically schedulable
  - Derive a valid schedule, i.e. a set containing one static schedule for each component
- Natural extension (with multiple balance equations) of SDF scheduling
- Still decidable

From schedule to C code



Σ={(t1 t2 t1 t2 t4 t6 t7 t5)
(t1 t3 t5 t6 t7 t5)}

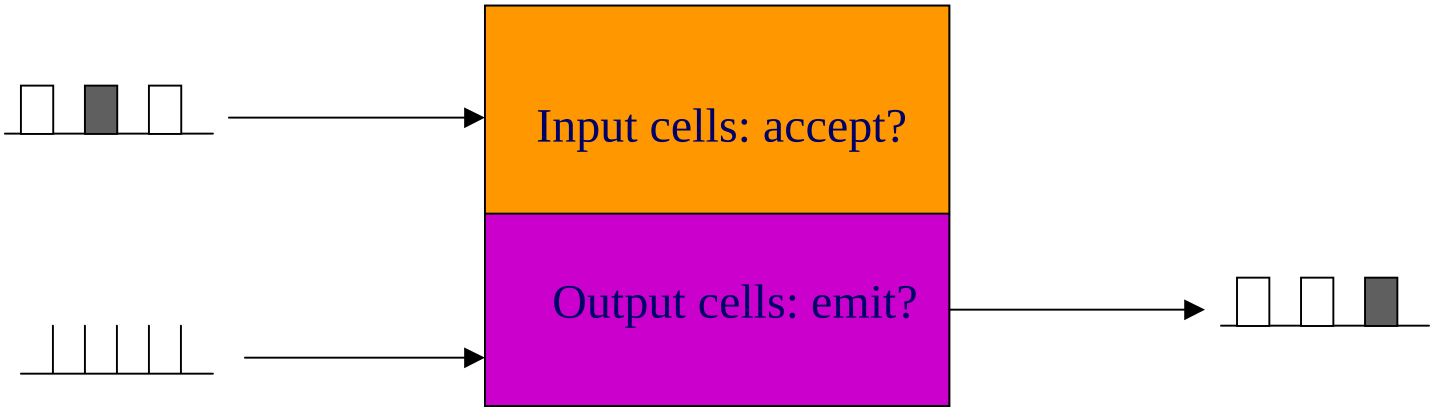
Task 1:

```
{   t1;
    if (p1) {
        t2;
        count(p2)++;
        if (count(p2) = 2) {
            t4;
            count(p2) = count(p2) - 2;
        }
    }
    else{
        t3;
        t5;
    }
}
```

Task 2:

