Hardware and Petri nets

Alex Yakovlev

Univ. Newcastle upon Tyne

Advanced Course on Petri nets,

Eichstätt, 24-26 Sept, 2003

Contents and schedule of lectures

- Introduction (Wednesday,10:30-10:45)
- Hardware modelling with Petri nets (Wednesday,10:45-12:00)
- Circuit Synthesis (Thursday 10:30-12:00):
  - Direct synthesis of Petri nets (Thursday, 10:30-11:15)
  - Logic synthesis from STGs (Thursday, 11:15-12:00)
- Analysis and verification (Friday, 10:30-11:15)
- Performance analysis (Friday, 11:15-12:00)

Main bib references

- A.V. Yakovlev, A.M.Koelmans. Petri nets and digital hardware design, Lectures on Petri nets II: Applications, Advances in Petri Nets, LNCS vol. 1492, Springer 1998, pp. 154-236
- A. Kondratyev, M. Kishinevsky, A. Taubin, J. Cortadella, L. Lavagno. The use of Petri nets for the design and verification of asynchronous circuits and systems, Jour. Cir.,Syst. And Comp., vol.8, no.1, Feb 1998, pp. 67-118.
- Hardware Design and Petri Nets (Editors: A. Yakovlev, L. Gomes, L. Lavagno), Kluwer Academic Publishers, March 2000, ISBN 0-7923-7791-5, 344 pp
- J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev, Logic Synthesis of Asynchronous Controllers and Interfaces, Springer, March 2002, ISBN3-540-43152-7
- Hardware Design and Concurrency, Advances in Petri nets, LNCS vol. 2549, Springer, ISBN 3-540-00199-9, 345pp.

Hardware and Petri Nets: Introduction

Alex Yakovlev

Univ. Newcastle upon Tyne

Advanced Course on Petri nets,

Eichstätt, 24-26 Sept, 2003

Introduction. Outline

- **Role of Hardware in modern systems**
- **Role of Hardware design tools**
- **Role of a modeling language**
- **Why Petri nets are good for Hardware Design**
- **History of “relationship” between Hardware Design and Petri nets**
- **Asynchronous Circuit Design**

Role of Hardware in modern systems

- Technology allows putting 1 billion transistors on a chip
- System on Chip is a reality – 1 billion operations per second
- Hardware and software designs are no longer separate
- Hardware becomes distributed, asynchronous and concurrent

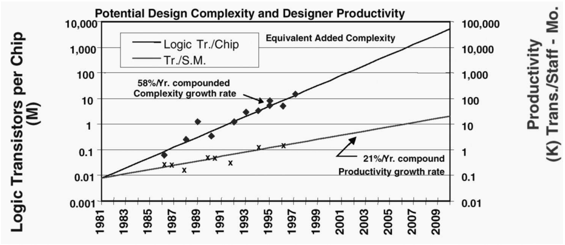
Role of Hardware design tools

- Design productivity is a problem due to chip complexity and time to market demands
- Need for well-integrated CAD with simulation, synthesis, verification and testing tools
- Modelling of system behaviour at all levels of abstraction with feedback to the designer
- Design re-use is a must but with max technology independence

Int. Technology Roadmap for Semiconductors says:

- 2010 will bring a system-on-a-chip with:
  - 4 billion 50-nanometer transistors, run at 10GHz
  - Moore's law: steady growth at 60% in the number of transistors per chip per year as the functionality of a chip doubles every 1.5-2 years.
- Technology troubles: process parameter variation, power dissipation (IBM S/390 chip operation PICA video), clock distribution etc. present new challenges for Design and Test
- But the biggest threat of all is **design cost**

Design productivity gap



A design team of 1000 working for 3 years on a MPU chip would cost some $1B (25% time spent on verification, 45% on redesign after first silicon)

Role of Modelling Language

- Design methods and tools require good modelling and specification techniques
- Those must be formal and rigorous and easy to comprehend (cf. timing diagrams, waveforms, traditionally used by logic designers)
- Today's hardware description languages allow high level of abstraction
- Models must allow for equivalence-preserving refinements
- They must allow for non-functional qualities such as speed, size and power

Why Petri nets are good for hardware design

- Finite State Machine is still the main formal tool in hardware design but it may be inadequate for distributed, concurrent and asynchronous hardware
- Petri nets:
  - simple and easy to understand graphical capture
  - modelling power adjustable to various types of behaviour at different abstraction levels
  - formal operational semantics and verification of correctness (safety and liveness) properties
  - possibility of mechanical synthesis of circuits from net models

A bit of history of their "marriage"

- 1950's and 60's: Foundations (Muller & Bartky, Petri, Karp & Miller, …)
- 1970's: Toward Parellel Computations (MIT, Toulouse, St. Petersburg, Manchester …)
- 1980's: First progress in VLSI and CAD, Concurrency theory, Signal Transition Graphs (STGs)
- 1990's: First asynchronous design (verification and synthesis) tools: SIS, Forcage, Petrify
- 2000's: Powerful asynchronous design flow (incl. hardware-software codesign and system-on-chip design)

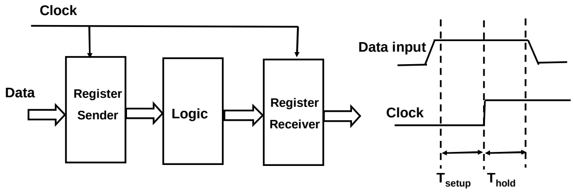
Introduction to Asynchronous Circuits

- What is an asynchronous circuit?
  - Physical (analogue) level
  - Logical level
  - Speed-independent and delay-insensitive circuits
- Why go asynchronous?
- Why control logic?
- Role of Petri nets
- Asynchronous circuit design based on Petri nets

What is an asynchronous circuit

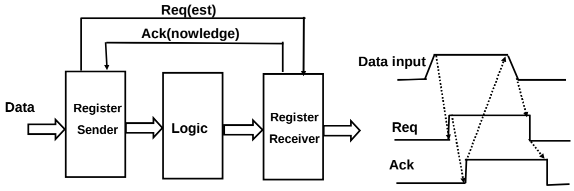
- No global clock; circuits are self-timed or self-clocked
- Can be viewed as hardwired versions of parallel and distributed programs – statements are activated when their guards are true
- No special run-time mechanism – the “program statements” are physical components: logic gates, memory latches, or hierarchical modules
- Interconnections are also physical components: wires, busses

Synchronous Design



Timing constraint: input data must stay unchanged within a setup/hold window around clock event. Otherwise, the latch may fail (e.g. metastability)

Asynchronous Design



Req/Ack (local) signal handshake protocol instead of global clock

Causal relationship

Handshake signals implemented with completion detection in data path

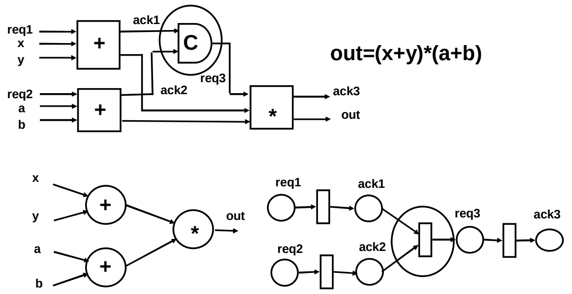
Physical (Analogue) level

- Strict view: an asynchronous circuit is a (analogue) dynamical system – e.g. to be described by differential equations
- In most cases can be safely approximated by logic level (0-to-1 and 1-to-0 transitions) abstraction; even hazards can be captured
- For some anomalous effects, such as metastability and oscillations, absolute need for analogue models
- Analogue aspects are not considered in this tutorial

Logical Level

- Circuit behaviour is described by sequences of up (0-to-1) and down (1-to-0) transitions on inputs and outputs
- The order of transitions is defined by causal relationship, not by clock (*a causes b,* directly or transitively)
- The order is partial if concurrency is present
- Two prominent classes of async circuits: speed-independent (work for any gate delay variations) and delay-insensitive (for both gate and wire delays)
- A class of async timed (not clocked!) circuits allows special timing order relations (*a occurs before b,* due to delay assumptions*)*

Simple circuit example



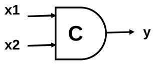
out=(x+y)*(a+b)

Data flow graph

Control flow graph – Petri net

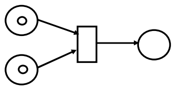
Muller C-element

Key component in asynchronous circuit design – like a Petri net transition

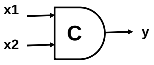


$$y=\underbrace{x1*x2}_{\text{Set-part}}+\underbrace{(x1+x2)y}_{\text{Reset-part}}$$

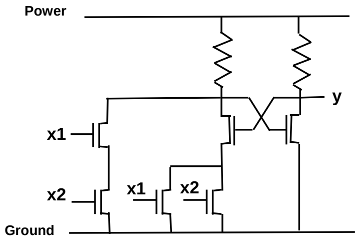
It acts symmetrically for pairs of 0-1 and 1-0 transitions – waits for both input events to occur



Muller C-element

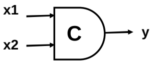


$$y=\underbrace{x1*x2}_{\text{Set-part}}+\underbrace{(x1+x2)y}_{\text{Reset-part}}$$

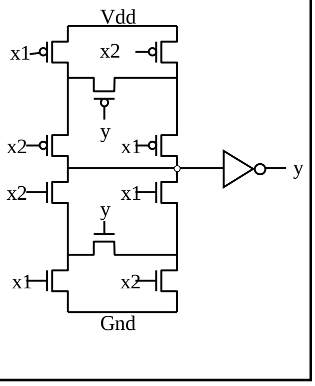


NMOS circuit implementation

Muller C-element (in CMOS)



$$y=\underbrace{x1*x2}_{\text{Set-part}}+\underbrace{(x1+x2)y}_{\text{Reset-part}}$$



Why asynchronous is good

- Performance (work on actual, not max delays)
- Robustness (operationally scalable; no clock distribution; important when gate-to-wire delay ratio changes)
- Low Power (‘change-based’ computing – fewer signal transitions)
- Low Electromagnetic Emission (more even power/frequency spectrum)
- Modularity and re-use (parts designed independently; well-defined interfaces)
- Testability (inherent self-checking via ack signals)

Obstacles to Async Design

- Design tool support – commercial design tools are aimed at clocked systems
- Difficulty of production testing – production testing is heavily committed to use of clock
- Aversion of majority of designers, trained ‘with clock’ – biggest obstacle
- Overbalancing effect of periodic (every 10 years) ‘asynchronous euphoria’

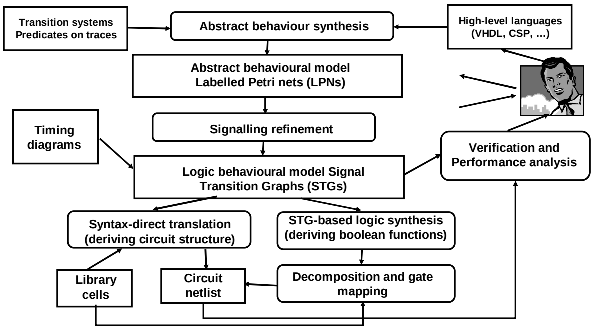
Why control logic

- Customary in hardware design to separate control logic from datapath logic due to different design techniques
- Control logic implements the control flow of a (possibly concurrent) algorithm
- Datapath logic deals with operational part of the algorithms
- Datapath operations may have their (lower level) control flow elements, so the distinction is relative
- Examples of *control-dominated* logic: a bus interface adapter, an arbiter, or a modulo-N counter
- Their behaviour is a combination of partial orders of signal events
- Examples of *data-dominated* logic are: a register bank or an arithmetic-logic unit (ALU)

Role of Petri Nets

- We concentrate here on control logic
- Control logic is behaviourally more diverse than data path
- Petri nets capture causality and concurrency between signalling events, deterministic and non-deterministic choice in the circuit and its environment
- They allow:
  - composition of labelled PNs (transition or place sync/tion)
  - refinement of event annotation (from abstract operations down to signal transitions)
  - use of observational equivalence (lambda-events)
  - clear link with state-transition models in both directions

Design flow with Petri nets



Transition systems
Predicates on traces
Abstract behaviour synthesis
High-level languages (VHDL, CSP, …)
Abstract behavioural model
Labelled Petri nets (LPNs)
Timing diagrams
Signalling refinement
Verification and Performance analysis
Logic behavioural model Signal Transition Graphs (STGs)
Syntax-direct translation (deriving circuit structure)
STG-based logic synthesis (deriving boolean functions)
Library cells
Circuit netlist
Decomposition and gate mapping

Hardware and Petri Nets: Modelling

Alex Yakovlev

Univ. Newcastle upon Tyne

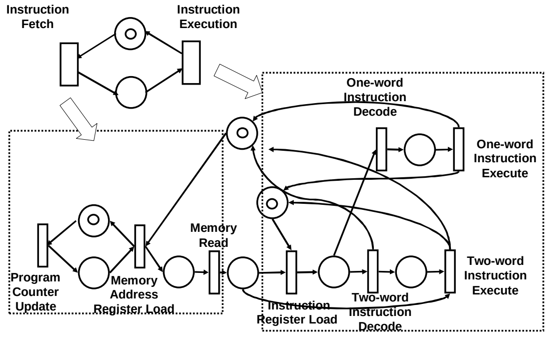
Advanced Course on Petri nets,

Eichstätt, 24-26 Sept, 2003

Modelling.Outline

- **High level modelling and abstract refinement; processor example**
- **Low level modelling and logic synthesis; interface controller example**
- **Modelling of logic circuits: event-driven and level-driven parts**
- **Properties analysed**

High-level modelling:Processor Example



Instruction Fetch
Instruction Execution
One-word Instruction Decode
One-word Instruction Execute
Memory Read
Program Counter Update
Memory Address Register Load
Instruction Register Load
Two-word Instruction Decode
Two-word Instruction Execute

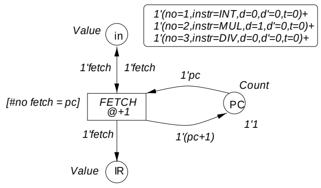
High-level modelling:Processor Example

- Details of further refinement, circuit implementation (by direct translation) and performance estimation (using UltraSan) are in:

  A. Semenov, A.M. Koelmans, L.Lloyd and A. Yakovlev. Designing an asynchronous processor using Petri Nets, IEEE Micro, 17(2):54-64, March 1997
- For use of Coloured Petri net models and use of Design/CPN in processor modelling:

  F.Burns, A.M. Koelmans and A. Yakovlev. Analysing superscalar processor architectures with coloured Petri nets, Int. Journal on Software Tools for Technology Transfer, vol.2, no.2, Dec. 1998, pp. 182-191.

Using Coloured Petri nets


1'(no=1,instr=INT,d=0,d'=0,t=0)+
1'(no=2,instr=MUL,d=1,d'=0,t=0)+
1'(no=3,instr=DIV,d=0,d'=0,t=0)+
Value
in
1'fetch
1'fetch
1'pc
Count
[#no fetch = pc]
FETCH
@+1
PC
1'1
1'fetch
1'(pc+1)
Value
IR

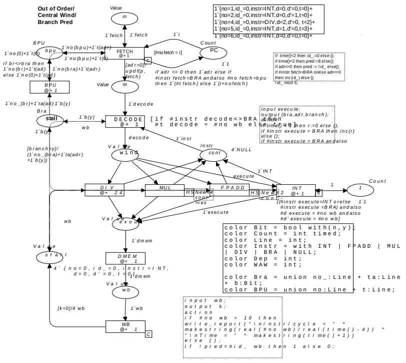
Color Set:

color Instr = with INT | FPADD | MUL | DIV | BRA | NULL;
color Line = int; color Dep = int;color Target = int; color Count = int timed;
*colour Value = record no:Line*instr:Instr*d:Dep*d':Dep*t:Target timed;*

Var Set:

var fetch : Value;
var pc : Count

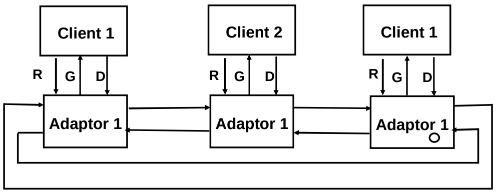
Using Coloured Petri nets


Out of Order/
Central Wind/
Branch Pred
Value
in
1`{no=1,id_=0,instr=INT,d=0,d'=0,t=0}+
1`{no=2,id_=0,instr=INT,d=0,d'=1,t=0}+
1`{no=4,id_=0,instr=DIV,d=2,d'=0, t=2}+
1`{no=5,id_=0,instr=INT,d=1,d'=0,t=3}+
1`{no=6,id_=0,instr=INT,d=0,d'=0,t=0}+
1`fetch
1`fetch
1`i
BPU
bpu
Count
PC
1`no(bpu)+1`t(adr)
1`no(0)+1`t(0)
1`no(bpu)+1`t(0)
FETCH
@+1
[#no fetch = i]
1`1
if br<>bra then
1`no(br)+1`t(ad)
else 1`no(0)+1`t(ad)
1`no(bra)+1`t(adr)
[adr=0]/
updf(p,
fetch)
if adr <> 0 then 1`adr else if
#instr fetch=BRA andalso #no fetch=bpu
then 1`(#t fetch) else 1`(i+nofetch)
if time()<2 then id_:=0 else ();
if time()<2 then pred:=8 else();
if adr<>0 then pred := !id_ else();
if #instr fetch=BRA orelse adr<>0
then inc(id_) else ();
!id_ mod 5;
BPU
@+1
Value
IR
1`no_(br)+1`ta(ad) 1`b(y)
1`decode
Bra
stall
1`b(y)
DECODE
@+ 1
[if #instr decode<>BRA then
#t decode = #no wb else true]
input execute;
output (bra,adr,branch);
action
if time() < 3 then r:=0 else ();
if #instr execute = BRA then inc(r)
else ();
if #instr execute = BRA andalso
wb
1`b(y)
decode
1`inst
Value
wind
Instr
cont
4`NULL
[branch=y]/
(1`no_(bra)+1`ta(adr)
+1`b(y))
1`INT
execute
DIV
@+ 24
MUL
FPADD
INT
@+1
1
Count
1
HS New#4
cont->res
HS New#2
cont->
[(#instr execute=INT orelse
#instr execute =BRA) andalso
#d execute = #no wb andalso
#d' execute = #no wb]
1`1
1`execute
wb
Value
exou
1`dmem
Value
stall
DMEM
@+ 1
4`{no=0,id_=0,instr=INT,
d=0,d'=0,t=0}
1`dmem
Value
wb
[k=0]/4`wb
1`wb
WB
@+ 1

