Direct synthesis of large-scale asynchronous controllers using a Petri-net-based approach

Ivan Blunno	Politecnico di Torino
Alex Bystrov	Univ. Newcastle upon Tyne
Josep Carmona	Univ. Politècnica de Catalunya
Jordi Cortadella	Univ. Politècnica de Catalunya
Luciano Lavagno	Università di Udine
Alex Yakovlev	Univ. Newcastle upon Tyne

Outline

- ☞ Motivation
- Design flow
- Verilog HDL specification
- Petri nets and trace expressions
- Synthesis process
- Conclusion

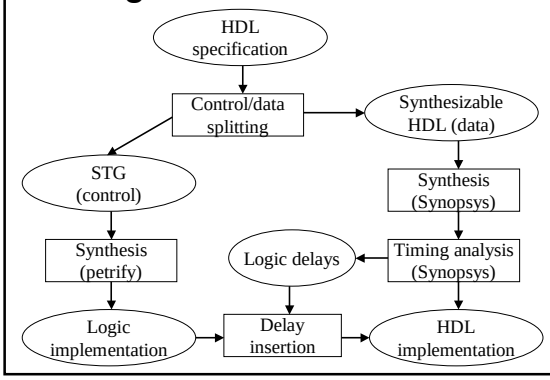
Motivation

- Language-based design key enabler to synchronous logic success
- Use HDL as single language for
  - specification
  - logic simulation and debugging
  - synthesis
  - post-layout simulation
- HDL must support multiple levels of abstraction

Motivation

- HDL generates large asynchronous controllers: need *direct synthesis*
- Guarantee an implementation
- Automatic exploration of the design space
- Benefit from existing structural methods for logic synthesis
- Benefit (at the design stage) from existing performance estimation approaches

Design flow



Design flow

- What is available?
  - simulators (no synchronous assumption…)
  - logic synthesis (from BFSM, STG, …)
  - layout (almost like synchronous…)
- What is missing?
  - translator from HDL to synthesis specification model
  - translator from synthesis implementation model to HDL

Other approaches

- Special-purpose languages
  - pros: syntax and semantics can be tailored to asynchronous Models of Computation (STG, BFSM, process algebrae)
  - cons: not familiar to designers, no standard tool support
- Examples
  - Tangram
  - Communicating Hardware Processes
  - Balsa

Our approach

- General-purpose language
  - pros: several tools available, broad user basis
  - cons: syntax and semantics oriented to gates, (not STGs or BFSMs or process algebrae)
  - need to define a subset for synthesis (full language only good for simulation)
- Choice
  - VHDL
  - Verilog [Blunno & Lavagno, ASYNC'00]

Outline

- Motivation
- Design flow
- ☞ Verilog HDL specification
- Petri nets and trace expressions
- Synthesis
- Conclusion

Asynchronous Verilog subset

- Module and signal declaration:
  - module example(a, b, c, d);
  - input a, b[7..0];
  - output c, d;
  - reg e, f, g[11..0];
- Currently only single module supported
  - always loop surrounds live behavior
  - initial block defines initialization sequence

Asynchronous Verilog subset

- Transitions:
  - input signals: wait statement
    - wait(a); ... wait (!b);
  - output signals: assignment statement
    - c = a + b;
- Each statement generates a trace expression and a datapath fragment

Asynchronous Verilog subset

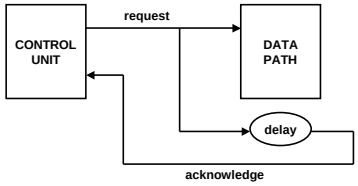
- Causality relations: Verilog statements
  - begin-end for sequencing
  - fork-join for concurrency
  - if-then-else for input choice
- Only structured mix of sequencing, concurrency and choice can be specified

Example: simple filter

```
always
begin
     wait(start);
     R = SMP * 3;
     RES = SMP * 4;
     if(b7 == 1) RES = 0;
     else begin
       if(b6 == 1) RES = 1;
     end;
     done = 1;
     wait(!start);
     done = 0;
end
```

Control-data partitioning

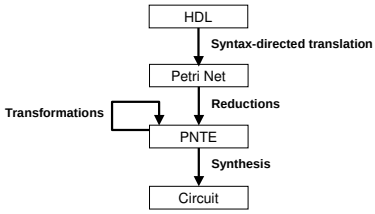
- **Splitting of asynchronous control and synchronous data path**
- **Automated insertion of bundling delays**