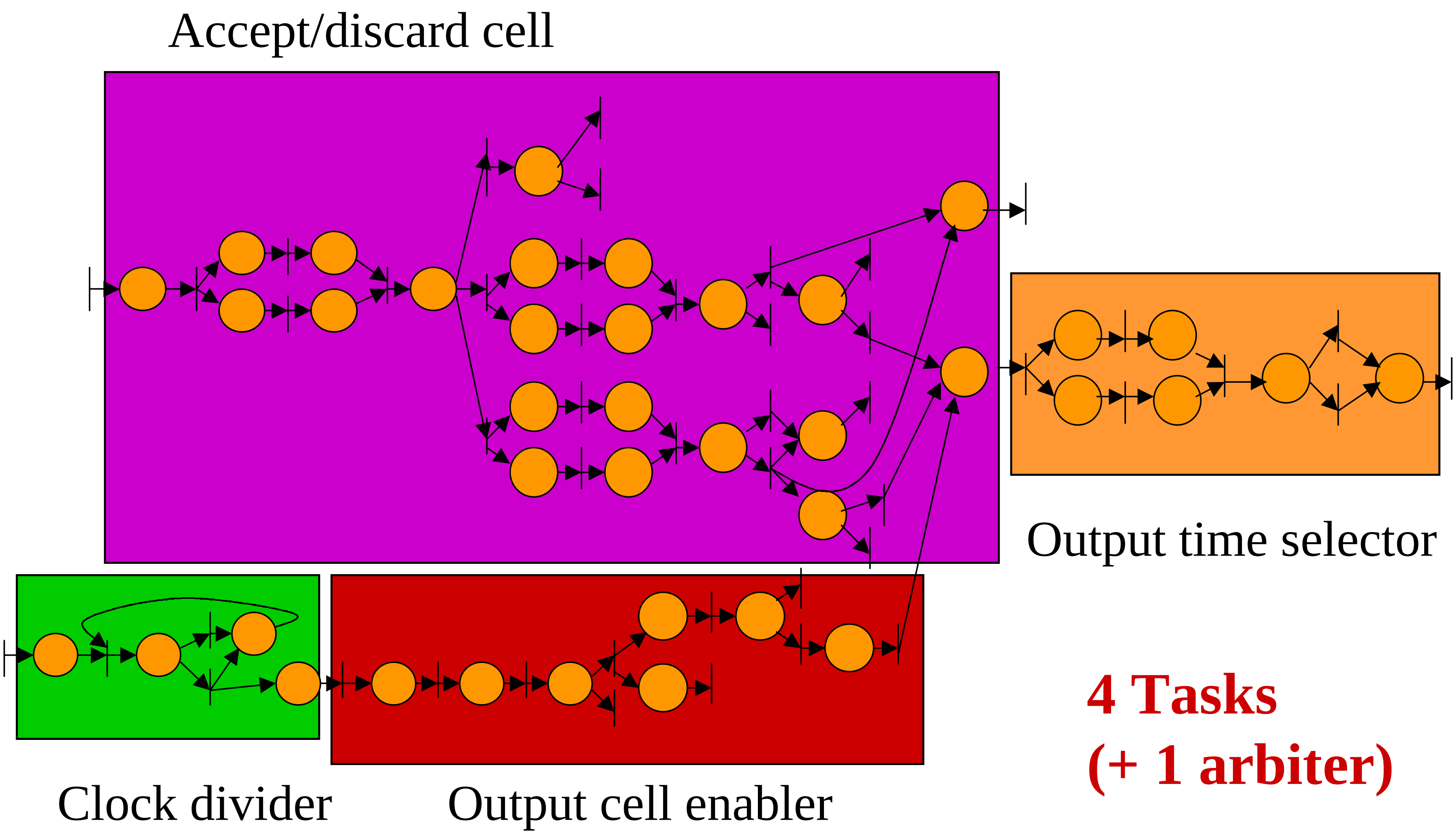
```
{   t6;
    t7;
    t5;
}
```

Application example: ATM Switch



- No static schedule due to:
  - Inputs with independent rates (need Real-Time dynamic scheduling)
  - Data-dependent control (can use Quasi-Static Scheduling)