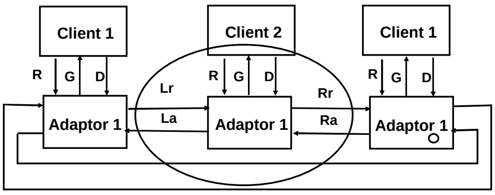
```
color Bit = bool with(n,y);
color Count = int timed;
color Line = int;
color Instr = with INT | FPADD | MUL
| DIV | BRA | NULL;
color Dep = int;
color WAW = int;

color Bra = union no_:Line + ta:Line
+ b:Bit;
color BPU = union no:Line + t:Line;
```

```
input wb;
output k;
action
if #no wb = 10 then
write_report("\ninstr/cycle = " ^
makestring(real(#no wb)/real(time()-4)) ^
"\nTime = " ^ makestring(time()+1))
else ();
if !pred=#id_ wb then 1 else 0;
```

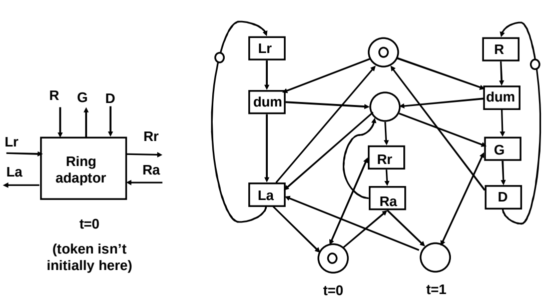
Low-level modelling: "lazy token" ring adaptor



Low-level modelling: "lazy token" ring adaptor



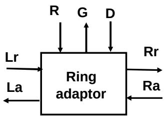
Lazy ring adaptor



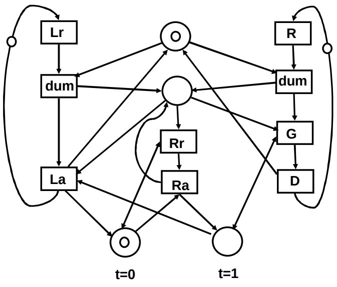
t=0

(token isn't initially here)

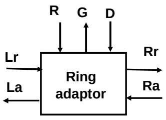
Lazy ring adaptor



t=0->1->0
(token must be
taken from the
right and past to
the left)

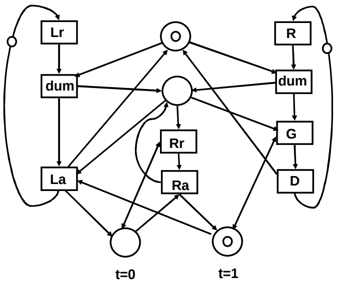


Lazy ring adaptor

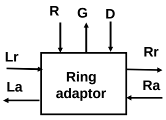


t=1

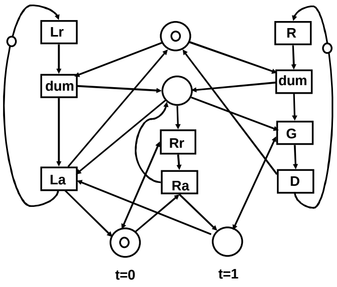
(token is
already
here)



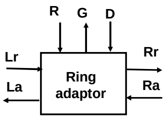
Lazy ring adaptor



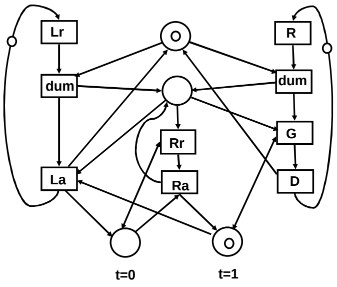
t=0->1
(token must be
taken from the
right)



Lazy ring adaptor



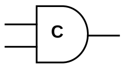
t=1
(token is here)



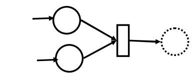
Logic Circuit Modelling

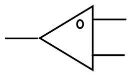
Event-driven elements

Petri net equivalents

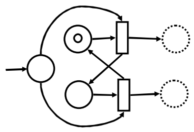


Muller C-element





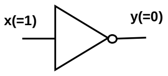
Toggle



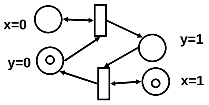
Logic Circuit Modelling

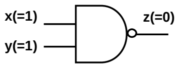
Level-driven elements

Petri net equivalents

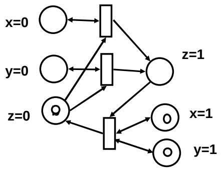


NOT gate

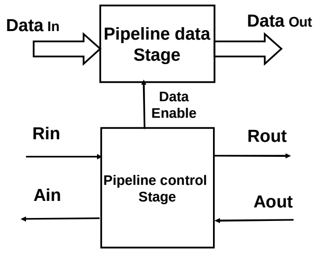




NAND gate



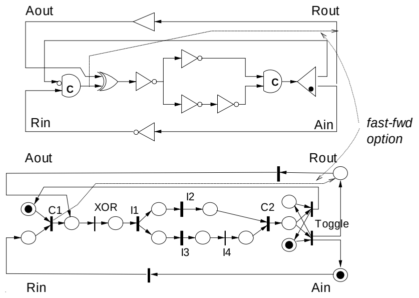
Logic Circuit Modelling: examples



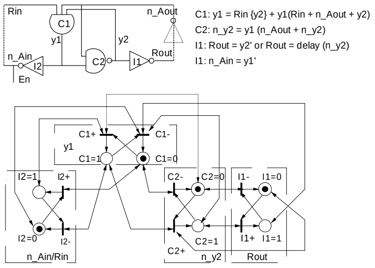
Pipeline control must guarantee:

- **Handshake protocols between the stages**
- **Safe propagation of the previous datum before the next one**

Event-driven circuit

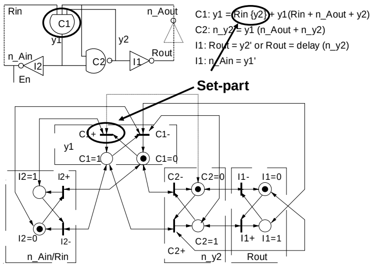


Level-driven circuit



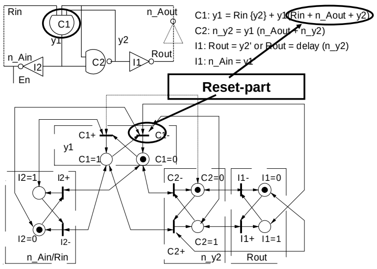
C1: y1 = Rin {y2} + y1(Rin + n_Aout + y2)
C2: n_y2 = y1 (n_Aout + n_y2)
I1: Rout = y2' or Rout = delay (n_y2)
I1: n_Ain = y1'

Level-driven circuit



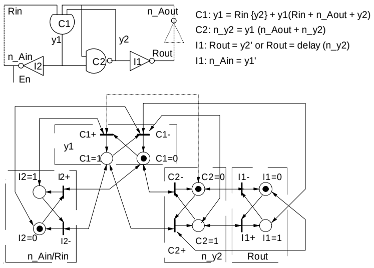
C1: y1 = Rin {y2} + y1(Rin + n_Aout + y2)
C2: n_y2 = y1 (n_Aout + n_y2)
I1: Rout = y2' or Rout = delay (n_y2)
I1: n_Ain = y1'

Level-driven circuit



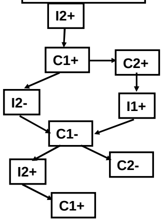
C1: y1 = Rin {y2} + y1(Rin + n_Aout + y2)
C2: n_y2 = y1 (n_Aout + n_y2)
I1: Rout = y2' or Rout = delay (n_y2)
I1: n_Ain = y1'

Level-driven circuit

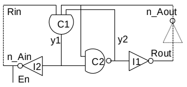


C1: y1 = Rin {y2} + y1(Rin + n_Aout + y2)
C2: n_y2 = y1 (n_Aout + n_y2)
I1: Rout = y2' or Rout = delay (n_y2)
I1: n_Ain = y1'

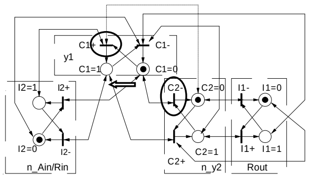
Without y2 in Set part of y1 this trace can happen:



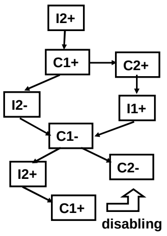
Level-driven circuit



C1: y1 = Rin {y2} + y1(Rin + n_Aout + y2)
C2: n_y2 = y1 (n_Aout + n_y2)
I1: Rout = y2' or Rout = delay (n_y2)
I1: n_Ain = y1'



Without y2 in Set part of y1 this trace can happen:



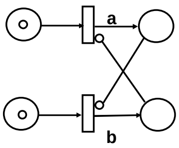
Properties analysed

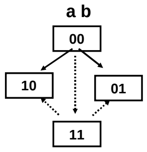
- Functional correctness (need to model environment)
- Deadlocks
- Hazards:
  - *non-1-safeness for event-based*
  - *non-persistency for level-based*
- Timing constraints
  - Absolute (need Time(d) Petri nets)
  - Relative (compose with a PN model of order conditions)

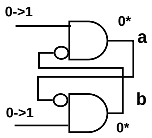
More about this in the Verification Part

How adequate is PN model?

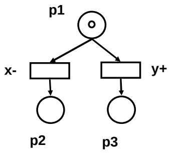
- Petri nets have events with *atomic action* semantics
- Asynchronous circuits may exhibit behaviour that does not fit within this domain – due to *inertia*

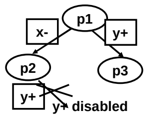




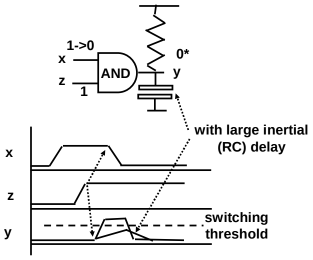


Petri Nets versus Circuits





y+ disabled (no memory of past)

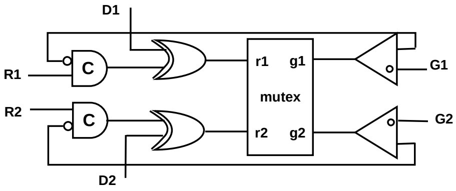


Race between x- and y+ causes nondeterministic behaviour on y:

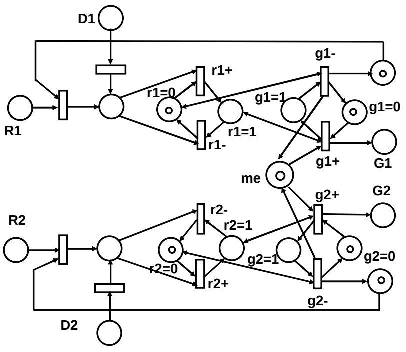
(1) Either there is a 0-1-0 pulse

(2) Or nothing

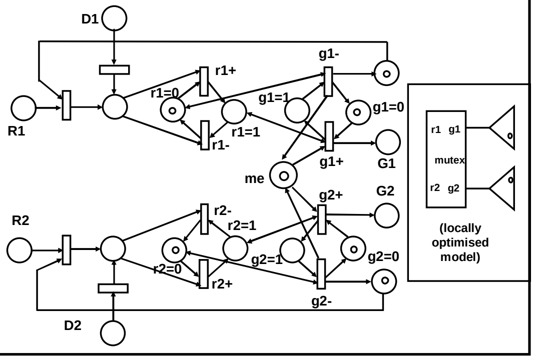
Request-Grant-Done (RGD) arbiter



Request-Grant-Done (RGD) arbiter

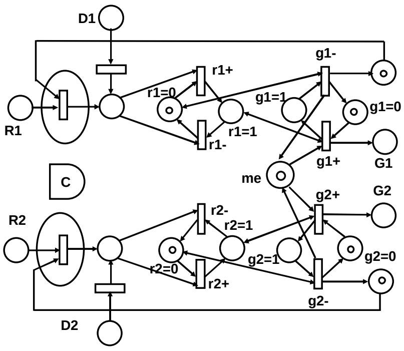


Request-Grant-Done (RGD) arbiter

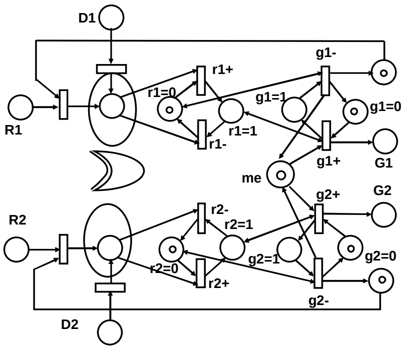


(locally optimised model)

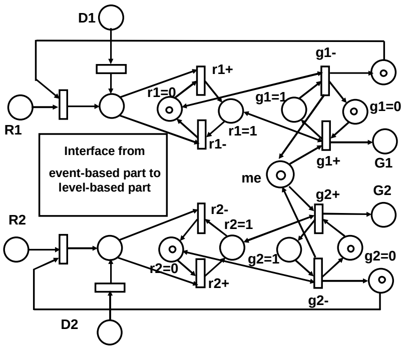
Request-Grant-Done (RGD) arbiter



Request-Grant-Done (RGD) arbiter

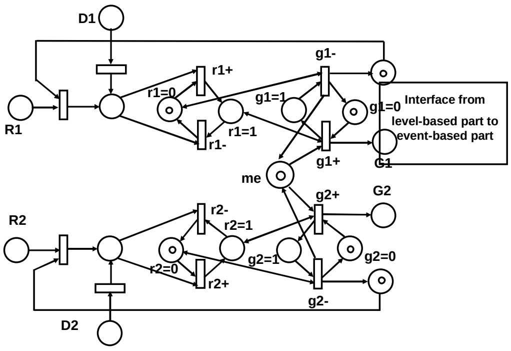


Request-Grant-Done (RGD) arbiter



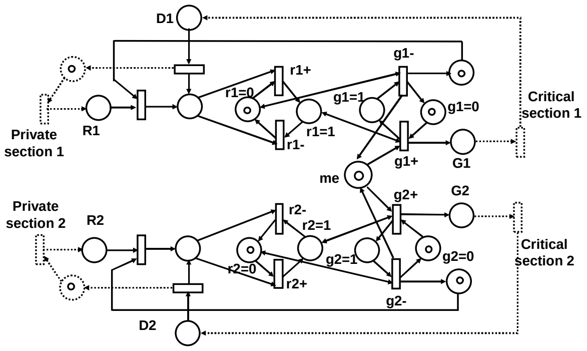
Interface from event-based part to level-based part

Request-Grant-Done (RGD) arbiter



Interface from level-based part to event-based part

Request-Grant-Done (RGD) arbiter with environment



Modelling. Conclusions

- Choosing the right level of modelling is crucial
- Refinement of Petri net models and interpretation can be used in hardware design
- Petri nets are too abstract to capture analogue phenomena in circuits
- However, non-persistence or non-safeness can (conservatively) approximate the possibility of hazards

Hardware and Petri Nets: Synthesis

Alex Yakovlev

Univ. Newcastle upon Tyne

Advanced Course on Petri nets,

Eichstätt, 24-26 Sept, 2003

Tutorial Outline

- **Introduction**
- **Modeling Hardware with PNs**
- **Synthesis of Circuits from PN specifications**
- **Circuit verification with PNs**
- **Performance analysis using PNs**

Hardware Design and Petri Nets – Adv. Tutorial

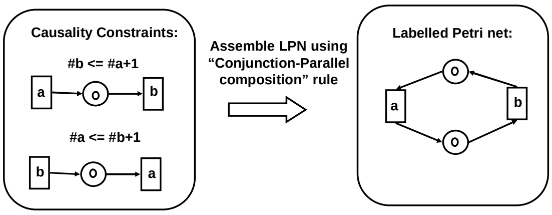
Synthesis.Outline

- Abstract synthesis of Labelled PNs (LPNs) from causality constraints and transition systems
- Handshake and signal refinement (LPN-to-STG)
- Direct translation of LPNs and STGs to circuits
- Logic synthesis from STGs

Synthesis from Causality Constraints (compositional approach)

- Behaviour defined in terms of *Causality Constraints* - characteristic predicates defined on traces
- These constraints produce LPN “snippets”
- Construction of LPNs as compositions of snippets
- Examples: n-place buffer, 2-way merge