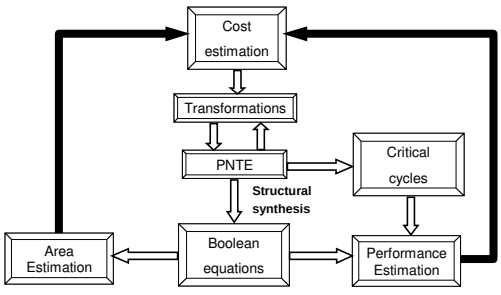
Outline

- **Motivation**
- **Design flow**
- **Verilog HDL specification**
- ☞ **Petri nets and trace expressions**
- **Synthesis**
- **Conclusion**

Controller design flow



Design flow



PNTE

- Free-choice Petri net
- Transitions are trace expressions
- Trace expressions represent well-structured event relations
  - Causality
  - Concurrency
  - Choice

Trace expressions (TE)

TE ⇨ *e*

⇨ TE ; TE

⇨ TE || TE

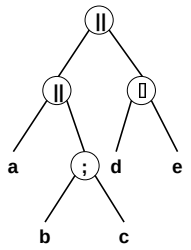
⇨ TE □ TE

⇨ • TE

trace expressions **are a subset of CCS agent expressions [Milner 80]**

Trace expressions: example

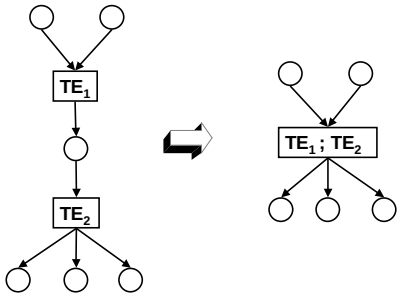
(a || (b ; c)) || (d □ e)



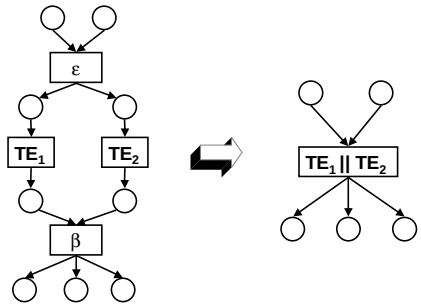
From PN to PNTE

- **Reductions to simplify the net structure**
- **Concurrency relations take**
  - ***O(n²)*** **in Trace expressions**
  - ***O(n³)*** **in Free-Choice systems [Kovalyov & Esparza]**

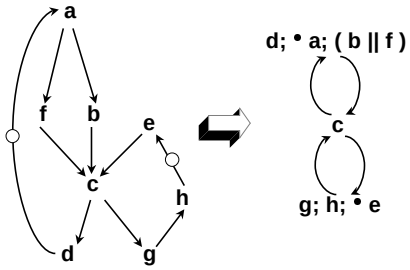
Reductions



Reductions



Example



Outline

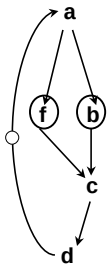
- Motivation
- Design flow
- Verilog HDL specification
- Petri nets and trace expressions
- ☞ Synthesis
- Conclusion

Exploration of the design space

- Kit of transformations at Petri net
  - Concurrency reduction
  - Increase of concurrency
  - Event hiding
- Fast cost estimation
  - Area (Boolean equations)
  - Performance (critical cycles)

Transformations at the net level

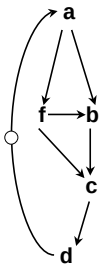
Concurrency reduction



f and b are concurrent !

Transformations at the net level

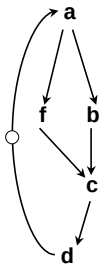
Concurrency reduction



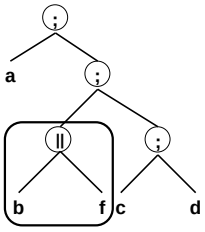
f and b are ordered !