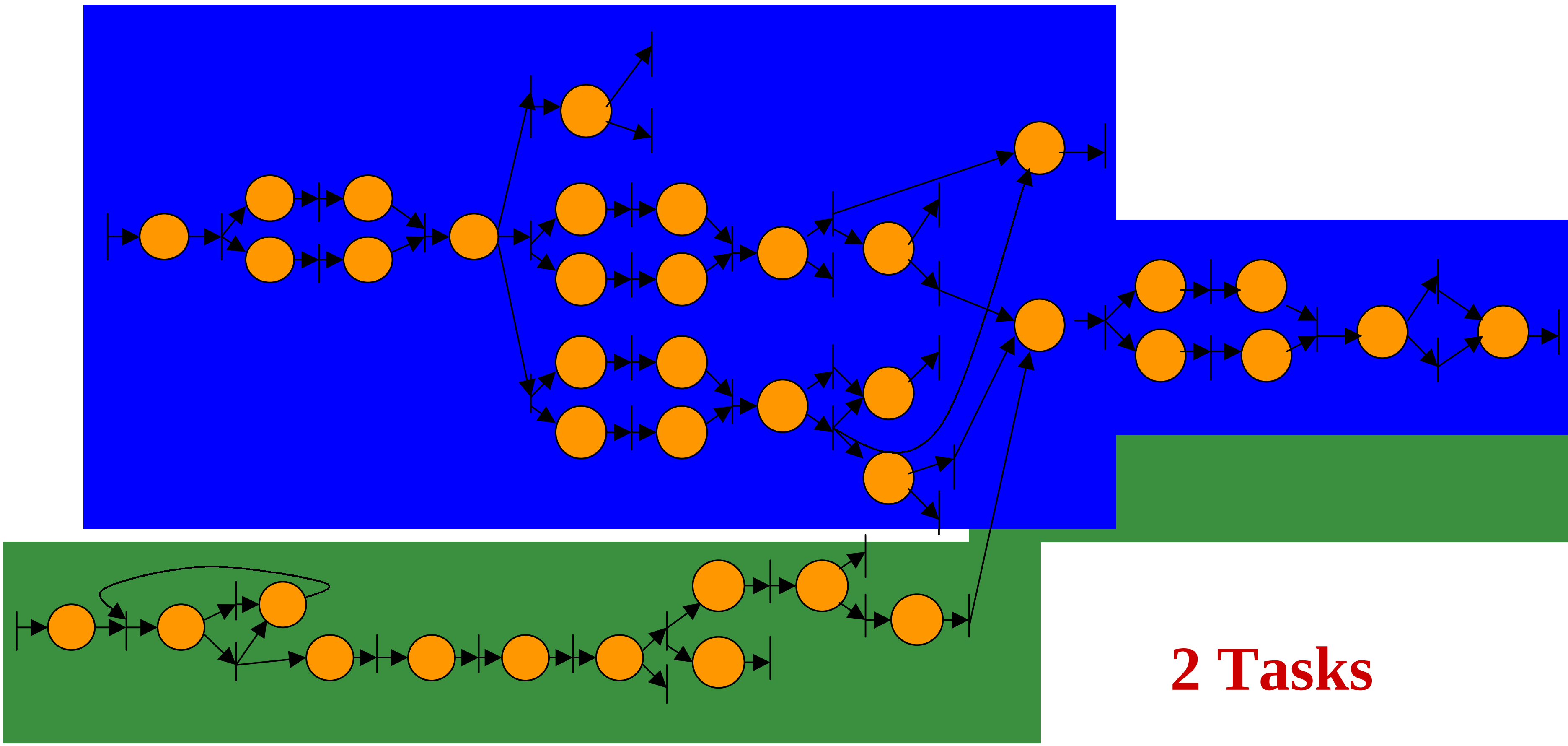
Functional Decomposition



4 Tasks
(+ 1 arbiter)

Minimal (QSS) Decomposition

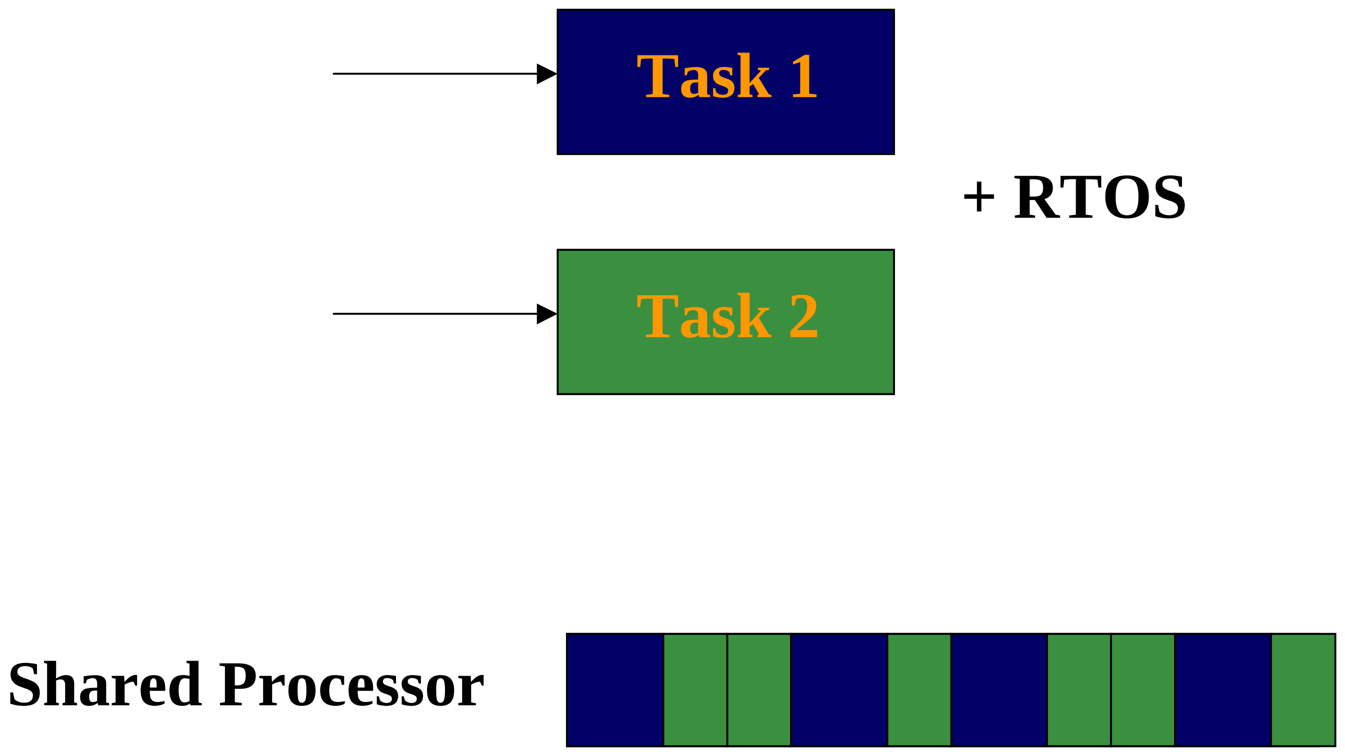
Input cell processing



Output cell processing

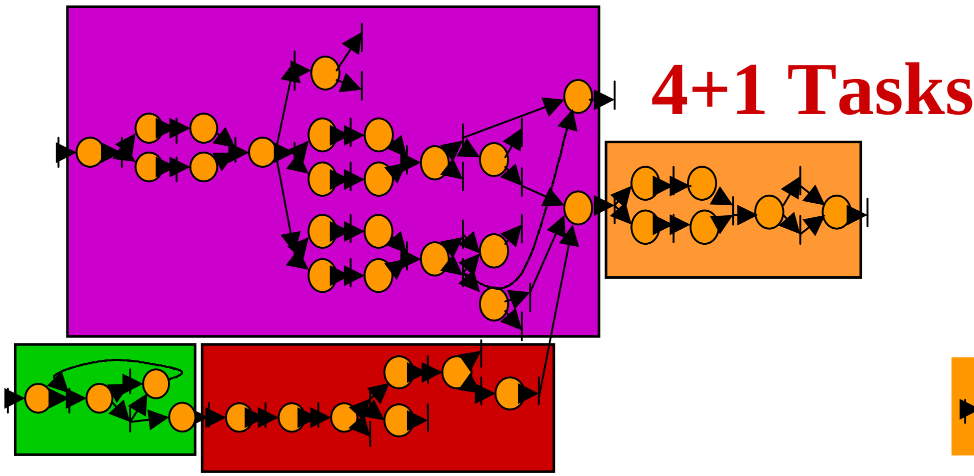
2 Tasks

Real-time scheduling of tasks

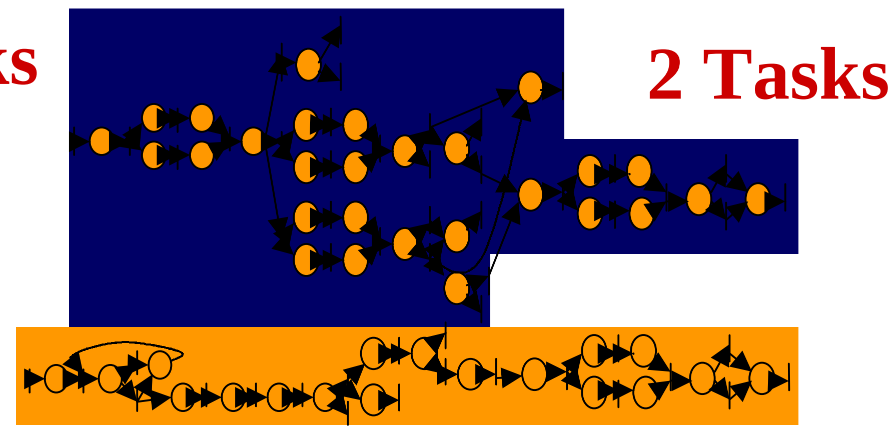


ATM: experimental results

Functional partitioning



QSS



Sw Implementation	QSS	Functional partitioning
Number of tasks	2	5
Lines of C code	1664	2187
Clock cycles	197,526	249,726