Synthesis from Causality Constraints



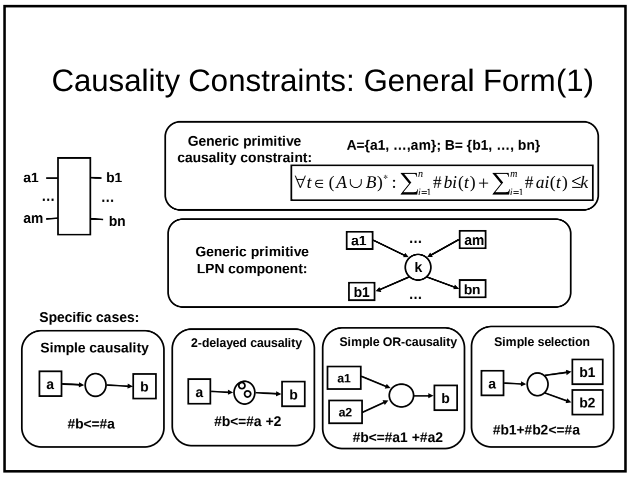
Causality Constraints:

#b <= #a+1

#a <= #b+1

Assemble LPN using “Conjunction-Parallel composition” rule

Labelled Petri net:



Causality Constraints: General Form(1)

Generic primitive causality constraint: A={a1, …,am}; B= {b1, …, bn}

$$\forall t \in (A \cup B)^* : \sum_{i=1}^{n} \#bi(t) + \sum_{i=1}^{m} \#ai(t) \le k$$

Generic primitive LPN component:

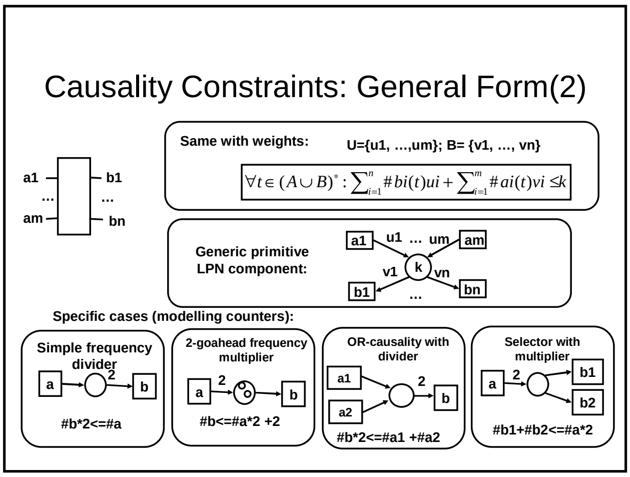
Specific cases:

Simple causality: #b<=#a

2-delayed causality: #b<=#a +2

Simple OR-causality: #b<=#a1 +#a2

Simple selection: #b1+#b2<=#a



Causality Constraints: General Form(2)

Same with weights: U={u1, …,um}; B= {v1, …, vn}

$$\forall t \in (A \cup B)^* : \sum_{i=1}^{n} \#bi(t)ui + \sum_{i=1}^{m} \#ai(t)vi \le k$$

Generic primitive LPN component:

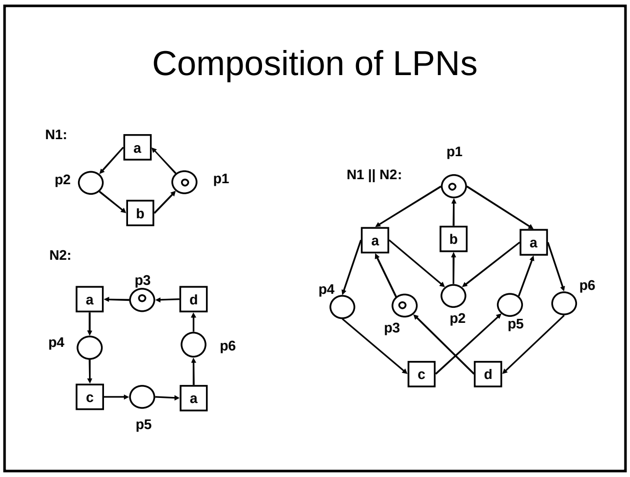
Specific cases (modelling counters):

Simple frequency divider: #b*2<=#a

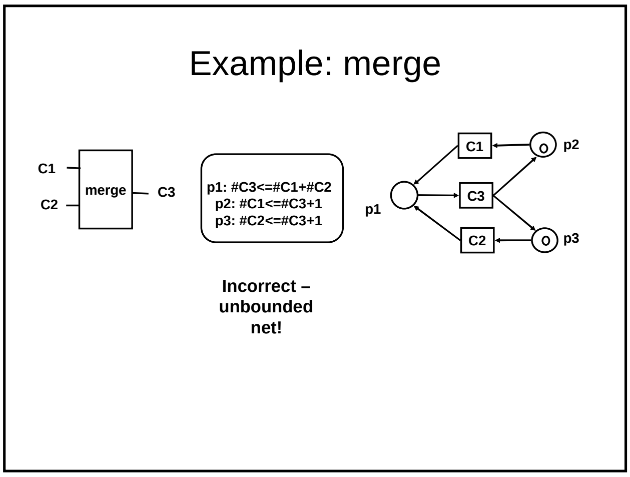
2-goahead frequency multiplier: #b<=#a*2 +2

OR-causality with divider: #b*2<=#a1 +#a2

Selector with multiplier: #b1+#b2<=#a*2



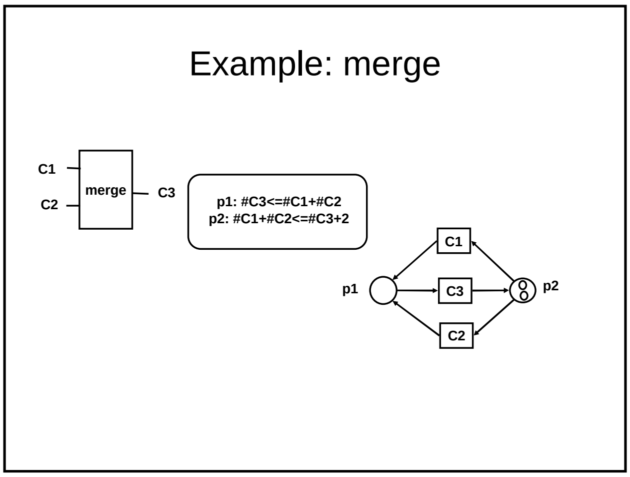
Composition of LPNs



Example: merge

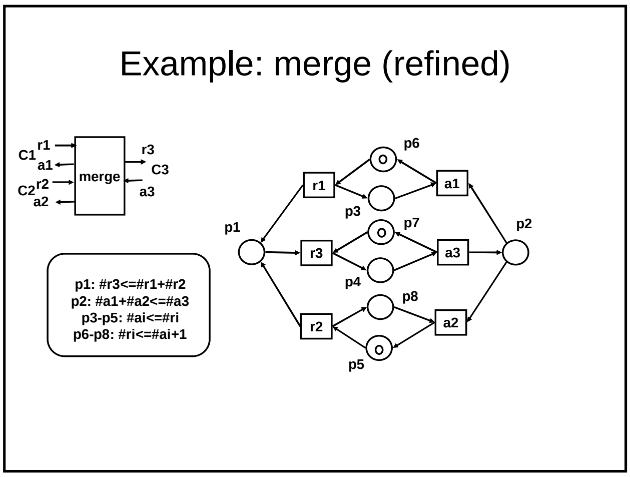
p1: #C3<=#C1+#C2
p2: #C1<=#C3+1
p3: #C2<=#C3+1

Incorrect – unbounded net!



Example: merge

p1: #C3<=#C1+#C2
p2: #C1+#C2<=#C3+2

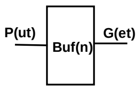


Example: merge (refined)

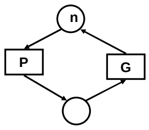
p1: #r3<=#r1+#r2
p2: #a1+#a2<=#a3
p3-p5: #ai<=#ri
p6-p8: #ri<=#ai+1

More examples

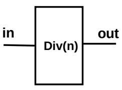
n-place buffer



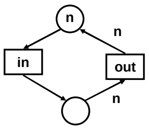
p1: #P<=#G+n
p2: #G<=#P+n



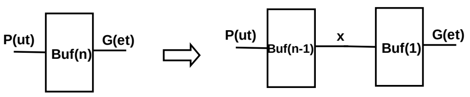
Modulo-n counter (frequency divider)

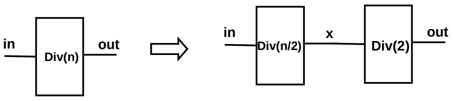


p1: #<in=#out*2+2
p2: #out*2<=#in

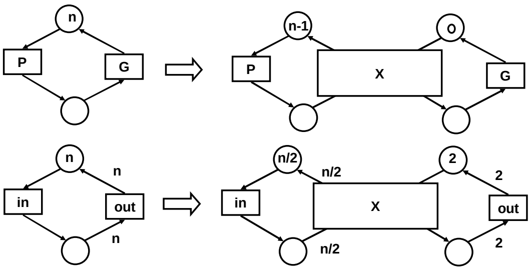


Decomposition of LPNs

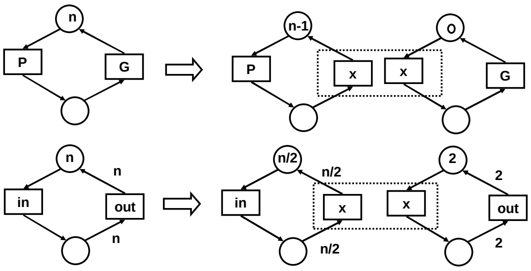




Decomposition of LPNs



Decomposition of LPNs

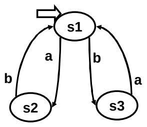


Synthesis from transition systems

- Modelling behaviour in terms of a sequential capture – *Transition System*
- Synthesis of LPN (distributed and concurrent object) from TS (using *theory of regions*)
- Examples: one place buffer, counterflow pp

Transition Systems

Original TS specification



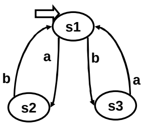
Not (semi-)elementary

The relationship between Transition Systems and Petri nets and conditions for synthesizability of a PN from a TS are based on *Theory of Regions*

(Ehrenfeucht, Rozenberg, Nielsen, Thiagarajan, Mukund, Darondeau et al.)

Transition Systems and regions

Original TS specification

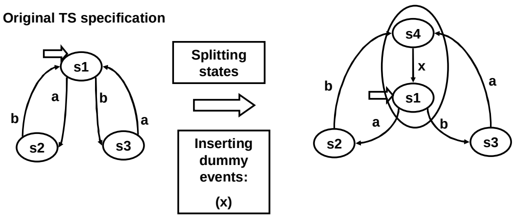


Not (semi-)elementary

No non-trivial regions!

Transition Systems and regions

Original TS specification



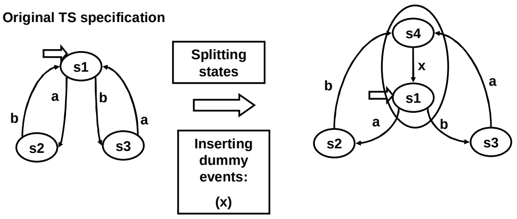
Splitting states

Inserting dummy events:

(x)

Transition Systems and regions

Original TS specification



Splitting states

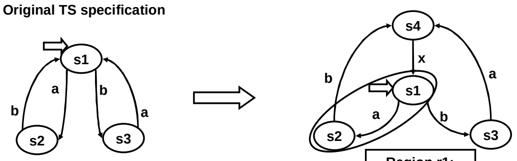
Inserting dummy events:

(x)

This transformation preserves observational equivalence

Transition Systems and regions

Original TS specification



Region r1:

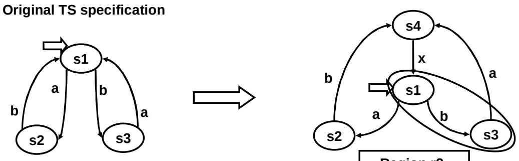
exit(b)

enter(x)

no-cross(a)

Transition Systems and regions

Original TS specification



Region r2:

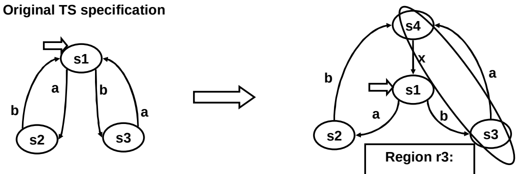
exit(a)

enter(x)

no-cross(b)

Transition Systems and regions

Original TS specification



Region r3:

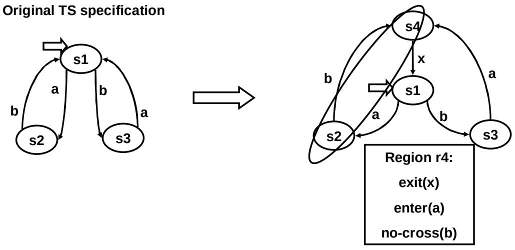
exit(x)

enter(b)

no-cross(a)

Transition Systems and regions

Original TS specification



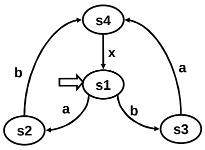
Region r4:

exit(x)

enter(a)

no-cross(b)

From Transition System to LPN



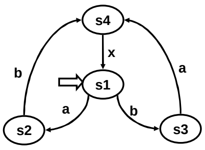
Regions in the TS are associated with places in the LPN

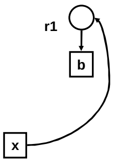
Events are associated with transitions

Exit/entry/no-cross relations are associated with pre/post relations

From Transition System to LPN

r1: exit (b), enter(x), no-cross(a)

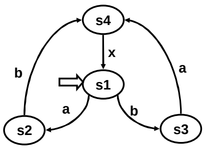


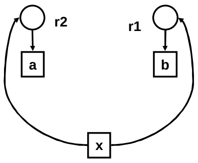


From Transition System to LPN

r1: exit (b), enter(x), no-cross(a)

r2: exit (a), enter(x), no-cross(b)



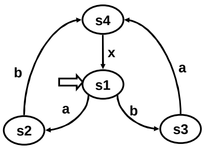


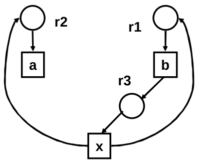
From Transition System to LPN

r1: exit (b), enter(x), no-cross(a)

r2: exit (a), enter(x), no-cross(b)

r3: exit (x), enter(b), no-cross(a)





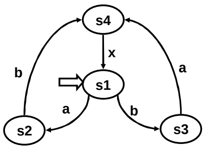
From Transition System to LPN

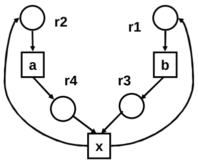
r1: exit (b), enter(x), no-cross(a)

r2: exit (a), enter(x), no-cross(b)

r3: exit (x), enter(b), no-cross(a)

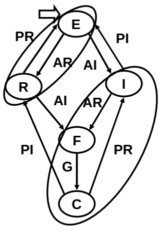
r4: exit (x), enter(a), no-cross(b)





Example: counterflow pipeline

Molnar's 5-state TS:



Examples of regions:

r1={E,R} – pre-region(AI), post-region(PI)

r2={I,F,C} – pre-region(PI), post-region(AI), co-region(G)

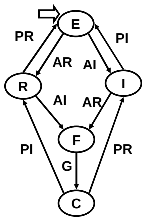
Notation: exit(a) -> pre-region(a), entry(a) -> post-region(a), inside(a) -> co-region(a)

Violation of semi-elementarity:

1. Intersection of pre-regions (only r2!) for PI {I,F,C} is not equal to Excitation Region for PI {I,C}
2. Intersection of pre-regions (empty!) for G is not equal to Excitation Region for G {F}

Example: counterflow pipeline

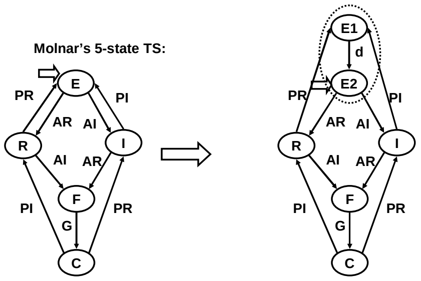
Molnar's 5-state TS:



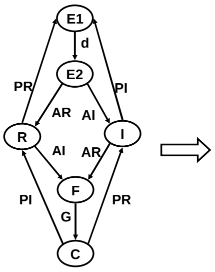
Solution:

Split a state (E) and insert a silent action (d), preserving behavioural (observational) equivalence

Example: counterflow pipeline



Example: counterflow pipeline



Minimal set of regions:

r1 = {E1,E2,I} <- pre(AR), post(PR)

r2 = {E1,E2,R} <- pre(AI), post (PI), co(d)

r3 = {R,F,C} <- pre(PR), post(AR), co(G)

r4 = {I,F,C} <- pre(PI), post(AI)

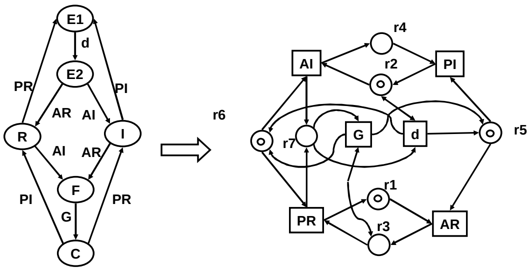
r5 = {E2,I,C} <- pre(PI,AR), post(G,d)

r6 = {E2,R,C} <- pre(PR,AI), post(G,d)

r7 = {E1,I,F} <- pre(G,d), post(PR,AI)

Semi-elementary TS

Example: counterflow pipeline



Semi-elementary TS

Semi-elementary Petri net

Synthesis from process-based languages

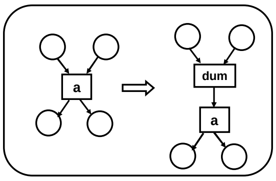
- Modelling behaviour in terms of a process (-algebraic) specifications (CSP, …)
- Synthesis of LPN (concurrent object with explicit causality) from process-based model (concurrency is explicit but causality implicit)
- Examples: modulo-N counter

Refinement at the LPN level

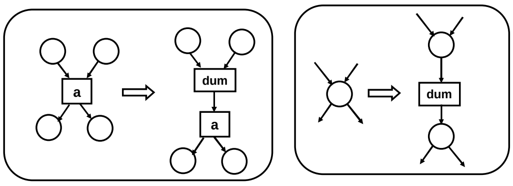
- Examples of refinements:
  - Introduction of “silent” events
  - Handshake refinement
  - Signalling protocol refinement (return-to-zero versus non-return-to-zero)
  - Arbitration refinement

  All these refinements must preserve behavioural equivalence (discussed below) and some other properties at the STG level (discussed later)
- What is implemented in Petrify and what isn’t (yet)

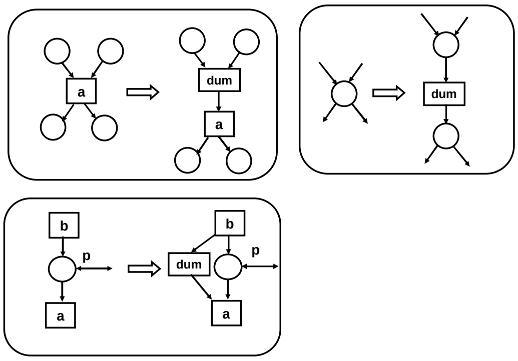
Structural refinement in LPN



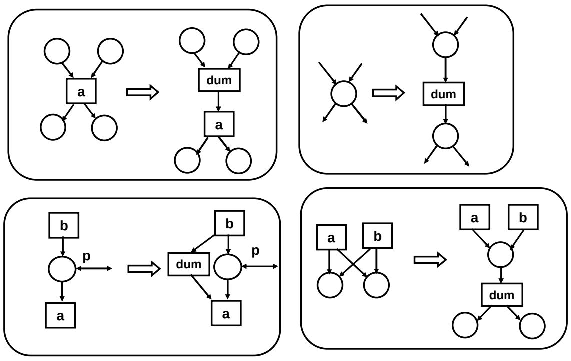
Structural refinement in LPN



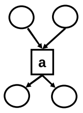
Structural refinement in LPN



Structural refinement in LPN

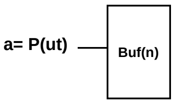


Handshake refinement



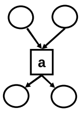
Let abstract event (action) “a” be associated with some port of the control circuit

E.g.