Transformations at the net level

Concurrency reduction in TE

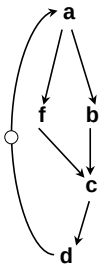


Concurrency in TE:
b and f have a common parallel antecessor

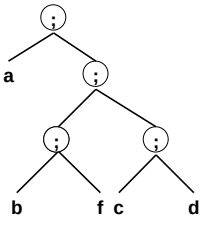


Transformations at the net level

Concurrency reduction in TE

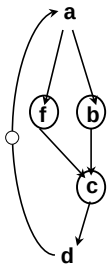


Concurrency reduction:
change the parallelizer by a sequencer



Transformations at the net level

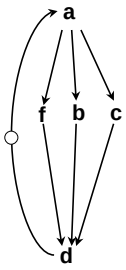
Increase of concurrency



c is ordered with f and b!

Transformations at the net level

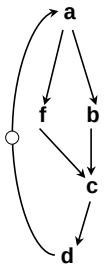
Increase of concurrency



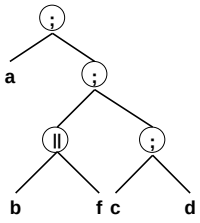
c, f and b are concurrent!

Transformations at the net level

Increase of concurrency in TE

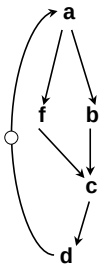


Increase of concurrency: reorganizing the subtree

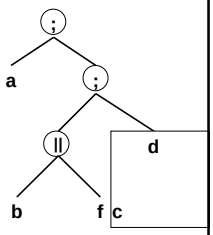


Transformations at the net level

Increase of concurrency in TE

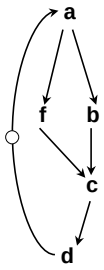


Increase of concurrency: reorganizing the subtree

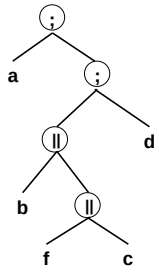


Transformations at the net level

Increase of concurrency in TE

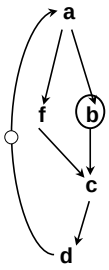


Increase of concurrency: reorganizing the subtree



Transformations at the net level

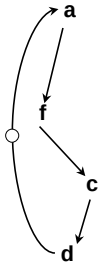
Event hiding



hiding of b !

Transformations at the net level

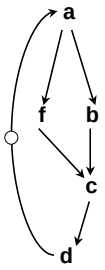
Event hiding



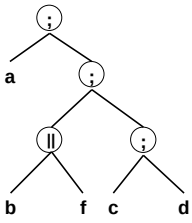
b hidden !

Transformations at the net level

Event hiding in TE

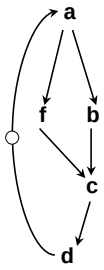


<u>Event hiding</u> : delete the corresponding leaf ...

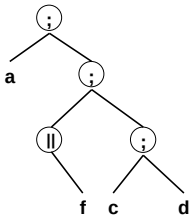


Transformations at the net level

Event hiding in TE

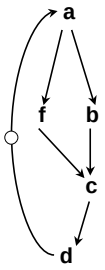


<u>Event hiding</u> : delete the corresponding leaf ...

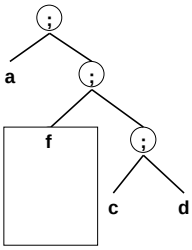


Transformations at the net level

Event hiding in TE



<u>Event hiding</u> : delete the corresponding leaf ... and simplify the tree structure



Synthesis of control logic

For large-scale controllers:

- **Direct translation from Petri Net (or STG-h/s-refined) specifications**
- **Logic synthesis from fully refined STGs with pseudo-one-hot encoding, structural techniques and STG-level optimisations**

Why direct translation?

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

What about performance?

Shifter Example

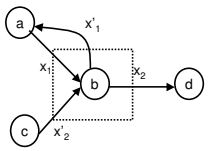
(x:=y;y:=a)* [Bystrov at al, 6th UK Async Forum,'99]

Control Logic option	Speed (ns)
Refined STG directly synthesized by Petrify	5.4
Circuit decomposition with two D-elements	4.2
Circuit decomposition and Petrify re-synthesis	3.3
Re-synthesis with relative timing	1.7

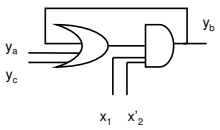
Direct Translation of Petri Nets

- Previous work dates back to 70s
- Synthesis into event-based (2-phase) circuits (similar to 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)