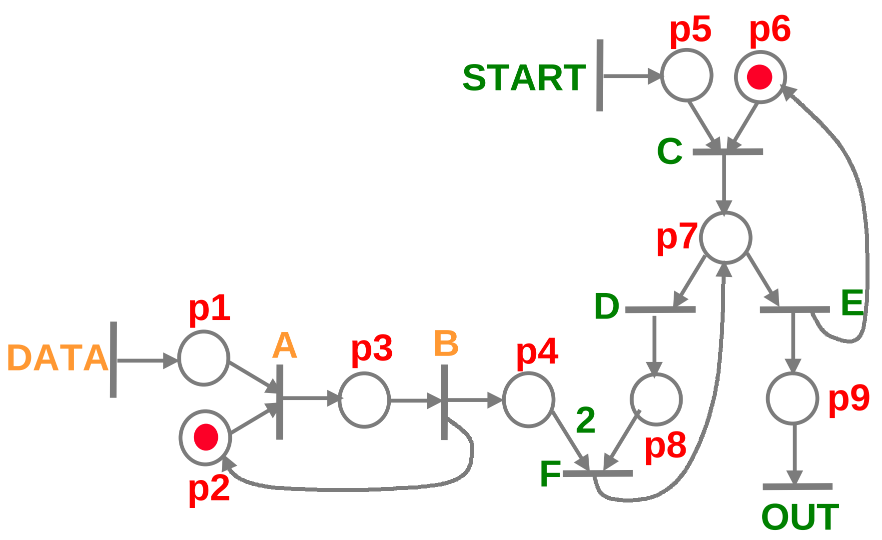
Outline

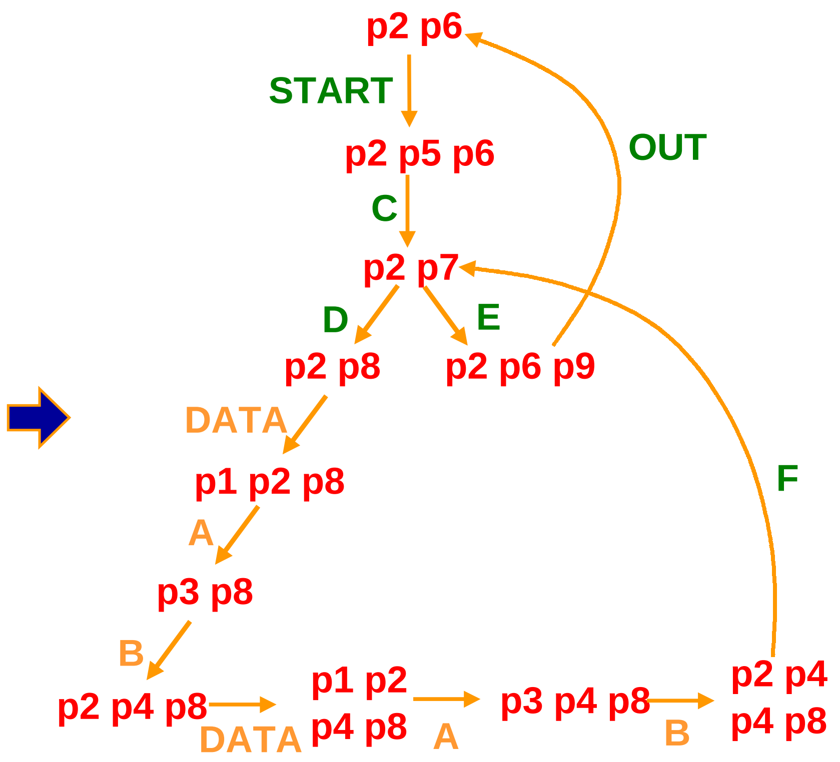
- Motivation
- Static Scheduling of dataflow networks
  - schedulability
  - code and data size optimization
- Quasi-Static Scheduling of process networks using Petri nets
  - Free Choice nets
  - Non-Free-Choice nets
- Conclusions

Extension beyond FCPNs

- Schedulability of FCPNs is decidable
- Algorithm may be exponential due to many components
- What if the resulting PN is non-free choice? (synchronization-dependent control)
- What if the PN is not schedulable for all choice resolutions?
(correlation between choices)