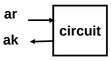


This may lead to the following refinements at the circuit level

Handshake refinement

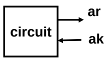


Passive handshake

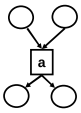


Environment produces (first) request

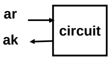
Active handshake



Handshake refinement

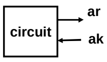


Passive handshake

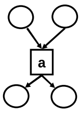


Environment produces (first) request

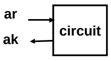
Active handshake



Handshake refinement

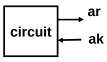


Passive handshake



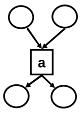
Environment produces (first) request

Active handshake

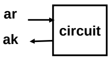


Circuit produces (first) request

Handshake refinement

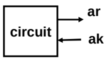


Passive handshake



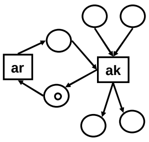
Environment produces (first) request

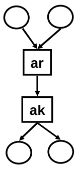
Active handshake



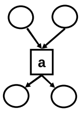
Circuit produces (first) request

Two phase, non-return-to-zero (NRZ) protocol

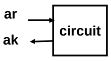




Handshake refinement

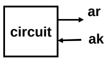


Passive handshake



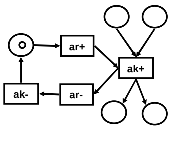
Environment produces (first) request

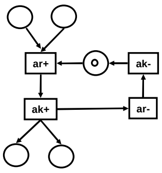
Active handshake



Circuit produces (first) request

Four phase, return-to-zero (RTZ) protocol

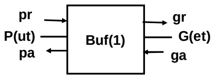




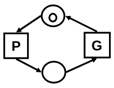
Handshake refinement example

passive h/s

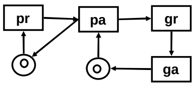
active h/s



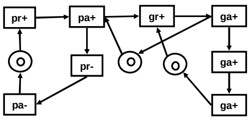
Initial LPN:



Two phase (NRZ) protocol:



Four phase (RTZ) protocol:

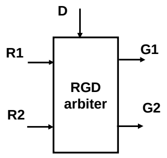


Subsequent transformations are possible at STG level – e.g. re-shuffling of non-critical (resetting) transitions (discussed later)

Arbitration refinement

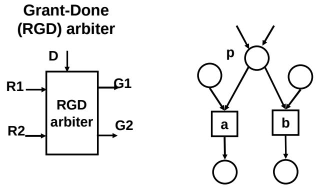
- •Asynchronous circuits often require elements to resolve *conflicts* which are intentionally “pre-programmed” in specifications
- •These elements are *similar to semaphores* (etc.) in concurrent programs
- •These elements are different from logical gates because they involve *internally analogue* components
- •The LPN model must be refined to explicitly *“factorise” non-persistent behavior* from the rest of the model – the latter can be synthesized using logic gates

E.g. Request-Grant-Done (RGD) arbiter



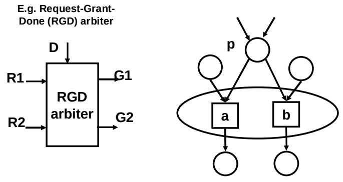
Arbitration refinement

E.g. Request-Grant-Done (RGD) arbiter



Arbitration refinement

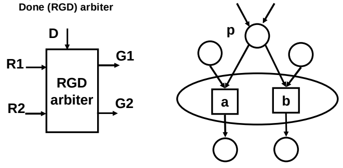
E.g. Request-Grant-Done (RGD) arbiter

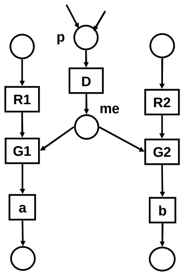


Assume a and b are circuit actions that are in conflict (may disable each other) and need to be protected

Arbitration refinement

E.g. Request-Grant-Done (RGD) arbiter

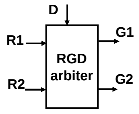


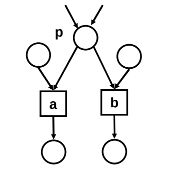


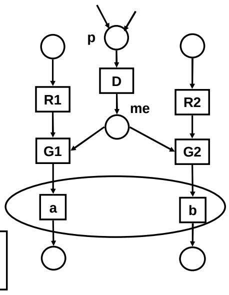
Assume a and b are circuit actions that are in conflict (may disable each other) and need to be protected

Arbitration refinement

E.g. Request-Grant-Done (RGD) arbiter







a and b are protected now (they are no longer disabled)

Translation of LPNs to circuits

- After appropriate refinements have been made one can translate Labelled Petri nets (or Signal Transition Graphs) into circuits
- Either by syntax-direct translation (discussed below)
- Or by using Logic Synthesis (discussed later)

Why direct translation?

- Direct translation has linear complexity but can be area inefficient (inherent one-hot encoding)
- Logic synthesis has problems with state space explosion, repetitive and regular structures (log-based encoding approach)

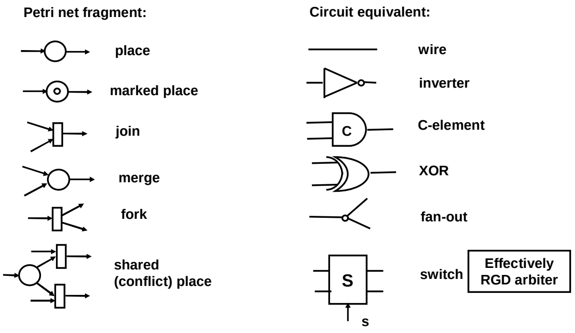
Direct Translation of Petri Nets

- Previous work dates back to 70s
- Synthesis into *event-based (two-phase)* circuits (similar to Sutherland’s micropipeline control)
  - S.Patil, F.Furtek (MIT)
- Synthesis into *level-based (4-phase)* circuits (similar to synthesis from one-hot encoded FSMs)
  - R. David (’69, translation FSM graphs to CUSA cells)
  - L. Hollaar (’82, translation from parallel flowcharts)
  - V. Varshavsky et al. (’90,’96, translation from PN into an interconnection of David Cells)

Synthesis into event-based circuits

- Patil’s translation method for simple PNs
- Furtek’s extension to 1-safe net
- “Pragmatic” extensions to Patil’s set (for non-simple PNs)
- Examples: modulo-N counter, Lazy ring adapter

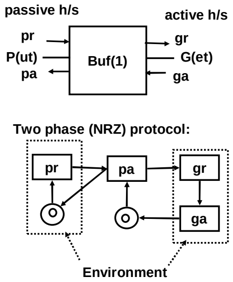
Patil’s set of modules



Petri net fragment:	Circuit equivalent:
place	wire
marked place	inverter
join	C-element
merge	XOR
fork	fan-out
shared (conflict) place	switch

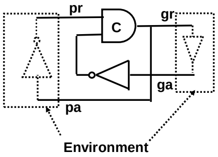
Effectively RGD arbiter

Example



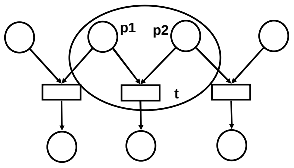
Two-phase implementation

(using Patil’s elements):



Simple Net restriction

Patil’s translation was restricted to (1-safe) Simple Nets

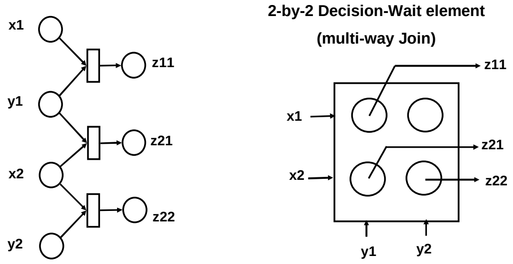


Violation of simplicity: transition t has more than one input place (p1 and p2) that is input to other transitions

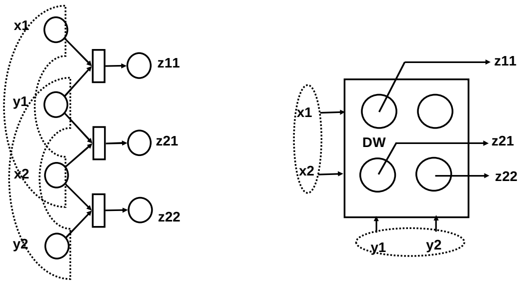
Extension to Simple Nets

2-by-2 Decision-Wait element

(multi-way Join)

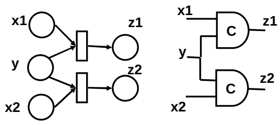


Extension to Simple Nets



(x1 and x2) and (y1 and y2) must pairs of mutually exclusive events

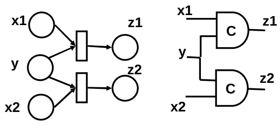
Problems with C-elements



Can we just use a pair of C-elements to implement a 2-by-1 Decision wait?

x1 and x2 are mutually exclusive – so no need for a S-switch (RGD arbiter)

Problems with C-elements



Can we just use a pair of C-elements to implement a 2-by-1 Decision wait?

No.

C-elements can only synchronise:

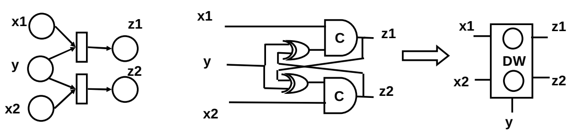
rising (0-1) with rising (0-1) or

falling (1-0) with falling (1-0) but

not rising (0-1) with falling (1-0)

x1 and x2 are mutually exclusive – so no need for a S-switch (RGD arbiter)

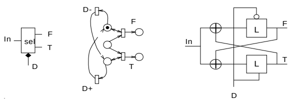
Problems with C-elements



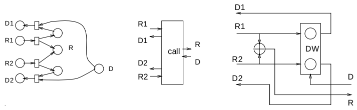
x1 and x2 are mutually exclusive – so no need for a S-switch (RGD arbiter)

Other useful elements

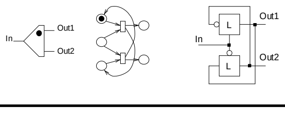
Select:



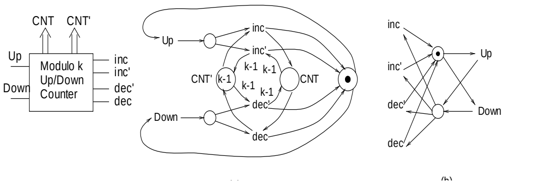
Call:



Toggle:

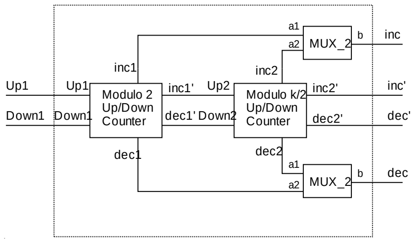


Direct synthesis example (modulo-k Up-Down counter)



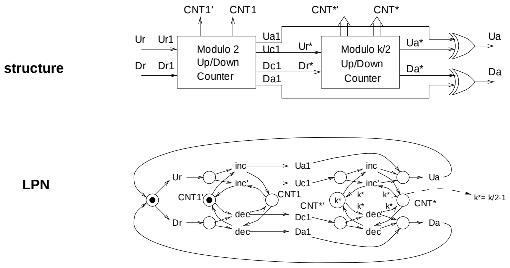
Mod-k counter LPN **Environment LPN**

Direct synthesis example (modulo-k Up-Down counter)

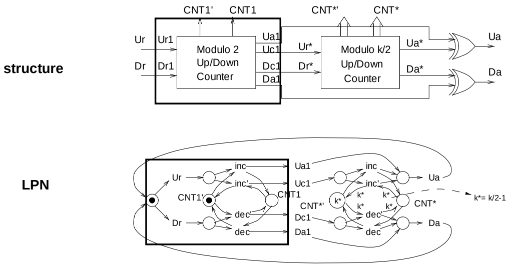


Decomposition (structural view)

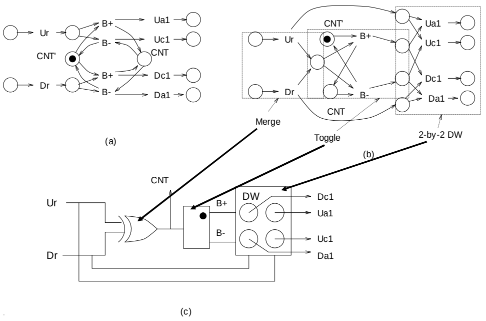
Direct synthesis example (modulo-k Up-Down counter)



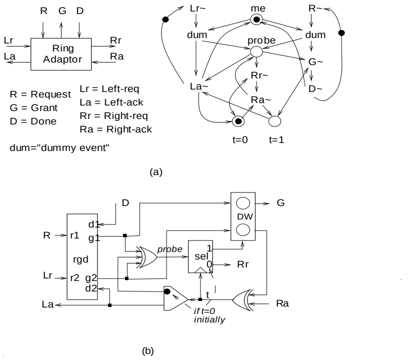
Direct synthesis example (modulo-k Up-Down counter)



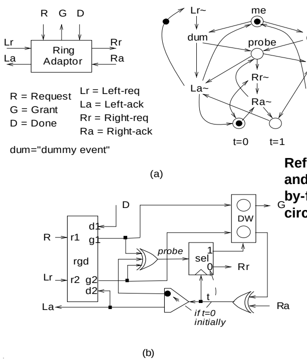
Direct synthesis example (modulo-k Up-Down counter)



Direct synthesis example (lazy token ring adapter)



Direct synthesis example (lazy token ring adapter)



(a)

(b)

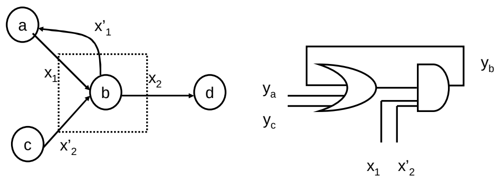
Exercise:

Refine this initial LPN and map it (fragment-by-fragment) to the circuit

Synthesis into level-based circuits

- David's method for asynchronous Finite State Machines
- Holaar's extensions to parallel flow charts
- Varshavsky's method for 1-safe Petri nets: based on associating *places with latches*
- Examples: counter, VME bus, butterfly circuit

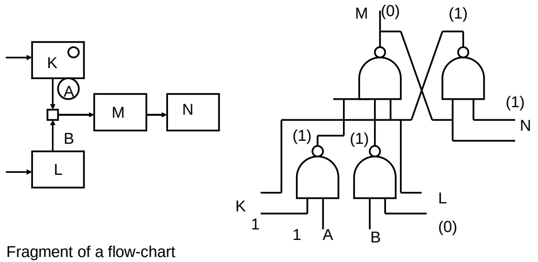
David's original approach



Fragment of a State Machine flow graph

CUSA element for storing state b

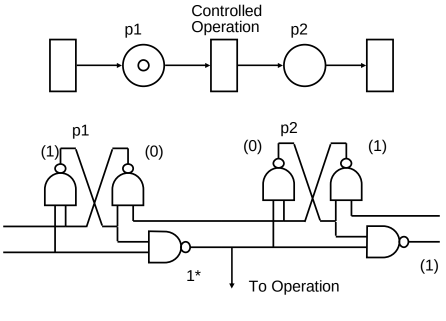
Hollaar's approach



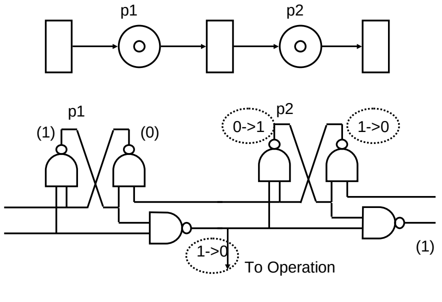
Fragment of a flow-chart (allows parallelism)