David's original approach

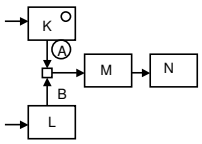


Fragment of flow graph

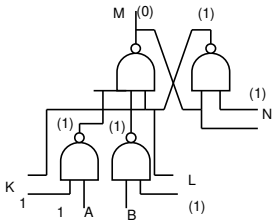


CUSA for storing state b

Hollaar's approach

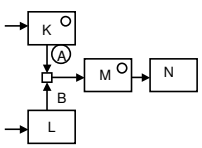


Fragment of flow-chart

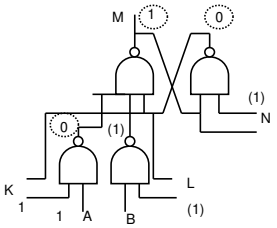


One-hot circuit cell

Hollaar's approach

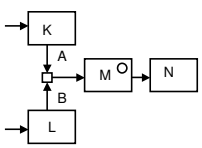


Fragment of flow-chart

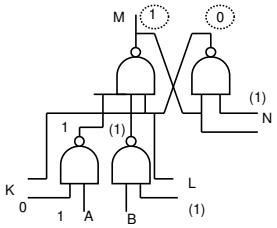


One-hot circuit cell

Hollaar's approach

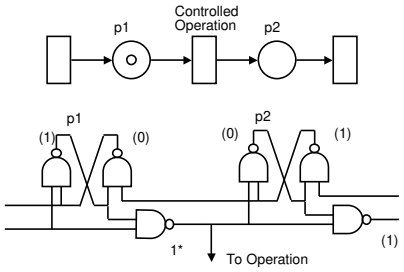


Fragment of flow-chart

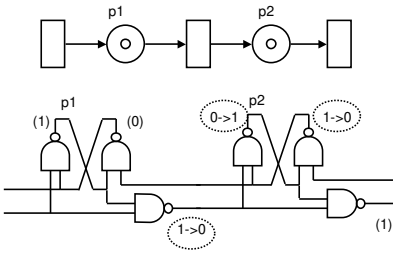


One-hot circuit cell

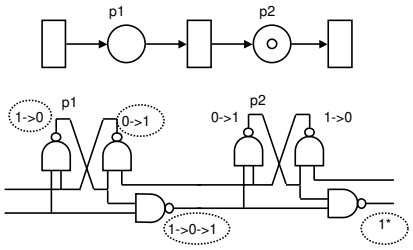
Varshavsky's Approach



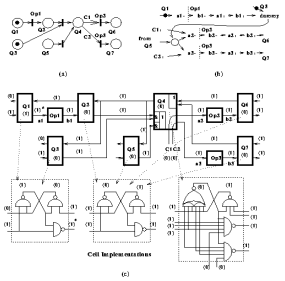
Varshavsky's Approach



Varshavsky's Approach



Translation in brief



This method has been used for designing control of a token ring adaptor [Yakovlev et al.,Async. Design Methods, 1995]

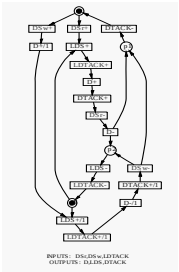
The size of control was about 80 David Cells with 50 controlled hand shakes

Direct translation examples

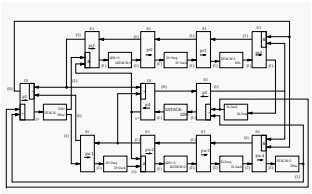
In this work we tried direct translation:

- 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 synthesisb

Example 1: VME bus controller

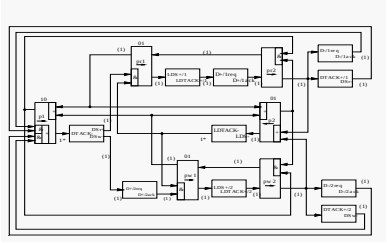


Result of direct translation (DC unoptimised):