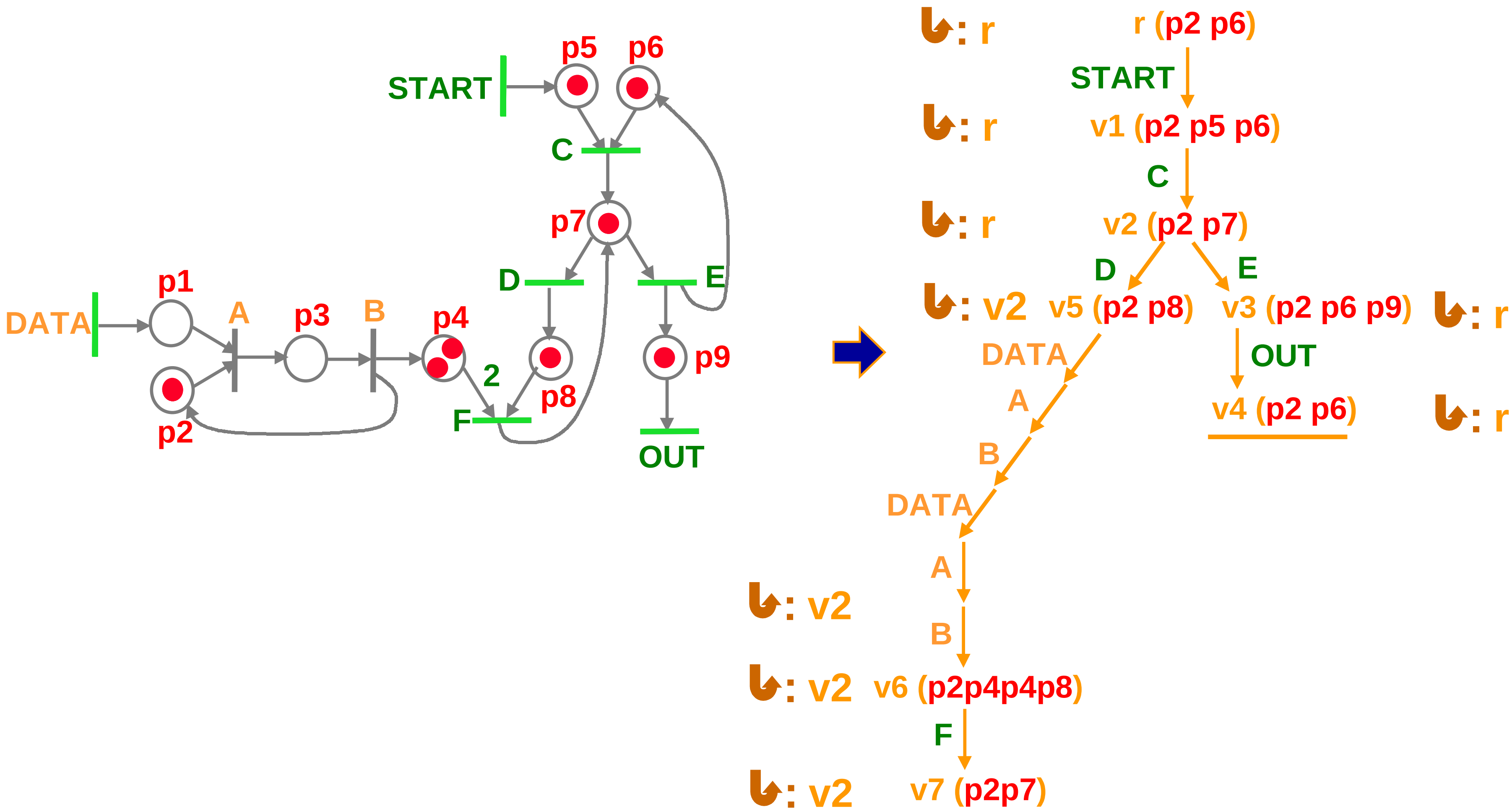
Finding a Schedule on the Petri Net





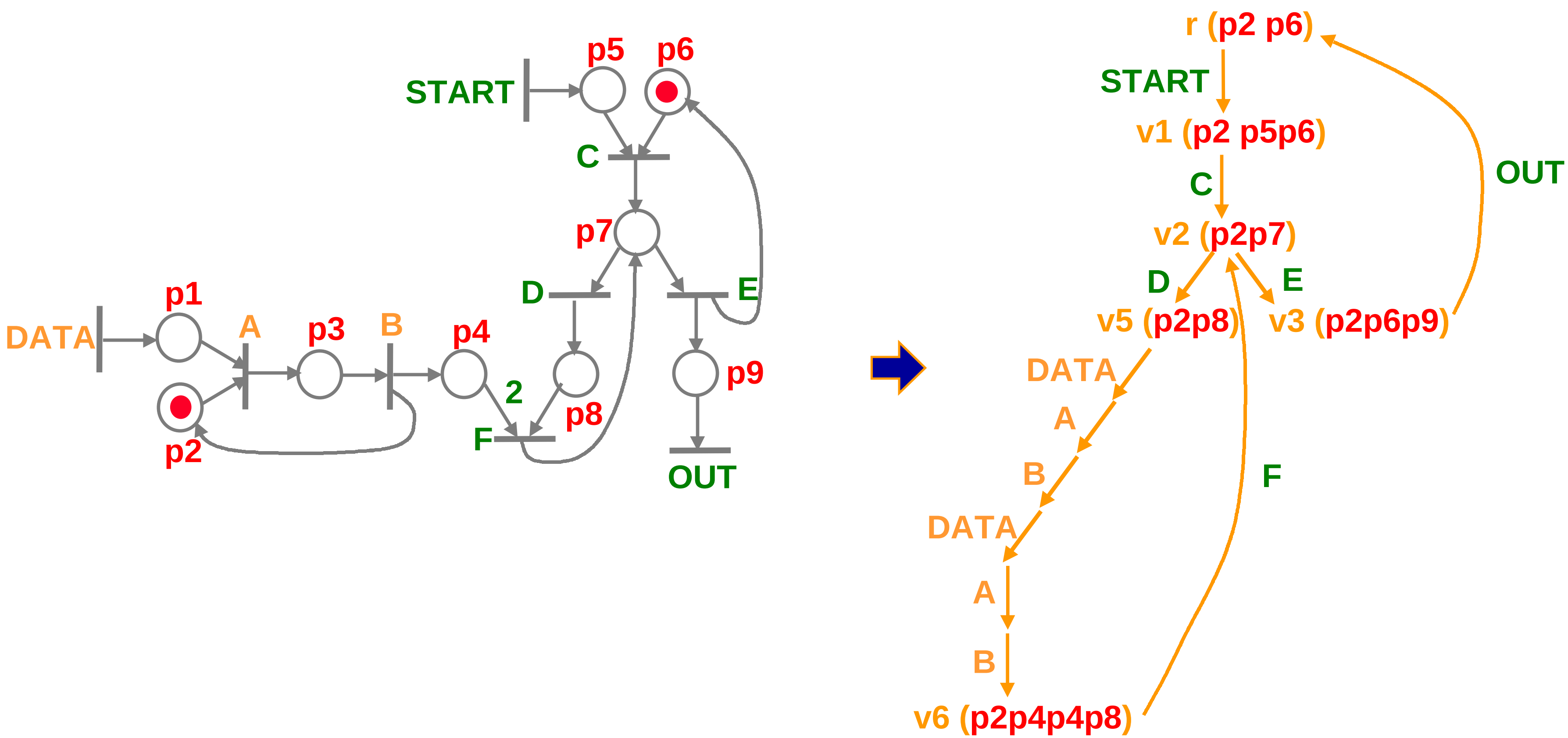
- Distinguished node **r** (p2 p6 in this case) associated with initial marking
- All and only transitions in conflict from each node
- A path to node **r** from each node

Finding a Schedule on the Petri Net



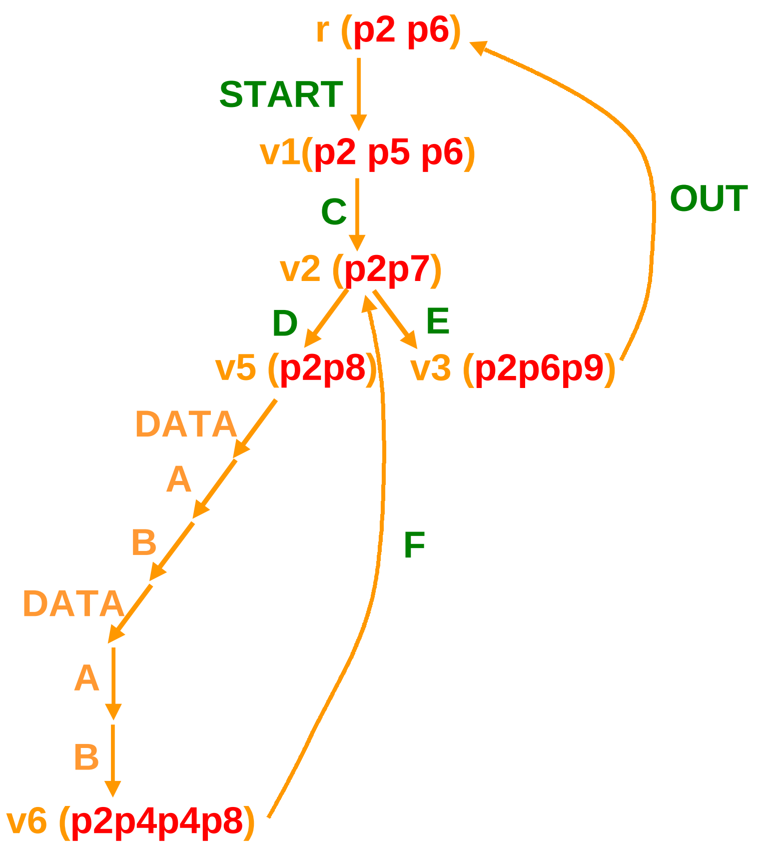
: the node at which a cycle was found.

Finding a Schedule on the Petri Net



- Choose a balance equation solution using a heuristic, and use it as much as possible
- Natural extension of FCPN (and SDF) scheduling

From schedule to C code



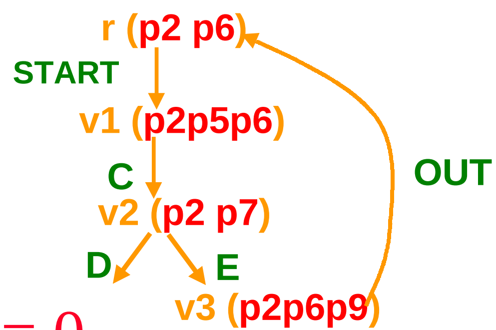
DATA START

```
Start:  read(START, N, 1); i=0; y=0;
DE: if(i < N){
        read(DATA, d, 1); D = d*d;
        x[0] = D;
        read(DATA, d, 1); D = d*d;
        x[1] = D;
        y=y+x[0]+2*x[1]; i++; goto DE;
    }  else{ write(OUT, y, 1); goto Start; }
```

OUT

Improving Efficiency

- Which transition should be chosen at each node?