One-hot circuit cell

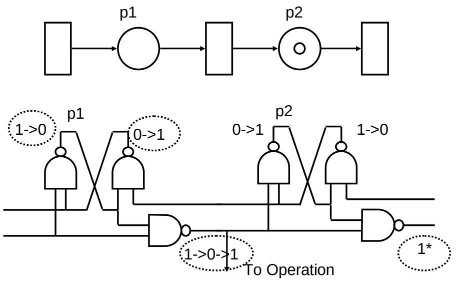
Varshavsky's Approach



Varshavsky's Approach



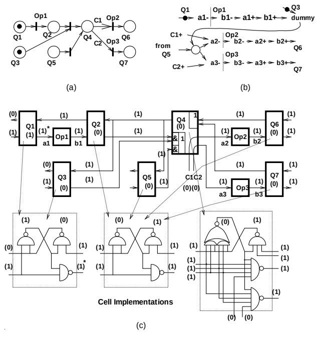
Varshavsky's Approach



Varshavsky's Approach

- This method associates *places* with *latches* (flip-flops) – so the state memory (marking) of PN is directly mimicked in the circuit's state memory
- *Transitions* are associated with *controlled actions* (e.g. activations of data path units or lower level control blocks – by using handshake protocols)
- Modelling discrepancy (be careful!):
  - in Petri nets removal of a token from pre-places and adding tokens in post-places is *instantaneous* (i.e. *no intermediate states*)
  - in circuits the "move of a token" has a *duration* and *there is an intermediate state*

Translation in brief



This method has been used for designing control of a token ring adaptor

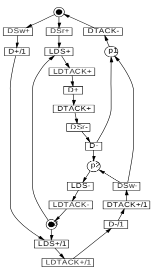
[Yakovlev, Varshavsky, Marakhovsky, Semenov, IEEE Conf. on Asynchronous Design Methodologies, London, 1995

Direct translation examples

In this work we tried direct translation:

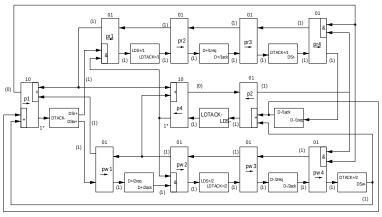
- From STG-refined specification (VME bus controller)
  - Worse than logic synthesis
- From a largish abstract specification with high degree of repetition (mod-6 counter)
  - Considerable gain to logic synthesis
- From a small concurrent specification with dense coding space ("butterfly" circuit)
  - Similar or better than logic synthesis

Example 1: VME bus controller



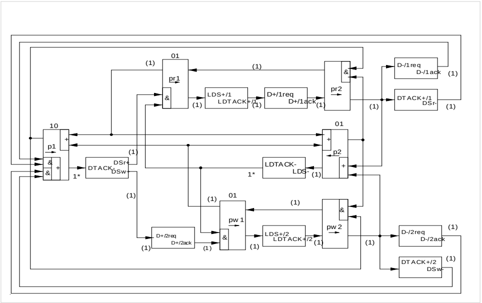
INPUTS: DSr,DSw,LDTACK
OUTPUTS: D,LDS,DTACK

Result of direct translation (unoptimised):

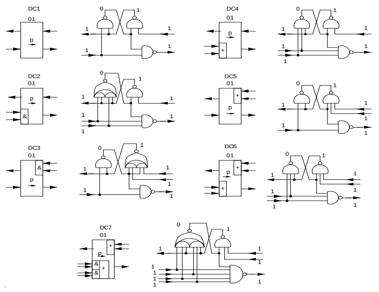


VME bus controller

After DC-optimisation (in the style of Varshavsky et al WODES'96)

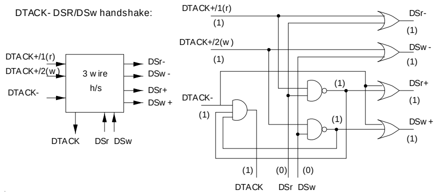


David Cell library



“Data path” control logic

Example of interface with a handshake control (DTACK, DSR/DSW):

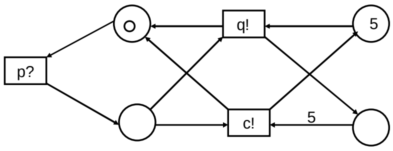


Example 2: “Flat” mod-6 Counter

CSP-like Specification:

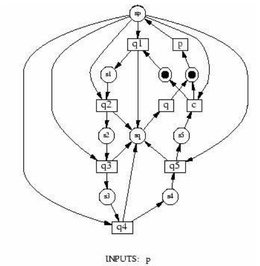
`((p?;q!)⁵;p?;c!)*`

Petri net (5-safe):



“Flat” mod-6 Counter

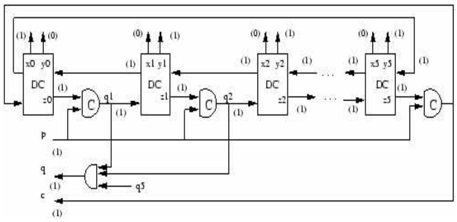
Refined (by hand) and optimised (by Petrify) Petri net:



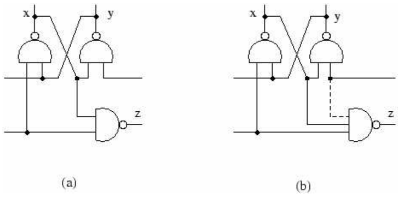
INPUTS: p
OUTPUTS: q,c
DUMMY: q1,q2,q3,q4,q5

“Flat” mod-6 counter

Result of direct translation (optimised by hand):

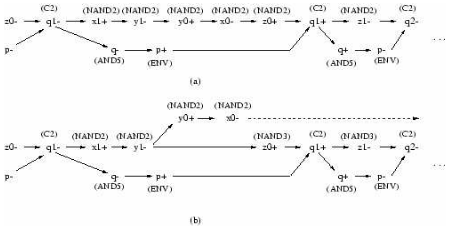


David Cells and Timed circuits



(a) Speed-independent (b) With Relative Timing

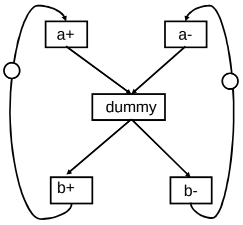
“Flat” mod-6 counter



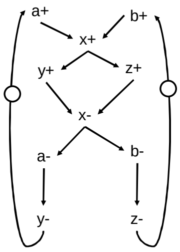
(a) speed-independent (b) with relative timing

“Butterfly” circuit

Initial Specification:

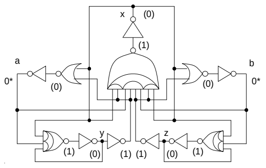


STG after CSC resolution:



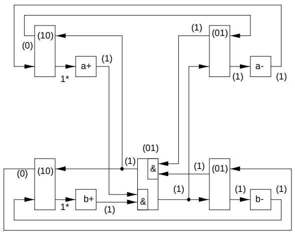
“Butterfly” circuit

Speed-independent logic synthesis solution:



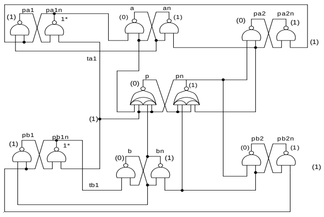
“Butterfly” circuit

Speed-independent DC-circuit:



“Butterfly” circuit

DC-circuit with aggressive use of timing assumptions:



Conclusion on direct synthesis

- Direct synthesis allows to implement practically any bounded labelled Petri net specification (suitably interpreted and refined to the level of signals)
- Direct synthesis produces control circuits that are structurally “homomorphic” to Petri nets (translation complexity is low)
- Direct synthesis is not affected by state explosion, so large controllers or prototypes can be constructed at low cost
- Larger size of direct translation circuits does not however mean less speed
- New synthesis methods will combine direct translation with logic synthesis

Hardware and Petri Nets: Verification of Asynchronous Circuits using Partial Order Techniques

Alex Yakovlev

Univ. Newcastle upon Tyne

Advanced Course on Petri nets,

Eichstätt, 24-26 Sept, 2003

Outline

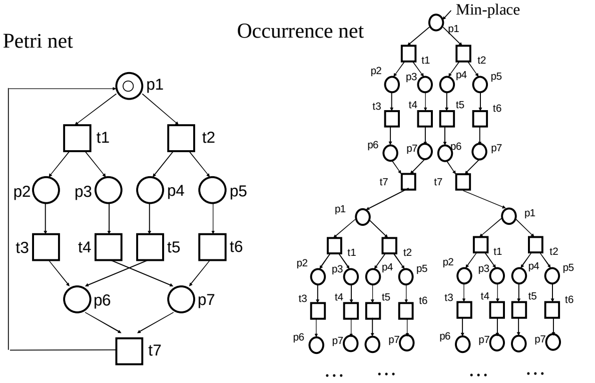
- Representing Petri net semantics with occurrence nets (unfoldings)
- Unfolding (finite) prefix construction
- Analysis of asynchronous circuits
- Problems with efficient unfolding

Approaches to PN analysis

- Reachable state space:
  - Direct or symbolic representation
  - Full or reduced state space (e.g. stubborn set method)

  in both cases knowledge of Petri net structural relations (e.g. conflicts) helps efficiency
- Unfolding the Petri net graph into an acyclic branching graph (occurrence net), with partial ordering between events and conditions and:
  - Considering a finite prefix of the unfolding which covers all reachable states and contains enough information for properties to be verified

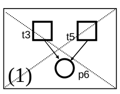
Occurrence nets



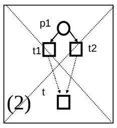
Occurrence nets

- The occurrence net of a PN *N* is a labelled (with names of the places and transitions of *N*) net (possibly infinite!) which is:
  - Acyclic
  - Contains no backward conflicts (1)
  - No transition is in self-conflict (2)
  - No twin transitions (3)
  - Finitely preceded (4)

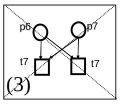
NO!



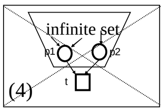
NO!



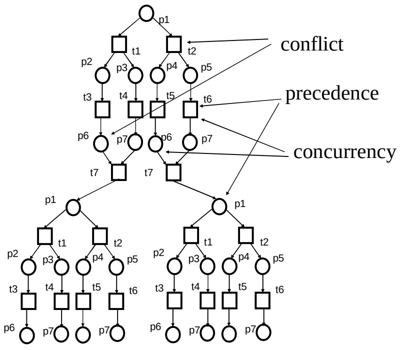
NO!



NO!

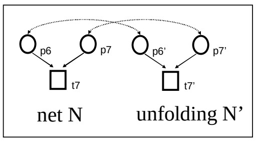


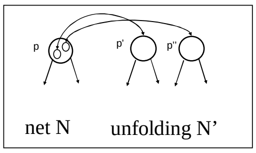
Relations in occurrence nets



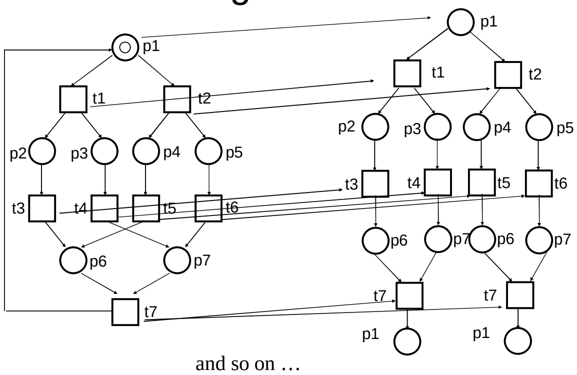
Unfolding of a PN

- The unfolding of Petri net *N* is a maximal labelled occurrence net (up to isomorphism) that preserves:
  - one-to-one correspondence (bijection) between the predecessors and successors of transitions with those in the original net
  - bijection between min places and the initial marking elements (which is multi-set)



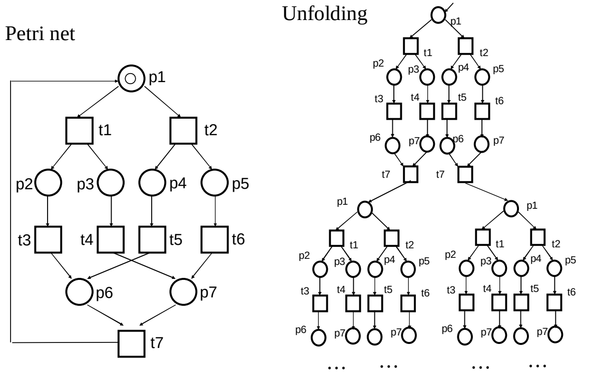


Unfolding construction

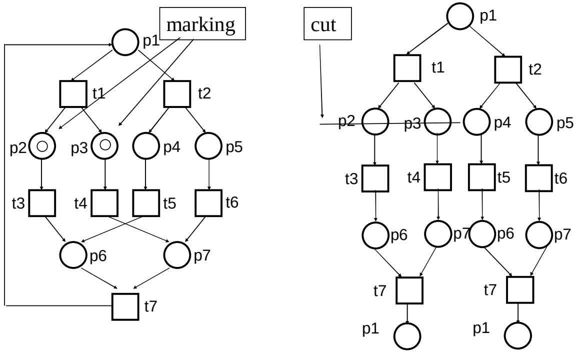


and so on …

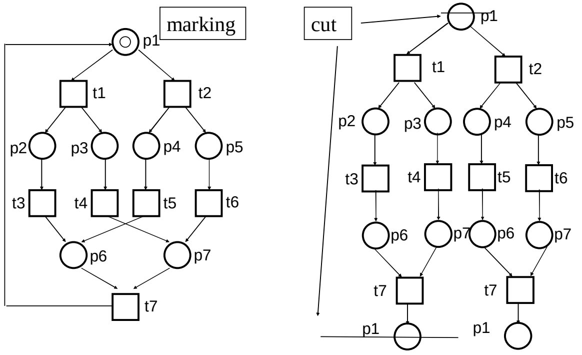
Unfolding construction



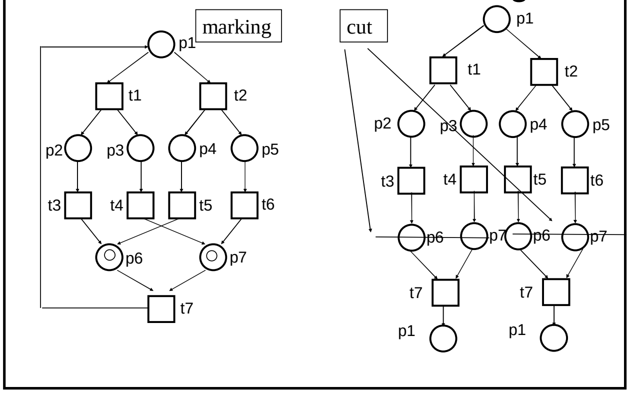
Petri net and its unfolding



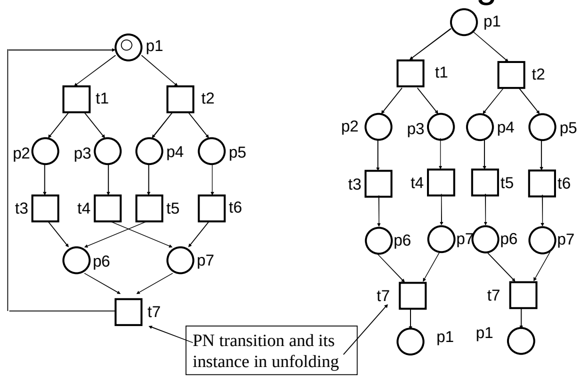
Petri net and its unfolding



Petri net and its unfolding

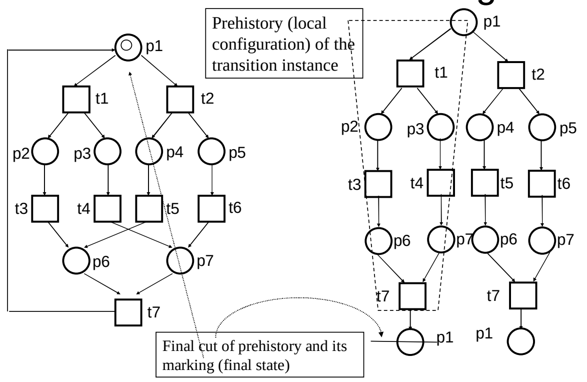


Petri net and its unfolding



PN transition and its instance in unfolding

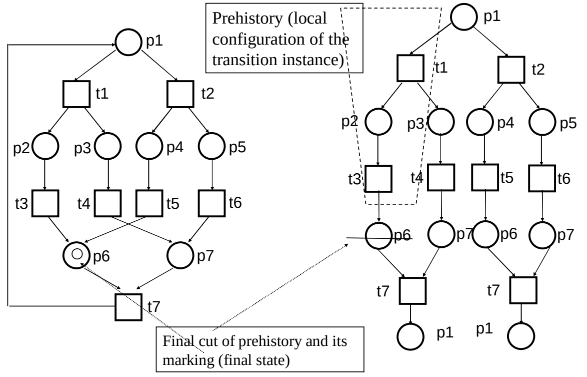
Petri net and its unfolding



Prehistory (local configuration) of the transition instance

Final cut of prehistory and its marking (final state)