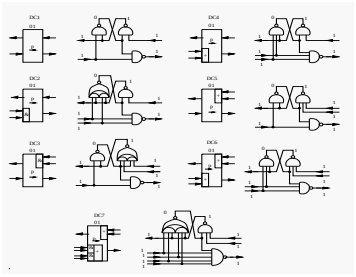


VME bus controller

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

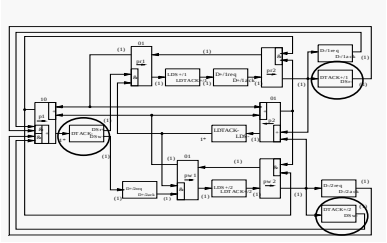


David Cell library



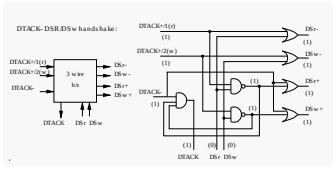
VME bus controller

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



"Data path" control logic

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

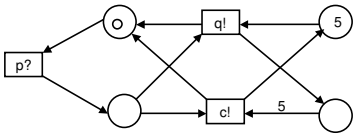


Ex 2: "Flat" mod-6 Counter

TE-like Specification:

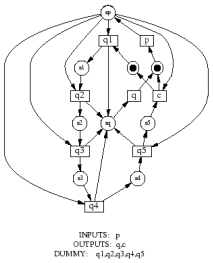
$((p?;q!)^5;p?;c!)^*$

Petri net (5-safe):



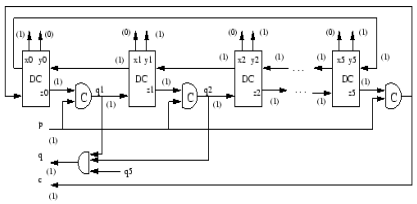
"Flat" mod-6 Counter

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

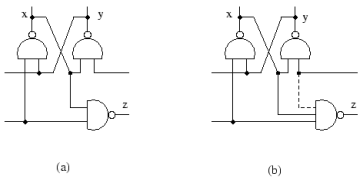


“Flat” mod-6 counter

Result of direct translation (optimised by hand):

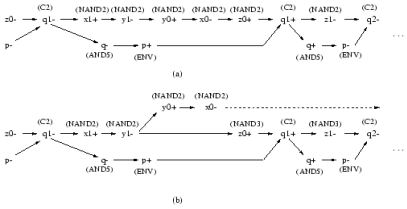


David Cells and Timed circuits



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

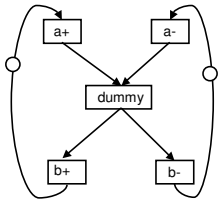
“Flat” mod-6 counter



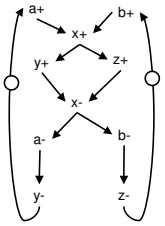
(a) speed-independent (b) with relative timing

“Butterfly” circuit

Initial Specification:

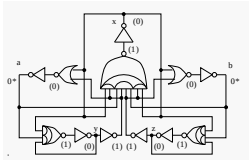


STG after CSC resolution:



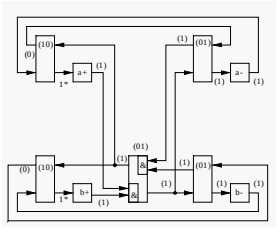
“Butterfly” circuit

Speed-independent logic synthesis solution:



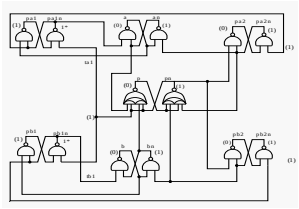
“Butterfly” circuit

Speed-independent DC-circuit:



“Butterfly” circuit

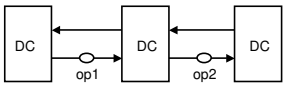
DC-circuit with aggressive relative timing:



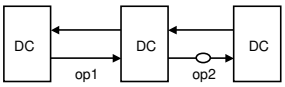
Comparison with logic synthesis

Example	Logic synthesis	DC-translation
VME-bus (overall operation cycle)	6ns	11ns
Mod-6 count (p->q/c, worst case cycle)	>5ns	1.6ns
Butterfly (with RT, operation cycle)	2ns	1.8ns

DC control with Relative Timing



DC control with Relative Timing

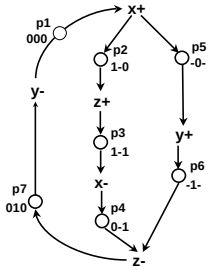


David Cell type	Token shift time
Speed-independent	1.2ns
Mild RT (fast bkwd reset)	0.8ns
Aggressive RT (fast fwd set)	0.4ns

Synthesis

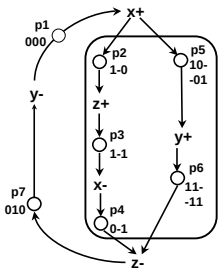
- **Encoding based on a David-cell approach**
- **Transformations to improve area and performance**
- **Structural methods to derive a circuit [Pastor *et al.*] *Transactions on CAD, Nov’98***

Synthesis



Next-state function of signal *y* ?