  - Find sequences of transitions to create cycles.

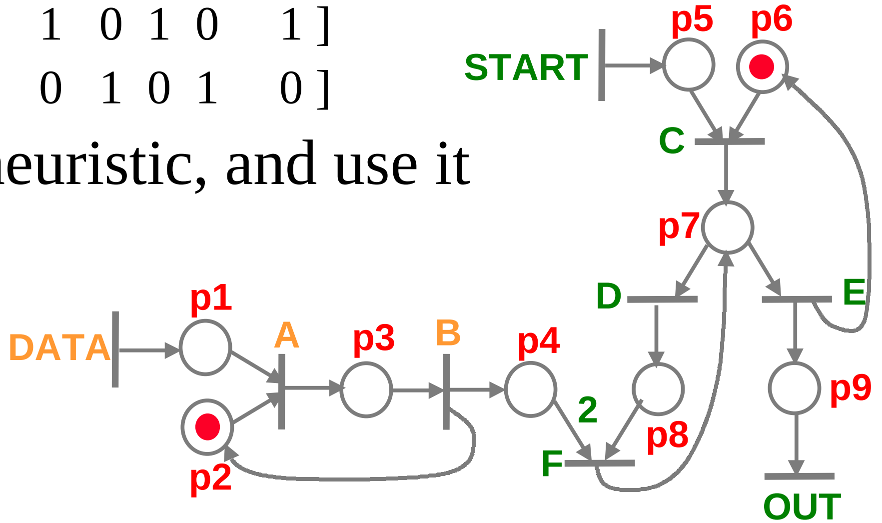
    T-invariant: a basis of the linear system $A\,x = 0$

    A[i, j]: # of tokens produced to the i-th place by the j-th transition.

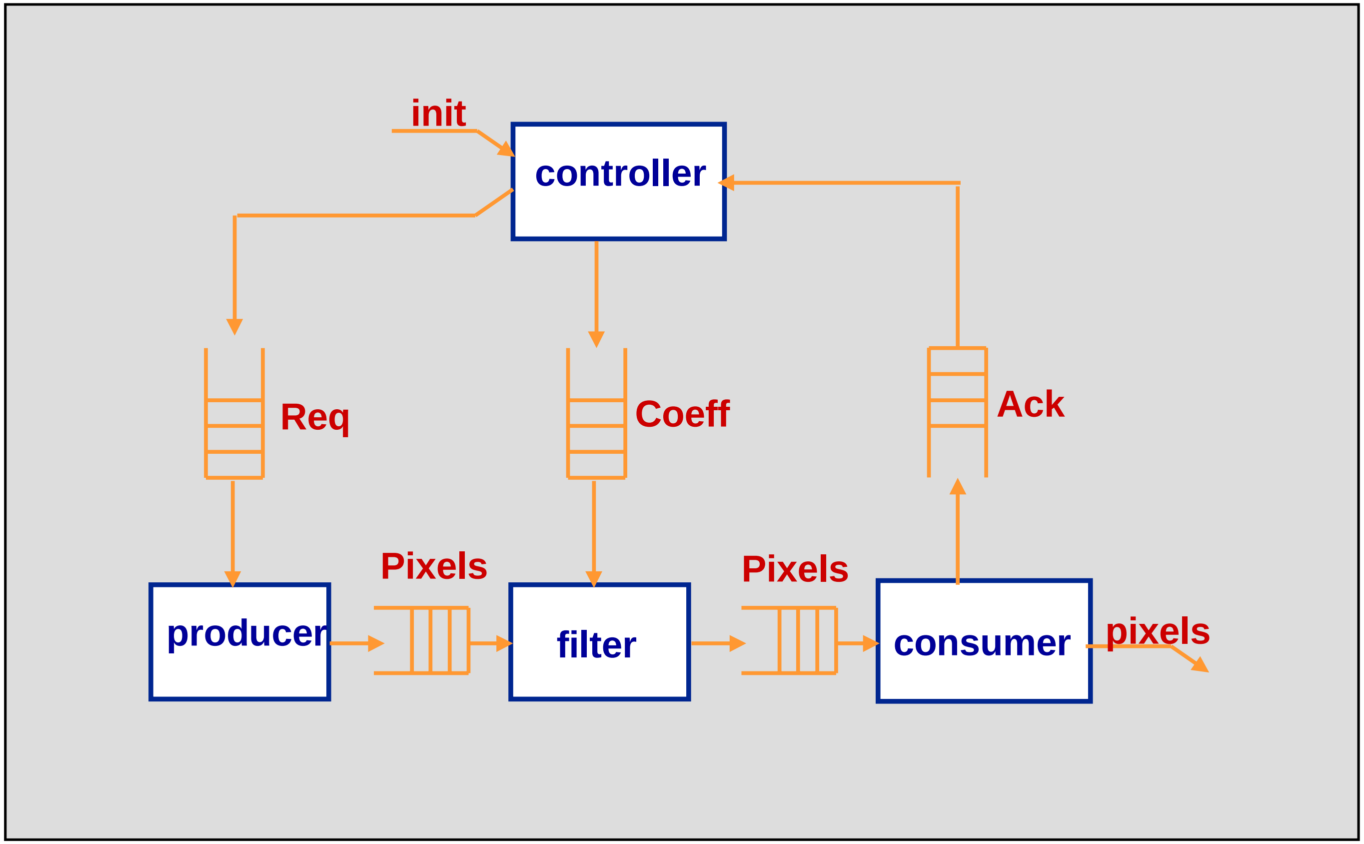
T-invariants:

	DATA	A	B	START	C	D	E	F	OUT	
[	0	0	0	1	1	0	1	0	1	]
[	2	2	2	0	0	1	0	1	0	]

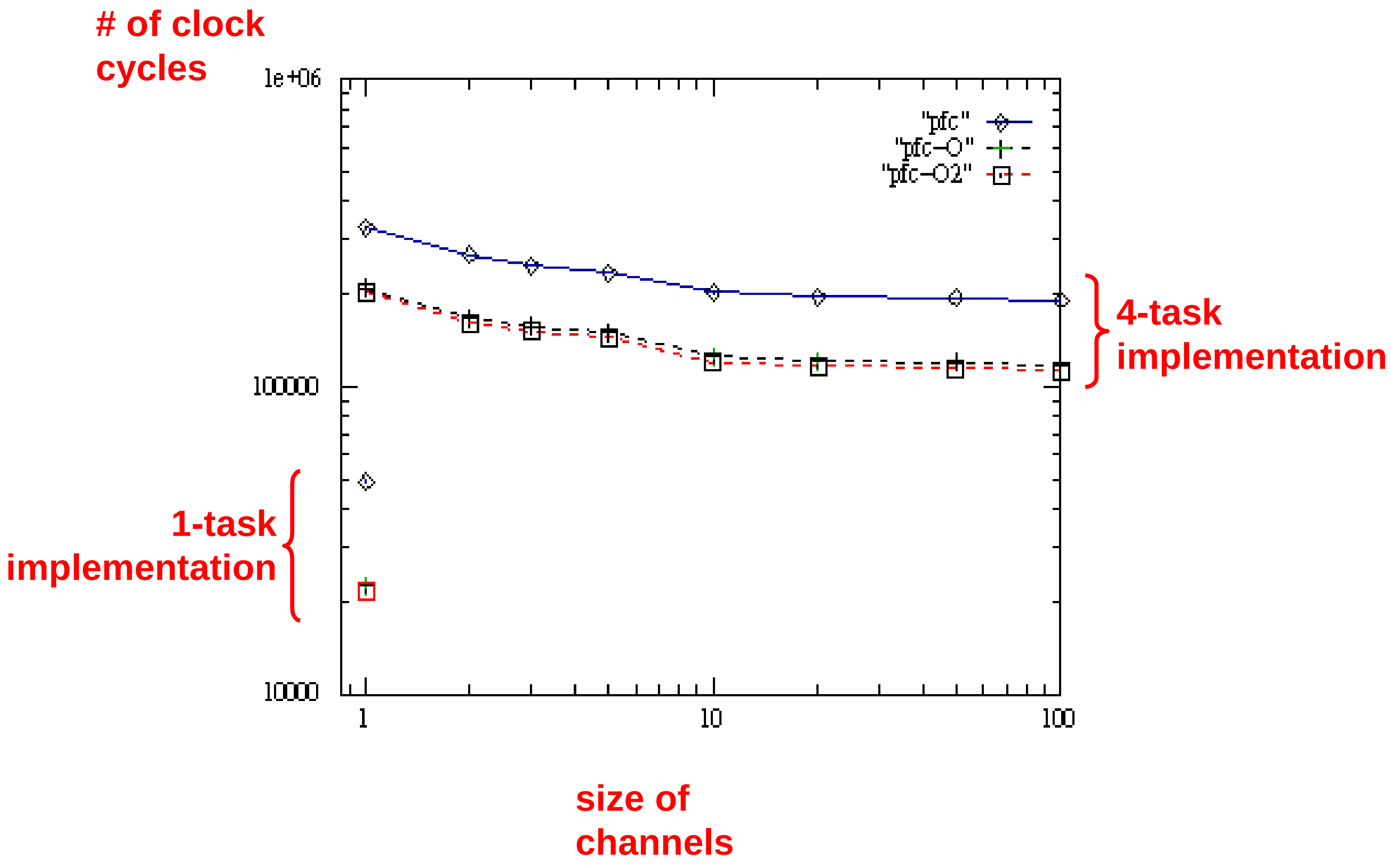
  - Choose a T-invariant using a heuristic, and use it as much as possible.



Producer-Filter-Consumer Example



Experimental Results



(Quasi) Static Scheduling approaches

- Lee *et al*. ‘86: Static Data Flow: cannot specify data-dependent control
- Buck *et al*. ‘94: Boolean Data Flow: undecidable schedulability check, heuristic pattern-based algorithm
- Thoen *et al*. ‘99: Event graph: no schedulability check, no task minimization
- Lin ‘97: Safe Petri Net: no schedulability check, single-rate, reachability-based algorithm
- Thiele *et al*. ‘99: Bounded Petri Net: partial schedulability check, reachability-based algorithm
- Cortadella *et al*. ‘00: General Petri Net: maybe undecidable schedulability check, balance equation-based algorithm

Conclusions

- Static and Quasi-Static Scheduling minimize run-time overhead by automatic partitioning of the system functions into a minimal number of concurrent tasks
  - sequentialize concurrent operations
  - data-dependent controls, multi-rate operations
  - technology-independent preprocessor
- Open issues:
  - correlated data-dependent controls
  - heuristic evaluation of different schedules
  - time-constrained scheduling
  - what about multiple processors? ☺