Petri net and its unfolding



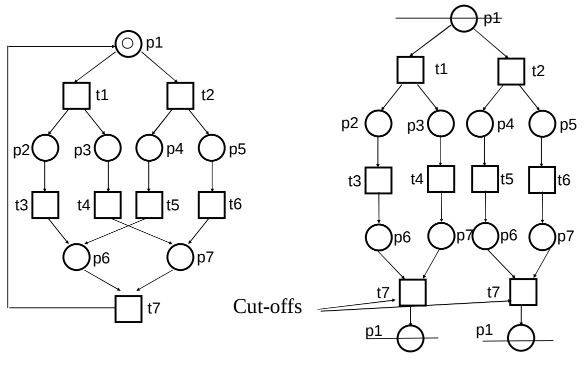
Prehistory (local configuration of the transition instance)

Final cut of prehistory and its marking (final state)

Truncation of unfolding

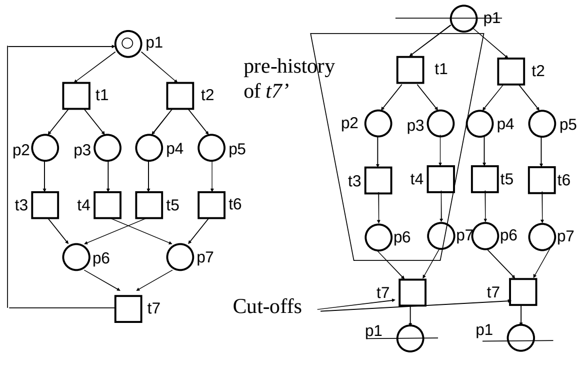
- At some point of unfolding the process begins to repeat parts of the net that have already been instantiated
- In many cases this also repeats the markings in the form of cuts
- The process can be stopped in every such situation
- Transitions which generate repeated cuts are called cut-off points or simply *cut-offs*
- The unfolding truncated by cut-off is called *prefix*

Cutoff transitions



Cut-offs

Cutoff transitions



pre-history of *t7'*

Cut-offs

Prefix Construction Algorithm

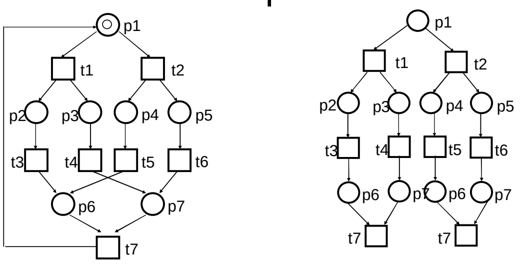
```
Proc Build prefix (N =<P,T,F,M0>)
        Initialise N' with instances of places in M0
        Initialise Queue with instances of t enabled at M0
        while Queue is not empty do
                Pull t' from Queue
                if t' is not cutoff then do
                        Add t' and succ(t') to N'
                for each t in T do
                        Find unused set of mutually concurrent
                                instances of pred(t)
                        if such set exists then do
                         Add t' to Queue in order of its prehistory size
                        end do
                end do
        end do
        return N'
end proc
```

Cut-off definition

- A newly built transition instance *t1'* in the unfolding is a cut-off point if there exists another instance *t2'* (of possibly another transition) whose:
  - Final cut maps to the same marking is the final cut of *t1'*, and
  - The size of prehistory (local configuration) of *t2'* is strictly greater than that of *t1'*

  [McMillan, 1992]
- Initial marking and its min-cut are associated with an imaginary "bottom" instance (so we can cut-off on *t7* in our example)

Finite prefix



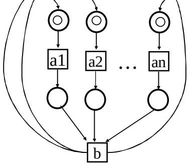
For a bounded PN the finite prefix of its unfolding contains all reachable markings

[K. McMillan]

Complexity issues

- The prefix covers all reachable markings of the original net but the process of prefix construction does not visit all these markings
- Only those markings (sometimes called *Basic Markings*) are visited that are associated with the final cuts of the local configurations of the transition instances
- These markings are analogous to primes in an algebraic lattice
- The (time) complexity of the algorithm is therefore proportional to the size of the unfolding prefix
- For highly concurrent nets this gives a significant gain in efficiency compared to methods based on the reachability graph

Size of Prefix



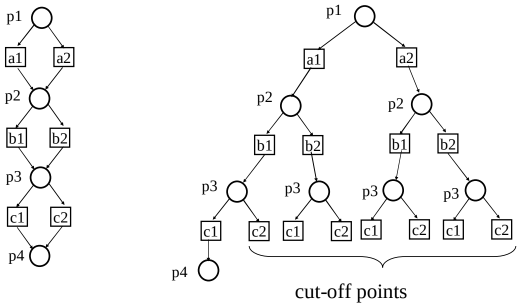
The size of the prefix for this net is *O(n)* – same as that of the original net while the size of the reachability graph is *O(2ⁿ)*

This is however not always true and the size depends on:

•the structure and class of the net, and

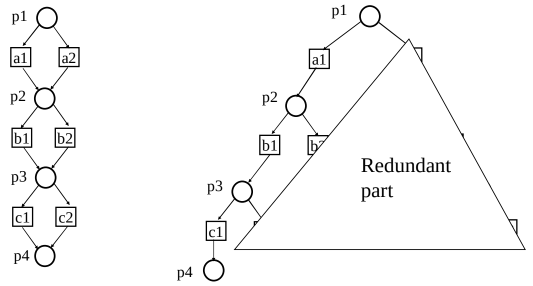
•initial marking

Size of Prefix

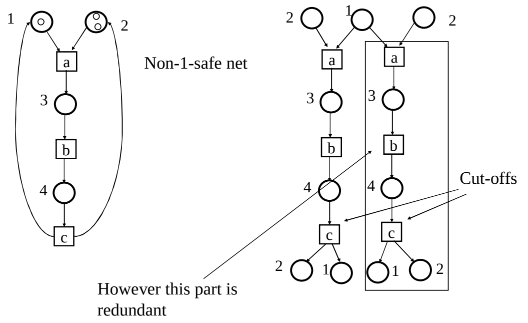


cut-off points

Size of Prefix



Size of Prefix



Cut-off Criteria

- McMillan's cutoff criterion, based on the size of pre-history, can be too strong
- A weaker criterion, based only on the matching of the final cuts, was proposed by Esparza, Vogler, and Römer
  - It uses a *total* (lexicographical) *order* on the transition set (when putting them into *Queue*)
  - It can be *only* applied to *1-safe nets* because for non-1-safe nets such a total order cannot be established (main reason auto-concurrency of instances of the same transition!)
- Unfolding k-safe nets can produce a lot of redundancy

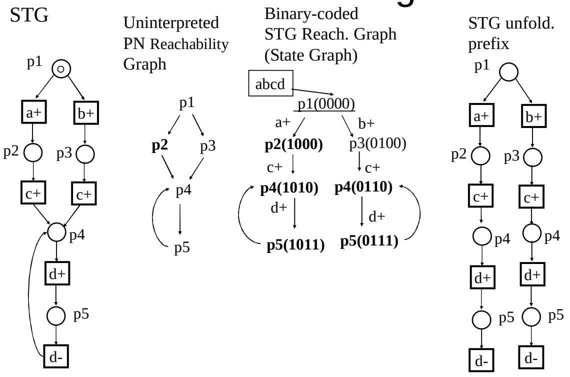
Property analysis

- A model-checker to verify a CTL formula (defined on place literals) has been built (Esparza) within the PEP tool (Hildesheim/Oldenburg)
- Various standard properties, such as k-boundedness, 1-safeness, persistency, liveness, deadlock freedom have special algorithms, e.g.:
  - Check for 1-safeness is a special case of auto-concurrency (whether a pair of place instances exist that are mutually concurrent – can be done in polynomial time)
  - Similar is a check for persistency of some transition (analysis of whether it is in immediate conflict with another transition)
  - Check for deadlock is exponential (McMillan) – involves enumeration of configurations (non-basic markings), however efficient linear-algebraic techniques have recently been found by Khomenko and Koutny (CONCUR'2000)

STG Unfolding

- Unfolding an interpreted Petri net, such as a Signal Transition Graph, requires keeping track of the interpretation – each transition is a change of state of a signal, hence each marking is associated with a *binary state*
- The prefix of an STG must not only "cover" the STG in the Petri net (reachable markings) sense but must also be complete for analysing the implementability of the STG, namely: consistency, output-persistency and Complete State Coding

STG Unfolding

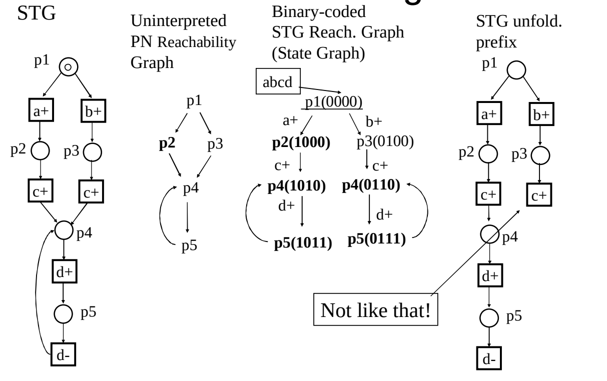

STG

Uninterpreted PN Reachability Graph

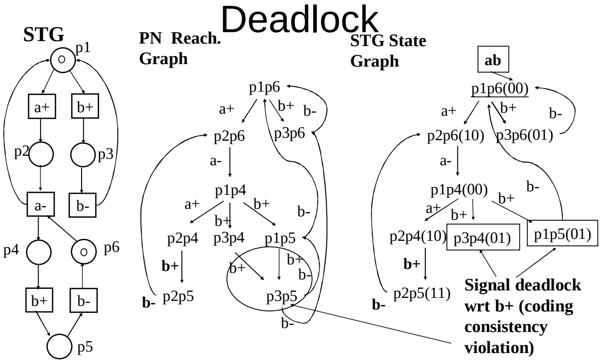
Binary-coded STG Reach. Graph (State Graph)

STG unfold. prefix

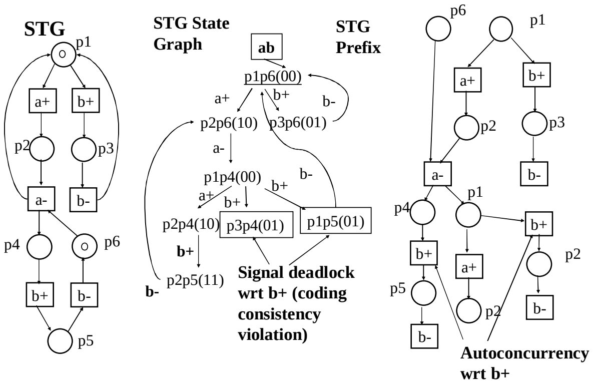
STG Unfolding



Consistency and Signal Deadlock



Signal Deadlock and Autoconcurrency



Verifying STG implementability

- Consistency – by detecting signal deadlock via *autoconcurrency* between transitions labelled with the same signal (a* || a*, where a* is a+ or a-)
- Output persistency – by detecting *conflict* relation between output signal transition a* and another signal transition b*
- *Complete State Coding* is less trivial – requires special *theory of binary covers on unfolding segments* (Kondratyev et.al.)

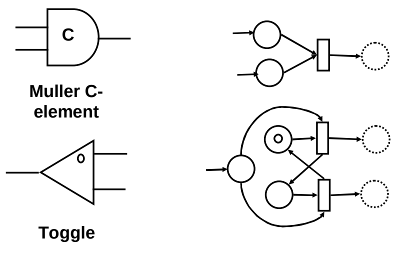
Experimental results (from Semenov)

Name	States	Versify		PUNT		
		Verif.only	Total	Trans.	Places	Time
c-elem	64	0.01	0.11	7	12	0.07
chu172	768	0.02	0.26	13	14	0.11
espinalt-ba	15360	0.07	0.74	13	17	0.12
espinalt-gc	27648	0.1	0.83	25	30	0.17
fair-arb-sg	1280	0.09	0.8	32	33	0.51
josepm	45056	0.7	0.72	21	29	0.12
master-rea	3.45E+07	0.39	7.4	51	78	0.37
t1	618496	2.65	8.97	67	104	2.87
irred.no1tc	41472	0.19	0.93	6	10	0.07
…	…	…	…	…	…	…
TOTAL		4.37	26.98			6.13

Example with inconsistent STG: PUNT quickly detects a signal deadlock "on the fly" while Versify builds the state space and then detects inconsistent state coding

Analysis of Circuit Petri Nets

Event-driven elements **Petri net equivalents**



Muller C-element

Toggle

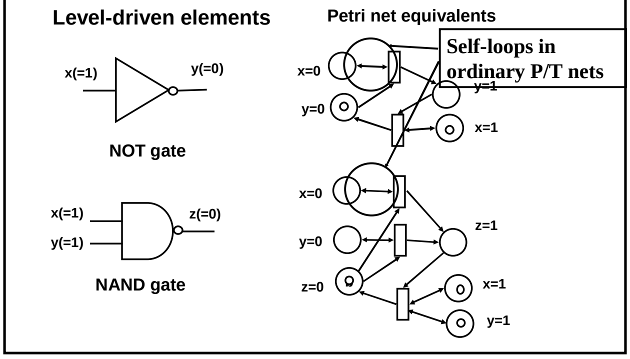
Analysis of Circuit Petri Nets

- Petri net models built for event-based and level-based elements, together with the models of the environment can be analysed using the STG unfolding prefix
- The possibility of hazards is verified by checking either 1-safeness (for event-based) or persistency (for level-based) violations

Experimental results (from Kondratyev)

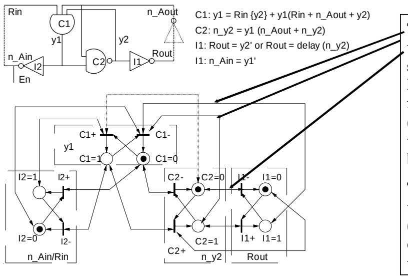
example	#stages	#places	#trans	#states	BDD			Prefix		
					peak size	final size	time	#places	#trans	time
philosoph	20	140	100	2.20E+13	none	3091	10	140	100	1
philosoph	40	280	200	2.90E+19	none	251839	455	280	200	1
philosoph	50	350	250	too many	none	1870847	>4hrs	350	250	1
philosoph	60	420	300	too many	none	none	none	420	300	1
muller-pipe	30	120	60	6.00E+07	7897	4784	132	490	240	1
muller-pipe	45	180	90	6.90E+11	23590	10634	740	1035	510	2
muller-pipe	60	240	120	8.40E+15	53446	18788	3210	1780	880	4
dme-arbiter	20	81	80	2.20E+07	1688	1688	11	81	80	1
dme-arbiter	40	161	160	4.50E+13	6568	6568	101	161	160	1
dme-arbiter	60	241	240	7.00E+19	14648	14648	342	241	240	1

Circuit Petri Nets



Self-loops in ordinary P/T nets

Circuit Petri nets



C1: y1 = Rin {y2} + y1(Rin + n_Aout + y2)

C2: n_y2 = y1 (n_Aout + n_y2)

I1: Rout = y2' or Rout = delay (n_y2)

I1: n_Ain = y1'

The meaning of these numerous self-loop arcs is however different from self-loops (which take a token and put it back)

These should be test or read arcs (without consuming a token)

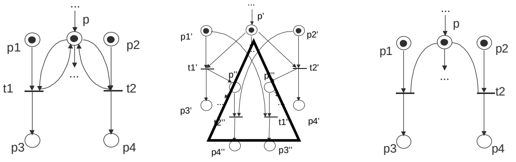
From the viewpoint of analysis we can disregard this semantic discrepancy (it does not affect reachability graph properties!) and use ordinary PN unfolding prefix for analysis, BUT …

Unfolding Nets with Read Arcs

PN with self-loops

Unfolding with self-loops

Unfolding with read arcs



Combinatorial explosion due to splitting the self-loops

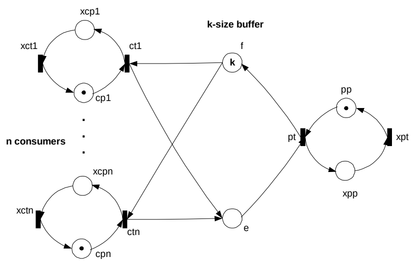
Unfolding k-safe nets

- How to cope with k-safe (k>1) nets and their redundancy
- Such nets are extremely useful in modelling various hardware components with:
  - Buffers of finite capacity
  - Counters of finite modulo count
- McMillan’s cutoff condition is too strong (already much redundancy)
- EVR’s condition is too weak – cannot be applied to k-safe nets

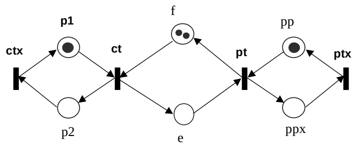
Proposed solution: introduce total order on tokens, e.g. by applying FIFO discipline of their arrival-departure (work with F. Alamsyah et al.)