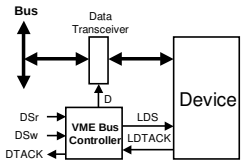
Synthesis

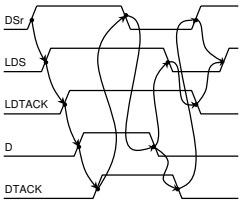


Next-state function of signal *y* ?

$y = x + z$

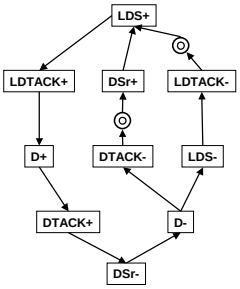
Synthesis example: VME bus



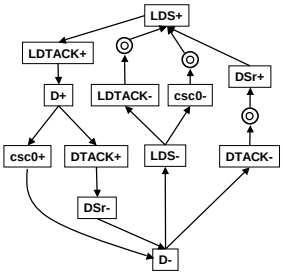


Read Cycle

Synthesis example: VME bus

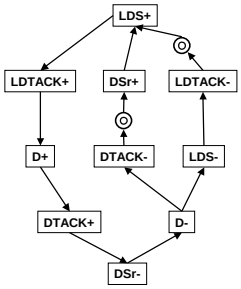


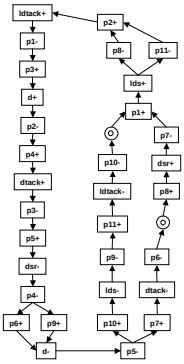
READ CYCLE SPECIFICATION



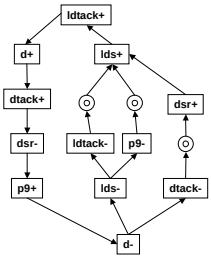
PETRIFY
(Optimizing Performance)

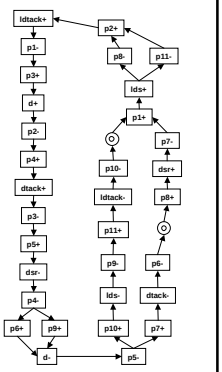
Synthesis example: VME bus



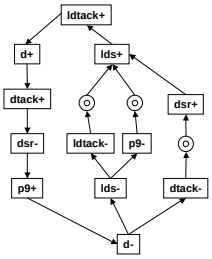


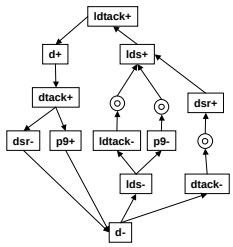
Synthesis example: VME bus





Synthesis example: VME bus

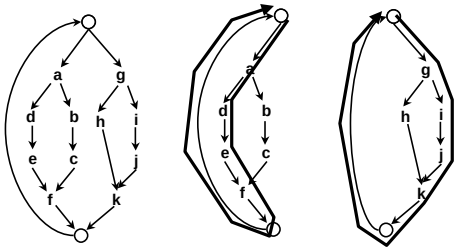




Cost estimation

- **Heuristics:**
  - **AREA :**
    Σ **{ # literals in each Excitacion Region}**
  - **PERFORMANCE :**
    length of critical cycle in the net
- **Exploration of the design space guided by cost estimations**

Performance estimation: critical cycles



Marked-Graph Decomposition

Conclusions

- **Fully automated design flow**
  - **From HDLs (control / data splitting)**
  - **Existing tools for data-path synthesis**
  - **Direct synthesis guarantees implementation (HDL $\rightarrow$ Petri net, Petri-net-based encoding)**
  - **Synthesis of large controllers by efficient spec models (Free-choice Petri nets + trace expressions)**
  - **Exploration of the design space (optimization) by property-preserving transformations**
  - **Logic synthesis by structural methods**