Unfolding k-safe Nets

Example: producer-consumer



Unfolding k-safe Nets



Consider the case:

n=1 consumer

k=2-place buffer

Three techniques have been studied (by F. Alamsyah):

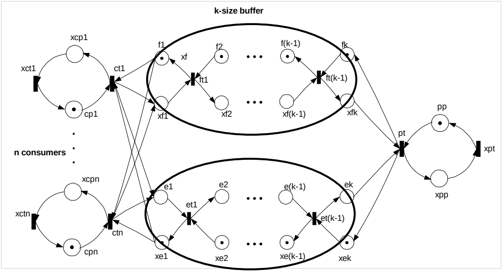
(1) Direct prefix using McMillan's cutoff criterion

(2) Unfolding the explicitly refined (with FIFO buffers) 1-safe net (using EVR cutoff criterion)

(3) Unfolding the original, unrefined net with FIFO semantics

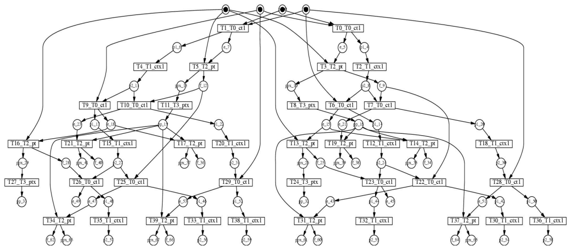
Unfolding k-safe Nets

Approach (2) for refining FIFO places into 1-safe subnets



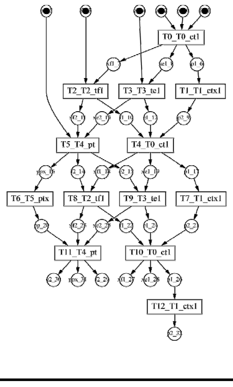
Unfolding k-safe Nets

(1) Direct unfolding prefix (using McMillan's cutoff)



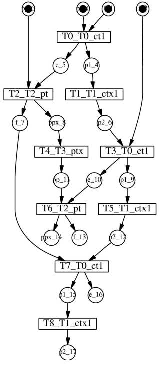
Unfolding k-safe Nets

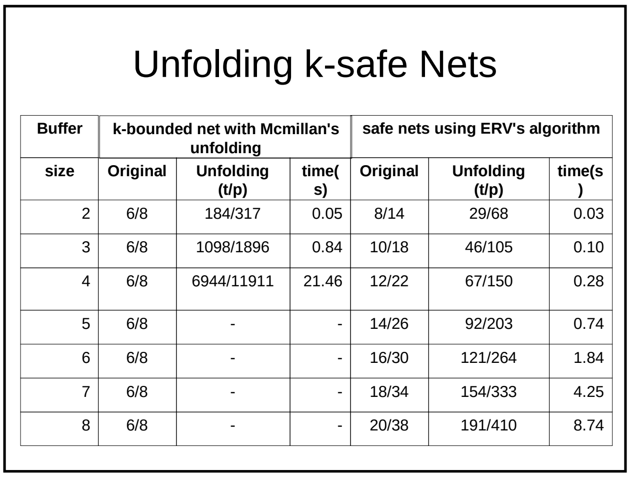
(2) Unfolding the explicitly refined (with FIFO buffers) 1-safe net (using EVR's cutoff)



Unfolding k-safe Nets

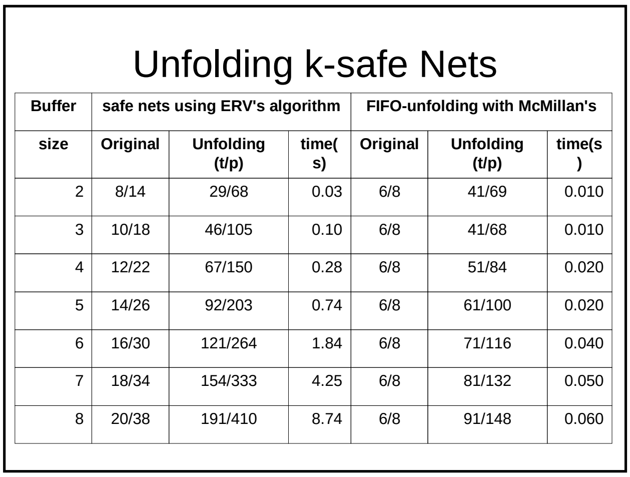
(3) Unfolding the original, unrefined net with FIFO semantics





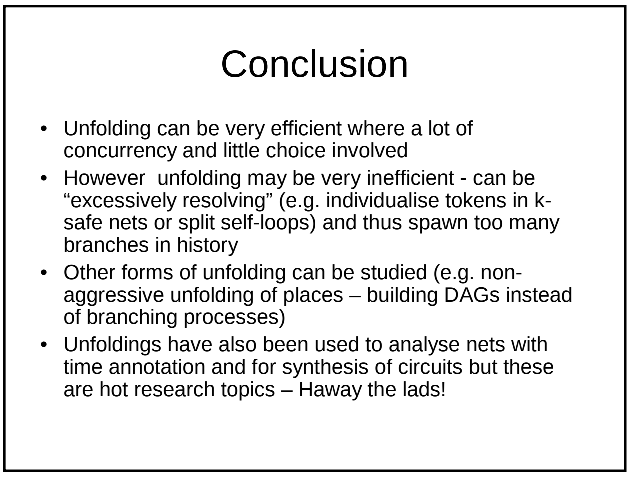
Unfolding k-safe Nets

Buffer	k-bounded net with Mcmillan's unfolding			safe nets using ERV's algorithm		
size	Original	Unfolding (t/p)	time(s)	Original	Unfolding (t/p)	time(s)
2	6/8	184/317	0.05	8/14	29/68	0.03
3	6/8	1098/1896	0.84	10/18	46/105	0.10
4	6/8	6944/11911	21.46	12/22	67/150	0.28
5	6/8	-	-	14/26	92/203	0.74
6	6/8	-	-	16/30	121/264	1.84
7	6/8	-	-	18/34	154/333	4.25
8	6/8	-	-	20/38	191/410	8.74



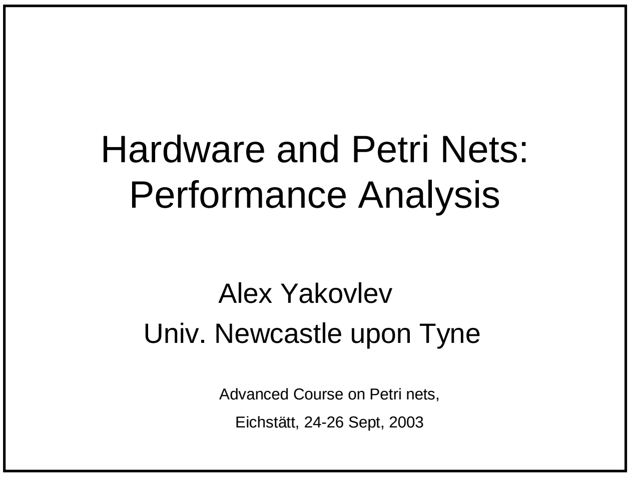
Unfolding k-safe Nets

Buffer	safe nets using ERV's algorithm			FIFO-unfolding with McMillan's		
size	Original	Unfolding (t/p)	time(s)	Original	Unfolding (t/p)	time(s)
2	8/14	29/68	0.03	6/8	41/69	0.010
3	10/18	46/105	0.10	6/8	41/68	0.010
4	12/22	67/150	0.28	6/8	51/84	0.020
5	14/26	92/203	0.74	6/8	61/100	0.020
6	16/30	121/264	1.84	6/8	71/116	0.040
7	18/34	154/333	4.25	6/8	81/132	0.050
8	20/38	191/410	8.74	6/8	91/148	0.060



Conclusion

- Unfolding can be very efficient where a lot of concurrency and little choice involved
- However unfolding may be very inefficient - can be “excessively resolving” (e.g. individualise tokens in k-safe nets or split self-loops) and thus spawn too many branches in history
- Other forms of unfolding can be studied (e.g. non-aggressive unfolding of places – building DAGs instead of branching processes)
- Unfoldings have also been used to analyse nets with time annotation and for synthesis of circuits but these are hot research topics – Haway the lads!



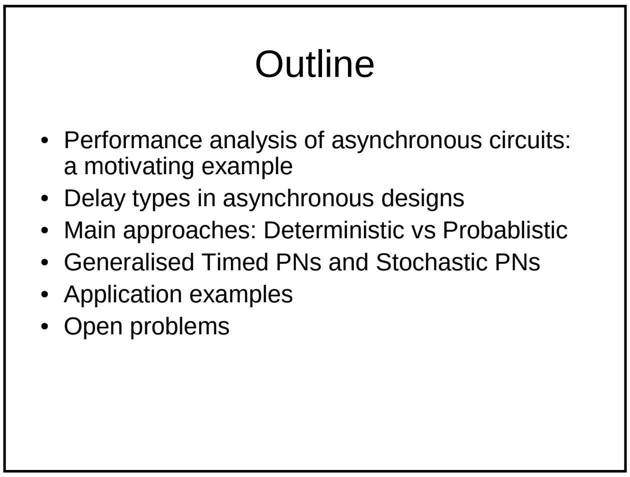
Hardware and Petri Nets: Performance Analysis

Alex Yakovlev

Univ. Newcastle upon Tyne

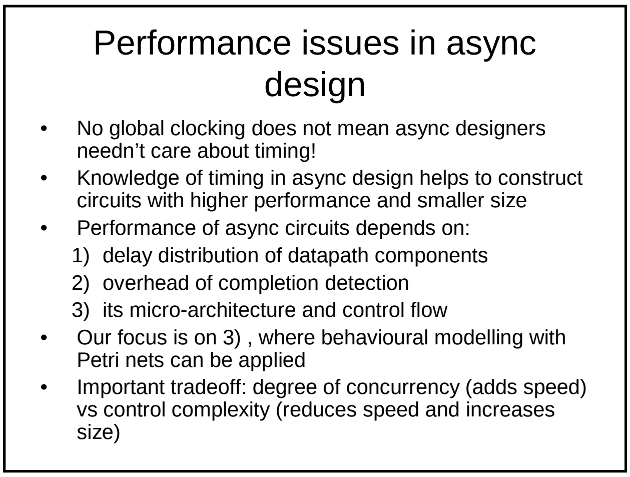
Advanced Course on Petri nets,

Eichstätt, 24-26 Sept, 2003



Outline

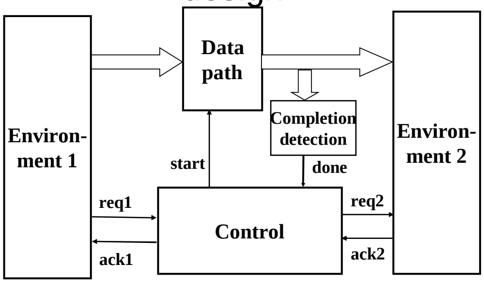
- Performance analysis of asynchronous circuits: a motivating example
- Delay types in asynchronous designs
- Main approaches: Deterministic vs Probablistic
- Generalised Timed PNs and Stochastic PNs
- Application examples
- Open problems



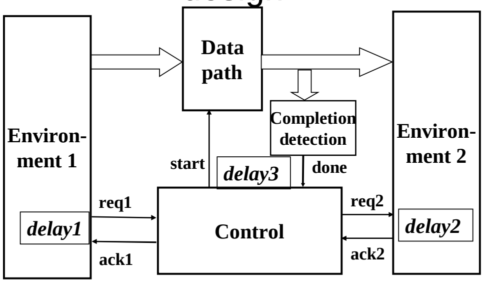
Performance issues in async design

- No global clocking does not mean async designers needn’t care about timing!
- Knowledge of timing in async design helps to construct circuits with higher performance and smaller size
- Performance of async circuits depends on:
  1) delay distribution of datapath components
  2) overhead of completion detection
  3) its micro-architecture and control flow
- Our focus is on 3) , where behavioural modelling with Petri nets can be applied
- Important tradeoff: degree of concurrency (adds speed) vs control complexity (reduces speed and increases size)

Performance issues in async design

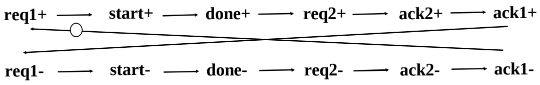


Performance issues in async design

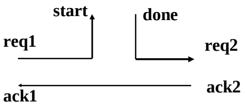


Concurrency vs Complexity

Control flow schedule:

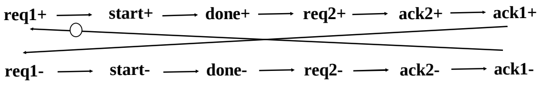


Control circuit implementation:



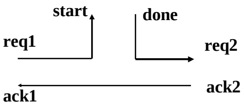
Concurrency vs Complexity

Control flow schedule: ***No concurrency!***



Control circuit implementation:

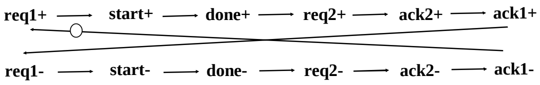
Zero complexity!



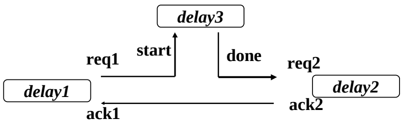
Control circuit adds minimum delay!

Concurrency vs Complexity

Control flow schedule:



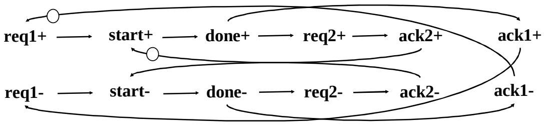
Control circuit implementation:



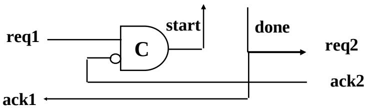
Total cycle time: 2(delay1+delay2+delay3)

Concurrency vs Complexity

Another schedule:

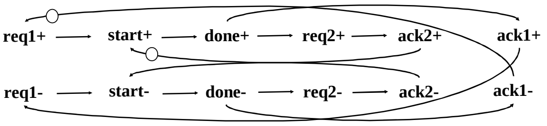


Control circuit implementation:



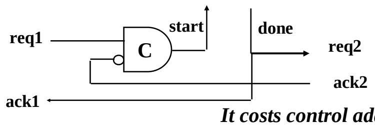
Concurrency vs Complexity

Another schedule:



Concurrency between environments

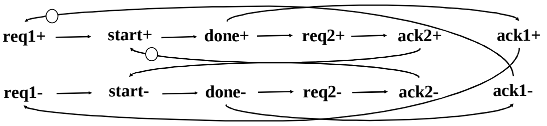
Control circuit implementation:



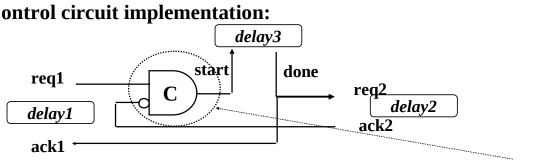
It costs control additional logic and extra delay

Concurrency vs Complexity

Another schedule:



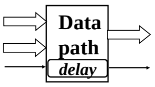
Control circuit implementation:

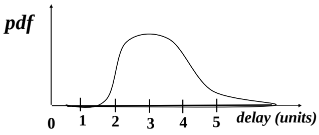


Total cycle time: 2(max(delay1,delay2)+delay3 + delayC)

Delays in async design

Data path delays **are introduced by: operational blocks (e.g adders, comparators, shifters, multiplexers etc.) and their completion logic, buffer registers, switches, buses etc.**

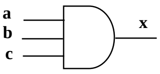


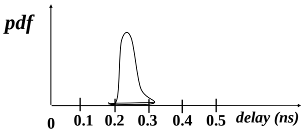


These delays are usually distributed in a way specific to the unit’s function and data domain, e.g. delay in a ripple-carry adder is dependent on the length of the carry chain (can vary from from 1 to N, dependent on the values of operands), with the mean at log(N)

Delays in async design

Control logic delays **are introduced by logic gates (with good discrete behavioural approx.) and wires (often taken as negligible in the past, but now this is too optimistic)**



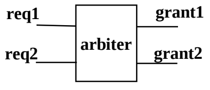


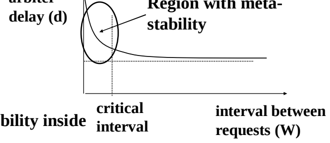
Gate (switching) delays are usually taken as either deterministic or distributed uniformly or normally around some mean with small deviation.

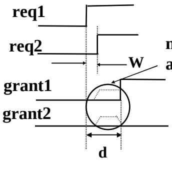
For greater accuracy inherent gate delay may sometimes be seen dependent on the state (say transition 0-1 on x may take longer when a=b=1 and c goes 0-1 than when a goes 0-1 when b=c=1)

Delays in async design

Control delays **may also be introduced by non-logic (internally analogue) components, such as arbiters and synchronisers which may exhibit meta-stable nondeterministic behaviour**







meta-stability inside arbiter

Arbiter delay is state-dependent, it is exponentially distributed if both inputs arrive with a very short (less than critical interval) – This effect may often be ignored in average performance (but not in hard-real time!) analyses due to low frequency of meta-stable condition

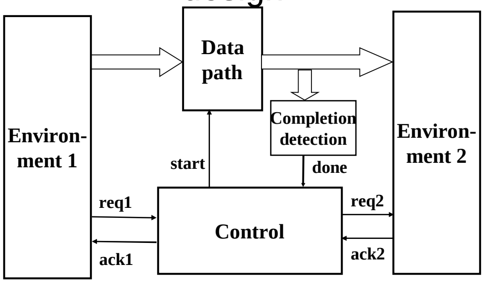
Delays in async design

Environment delays **may be introduced by:**

• some known or partially known design components, like data path elements or controllers at the same level of abstraction (with deterministic or data specific pdf/pmf), or

• unknown parts of the system, which can be treated as “clients” (exponential distribution is often a good approximation)

Performance issues in async design



Performance parameters

Asynchronous circuits are often characterised by:

- average response/cycle time or throughput wrt some critical interfaces (e.g. throughput/cycle time at the req1/ack1 interface)
- latency between a pair of critical signals or parts (e.g. latency between req1 and req2)

These could be obtained through computation of time separation of events (TSEs)

At higher levels, they can be characterised by average resource utilisation (e.g. useful for estimating power consumption) or quantitative "versions" of system behaviour properties, e.g. fairness, freshness

Main approaches to perf. analysis

Two methodologically different approaches:

- Deterministic (delay information known in advance), sometimes the element of unknown is represented by delay intervals. Performance values are computed precisely (even if within lower/upper bounds or by average values). Good for hard-real time systems or for detailed, low level circuit designs where absolute performance parameters are important
- Probabilistic (delay information defined by distribution functions, standard or arbitrary pmf). Performance is estimated only approximately, mostly to assess and compare alternative design solutions at early stages of system design, where relative performance factors are needed. They may also be useful for guiding synthesis

Deterministic approach

- Timed Petri nets - early models by Ramchandani (MIT-TR, 1974) and Ramamoorthy&Ho (IEEE Trans SE1980)

Key result (for marked graphs):

$$C = \max\left\{\frac{T_k}{N_k} : k = 1,2,\ldots,q\right\}$$

where

C – average cycle time (lower) bound

T_k – sum of execution times of transitions in cycle k

N_k – total number of tokens in cycle k

q – number of cycles in the net

Proof based on:

(1) No. of tokens in every cycle of an MG is constant (Commoner et al)

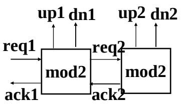
(2) All transitions in an MG have the same cycle time

A polynomial algorithm for verification of $CN_k - T_k \geq 0$ condition (based on Floyd algorithm); see also Nielsen&Kishinevsky(DAC'94)

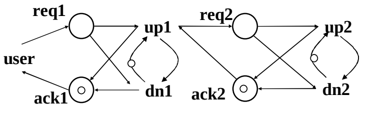
Method can also be used for safe persistent nets but proved NP-complete for general nets

Deterministic cycle time

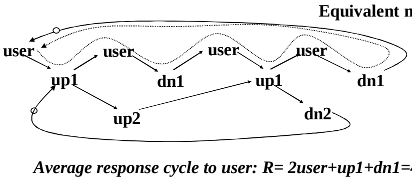
Pipeline counter (frequency divider)



Safe-persistent net



Equivalent marked graph



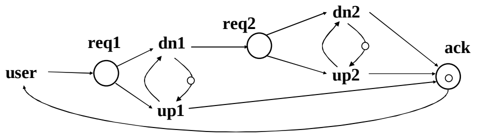
Critical cycle:

C= 4user+2up1+2dn1=8

Average response cycle to user: R= 2user+up1+dn1=4
(Remains constant regardless of the number of stages!)

Deterministic cycle time

Normal sequential counter



Critical cycle:

C= 4user+2up1+2dn1+up2+dn2=10

Exercise: "unfold" this safe-persistent net into a marked graph and check its cycle time

Average response cycle to user:

C= 10/4= 2.5 (depends on the number of stages)

Deterministic cycle time

Exercise 1:

Find the average cycle time for the ring of five Muller C-elements with inverters (assume each gate to have a delay of 1 unit)

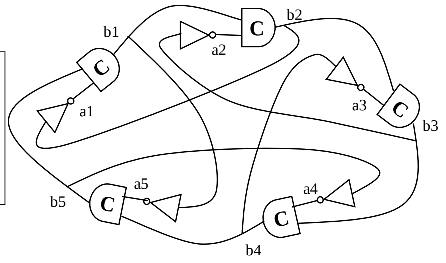
Initial state:

ai=1, i=1,…,5

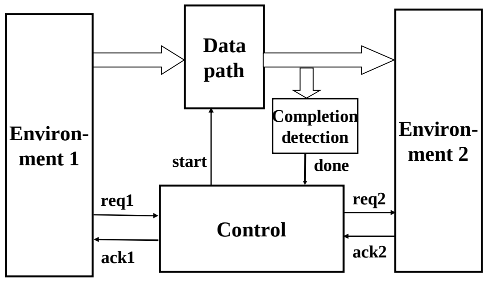
bj=0, j=1,…,4

b5=1

b1=0 is enabled



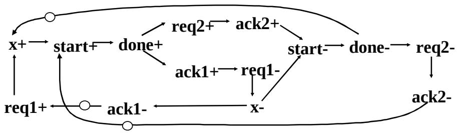
Deterministic Cycle time



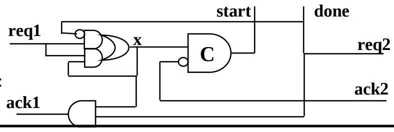
Exercise 2: Deterministic cycle time

Estimate the effect of additional decoupling between Environments 1 and 2 due to “flag (CSC) signal” x (by finding the critical cycle time using the assumption that delays in the environment are larger in the setting phase than in the resetting and much larger than the gate delay) and observe the trade-off between concurrency and complexity

STG:



Circuit implementation:



Probabilistic approach

Sources of non-determinism:

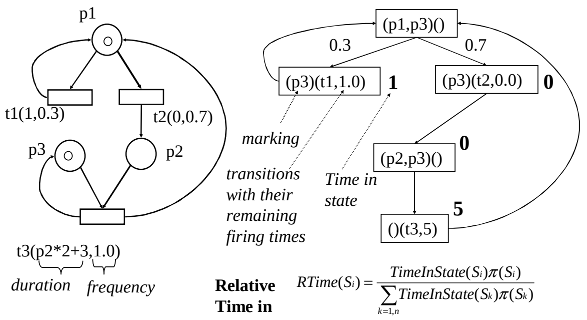
1) Environment may offer choice (e.g. Read/Write modes in VME bus interface, instruction decoding in a CPU) => probabilistic choice b/w transitions (cf. frequencies in TPNs)
2) Data path or environment delays may have stochastic nature (e.g. delay distribution in carry-chain, or user think time distribution)
3) Gate delays may be modelled using specific pdf/pmf's to allow for uncertainty in low-level implementation (layout and technology parameter variations)

2) and 3) => firing time distributions in Stochastic Petri nets (SPNs)

Generalised TPNs(GTPNs)

- Probabilistic choice was introduced in TPN by Zuberek (CompArchSymp80), Razouk&Phelps (ParallelProcConf84), and in GTPN by Holliday&Vernon (IEEE Trans SE-13,87)
- GTPN transitions have deterministic durations (though can be made state-dependent and with discrete geometric distribution)
- Analysis of GTPN models is based on:
  - (1) constructing the reachability graph with transition probabilities (due to choice with frequencies) between markings, generating a discrete time Markov chain (DTMC), and
  - (2) computing performance measures from DTMC analysis

GTPN



Relative Time in State:

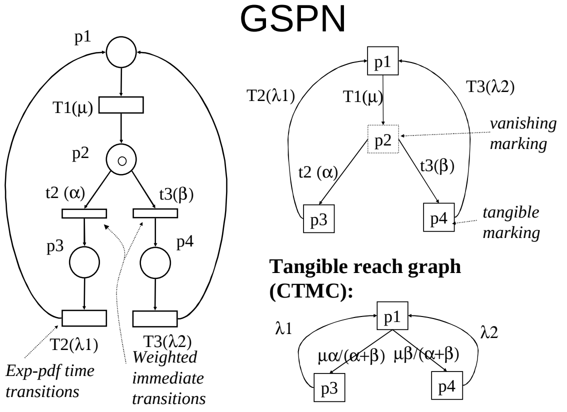
$$RTime(S_i) = \frac{TimeInState(S_i)\pi(S_i)}{\sum_{k=1,n} TimeInState(S_k)\pi(S_k)}$$

where $\pi(S_i)$ – stationary probability of being in state S_i (obtained from MC analysis)

Generalised Stochastic PNs

- Transitions with probabilistic (continuous) firing time were introduced in Stochastic Petri nets (SPNs)by Molloy (IEEE TC-31,82) and in GSPN by Marsan, Balbo&Conte (ACM TCS-2,84)
- Firing time can either be zero (immediate transitions) or exponential distributed (for Markovian properties of the reachability graph); Immed. transitions have higher priority
- More extensions have been introduced later leading to Generally Distributed Timed Transitions SPNs (GDTT-SPN) – see Marsan, Bobbio&Donatelli's tutorial in Adv.Lectures 98
- Analysis of GSPN based on:
  - (1) constructing a reachability graph with transition rates, thus generating a continnuous time Markov Chain, and
  - (2) computing performance measures from CTMC analysis

GSPN



Comparison b/w GTPN and GSPN

Modelling /Analysis Feature	GTPN	GSPN
Conflict resolution	Static (conflicts resolved before firing – using firing frequencies). Good to model free (environment) choice	Dynamic (competing transition delays – transition which fires first wins conflict). Good to model races and arbitration in hardware
Probability/ rate assignment	Probabilities are assigned according to transitions frequencies independent of transition delays	Probabilities/rates are assigned according to competing transition delays for non-immediate transitions
Firing durations	Arbitrary non-negative reals plus geometric holding times	Extended SPN (ESPN) can have arb. holding times but at the cost of restricted reach graphs
Complexity	Due to inherent complexity of deterministic delays, their state space is larger	Discrete-time SPNs can have large state spaces when deterministic holding times are used

What is needed for async hardware?

Asynchronous circuit modelling requires:

- both deterministic and stochastic delay modelling,
- stochastic static ("free-choice") and dynamic (with races) conflict resolution
- competing (with races) transitions with deterministic timing

Any idea of a tractable model with these features?

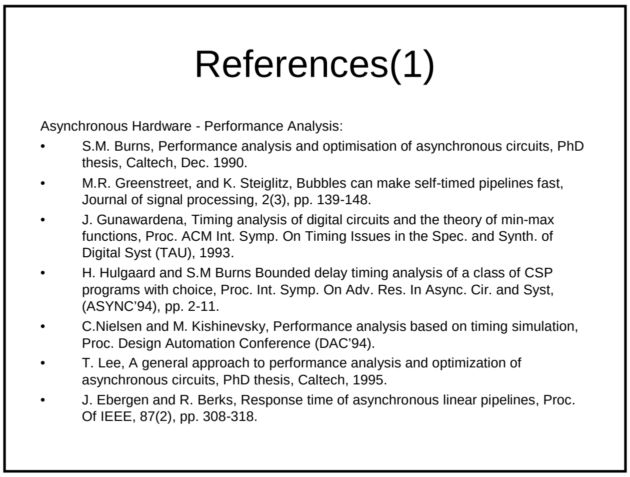
Recent application examples

These are examples of using PNs in analytic and simulation environments:

- Use of unfoldings (tool PUNT) and SPNs (tool UltraSan) for performance estimation of a CPU designed with PNs (Semenov,etal, IEEEMicro,1997)
- Multi-processor, multi-threaded architecture modelling using TPNs (Zuberek, HWPN'99)
- Response time (average bounds) analysis using STPNs and Monte-Carlo, for Instruction length Decoder; developed tool PET (Xie&Beerel, HWPN'99)
- Analysis of data flow architectures using tool ExSpect (Witlox etal, HWPN'99)
- Modelling and analysis of memory systems using tool CodeSign (Gries, HWPN'99)
- Superscalar processor modelling and analysis using tool Design/CPN (Burns,etal,J.ofRT,2000)
- SPN modelling and quantification of fairness in arbiter analysis using tool GreatSPN (Madalinski,etal,UKPEW'00)

Conclusions

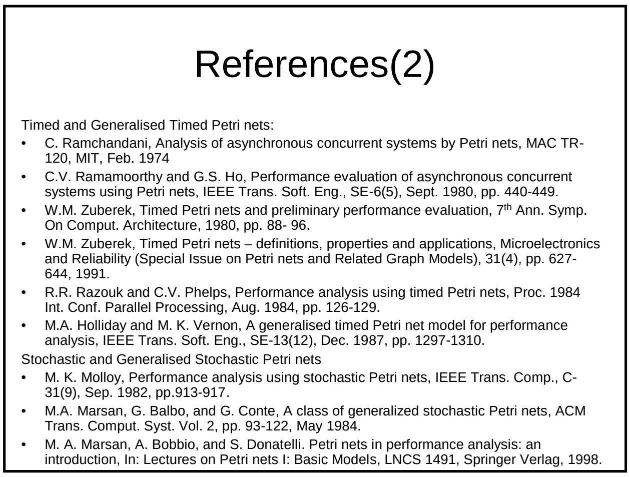
- Asynchronous circuits, whether speed-independent or with timing assumptions/constraints, require flexible and efficient techniques for performance analysis
- The delay models cover both main types: deterministic, stochastic (with different pdf/pmf's) and must allow for races; conflicts both static and dynamic
- Clearly two different levels of abstraction need to be covered – logic circuit (STG) level and abstract behaviour (LPN) level; those often have different types of properties to analyse
- The number of async IP cores (for Systems-on-Chip) are on the increase in the near future, so big help from performance analysis is urgently needed to evaluate these new core developments



References(1)

Asynchronous Hardware - Performance Analysis:

- S.M. Burns, Performance analysis and optimisation of asynchronous circuits, PhD thesis, Caltech, Dec. 1990.
- M.R. Greenstreet, and K. Steiglitz, Bubbles can make self-timed pipelines fast, Journal of signal processing, 2(3), pp. 139-148.
- J. Gunawardena, Timing analysis of digital circuits and the theory of min-max functions, Proc. ACM Int. Symp. On Timing Issues in the Spec. and Synth. of Digital Syst (TAU), 1993.
- H. Hulgaard and S.M Burns Bounded delay timing analysis of a class of CSP programs with choice, Proc. Int. Symp. On Adv. Res. In Async. Cir. and Syst, (ASYNC'94), pp. 2-11.
- C.Nielsen and M. Kishinevsky, Performance analysis based on timing simulation, Proc. Design Automation Conference (DAC'94).
- T. Lee, A general approach to performance analysis and optimization of asynchronous circuits, PhD thesis, Caltech, 1995.
- J. Ebergen and R. Berks, Response time of asynchronous linear pipelines, Proc. Of IEEE, 87(2), pp. 308-318.



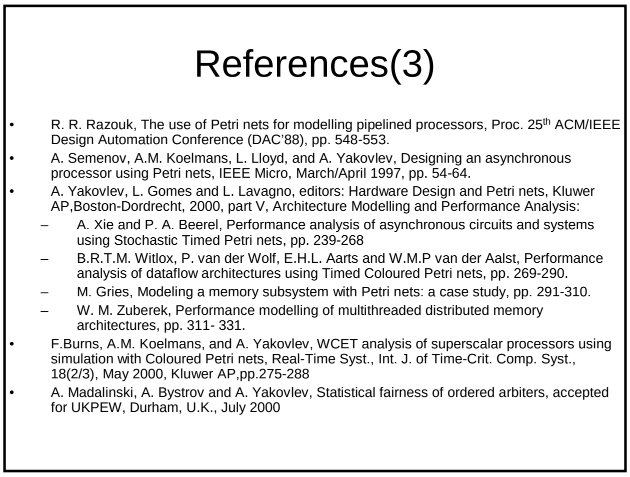
References(2)

Timed and Generalised Timed Petri nets:

- C. Ramchandani, Analysis of asynchronous concurrent systems by Petri nets, MAC TR-120, MIT, Feb. 1974
- C.V. Ramamoorthy and G.S. Ho, Performance evaluation of asynchronous concurrent systems using Petri nets, IEEE Trans. Soft. Eng., SE-6(5), Sept. 1980, pp. 440-449.
- W.M. Zuberek, Timed Petri nets and preliminary performance evaluation, 7th Ann. Symp. On Comput. Architecture, 1980, pp. 88- 96.
- W.M. Zuberek, Timed Petri nets – definitions, properties and applications, Microelectronics and Reliability (Special Issue on Petri nets and Related Graph Models), 31(4), pp. 627-644, 1991.
- R.R. Razouk and C.V. Phelps, Performance analysis using timed Petri nets, Proc. 1984 Int. Conf. Parallel Processing, Aug. 1984, pp. 126-129.
- M.A. Holliday and M. K. Vernon, A generalised timed Petri net model for performance analysis, IEEE Trans. Soft. Eng., SE-13(12), Dec. 1987, pp. 1297-1310.

Stochastic and Generalised Stochastic Petri nets

- M. K. Molloy, Performance analysis using stochastic Petri nets, IEEE Trans. Comp., C-31(9), Sep. 1982, pp.913-917.
- M.A. Marsan, G. Balbo, and G. Conte, A class of generalized stochastic Petri nets, ACM Trans. Comput. Syst. Vol. 2, pp. 93-122, May 1984.
- M. A. Marsan, A. Bobbio, and S. Donatelli. Petri nets in performance analysis: an introduction, In: Lectures on Petri nets I: Basic Models, LNCS 1491, Springer Verlag, 1998.



References(3)

- R. R. Razouk, The use of Petri nets for modelling pipelined processors, Proc. 25th ACM/IEEE Design Automation Conference (DAC'88), pp. 548-553.
- A. Semenov, A.M. Koelmans, L. Lloyd, and A. Yakovlev, Designing an asynchronous processor using Petri nets, IEEE Micro, March/April 1997, pp. 54-64.
- A. Yakovlev, L. Gomes and L. Lavagno, editors: Hardware Design and Petri nets, Kluwer AP,Boston-Dordrecht, 2000, part V, Architecture Modelling and Performance Analysis:
  - A. Xie and P. A. Beerel, Performance analysis of asynchronous circuits and systems using Stochastic Timed Petri nets, pp. 239-268
  - B.R.T.M. Witlox, P. van der Wolf, E.H.L. Aarts and W.M.P van der Aalst, Performance analysis of dataflow architectures using Timed Coloured Petri nets, pp. 269-290.
  - M. Gries, Modeling a memory subsystem with Petri nets: a case study, pp. 291-310.
  - W. M. Zuberek, Performance modelling of multithreaded distributed memory architectures, pp. 311- 331.
- F.Burns, A.M. Koelmans, and A. Yakovlev, WCET analysis of superscalar processors using simulation with Coloured Petri nets, Real-Time Syst., Int. J. of Time-Crit. Comp. Syst., 18(2/3), May 2000, Kluwer AP,pp.275-288
- A. Madalinski, A. Bystrov and A. Yakovlev, Statistical fairness of ordered arbiters, accepted for UKPEW, Durham, U.K., July 2000