ADVANCED TUTORIAL

Hardware Design and Petri Nets

Organised by

Jordi Cortadella
Luciano Lavagno
Alex Yakovlev

June 2000
# Table of Contents

Overview ................................................................. 1  

Introduction to Hardware Design and Petri Nets ................................. 3  

Hardware Modelling with Petri Nets ........................................ 13  

Direct Synthesis of Petri Nets ............................................. 27  

Logic Synthesis of Asynchronous Circuits from Signal Transition Graphs .... 59  

Partial Order Methods for Analysis and Verification of Asynchronous Circuits 95  

Symbolic Methods for Analysis and Verification ................................ 117  

From Hardware Description Languages to Petri Nets .......................... 131  

Hardware-Software Co-Design Applications ................................... 143  

Bibliography .................................................................. 163
Hardware and Petri nets

Jordi Cortadella  Univ. Politècnica de Catalunya
Luciano Lavagno  Università di Udine
Alex Yakovlev    Univ. Newcastle upon Tyne

Outline

- Introduction to hardware design and Petri nets (AY, 9:00-9:30)
- Hardware modelling with Petri nets (AY, 9:30-10:00)
- Direct synthesis of Petri nets (AY, 10:00-10:30)
- Logic synthesis of async. circuits from STGs (JC, 11:00-12:30)
- Analysis and verification (AY,JC, 14:00-15:00)
  - Partial order methods
  - Symbolic methods
- Petri nets and hardware description languages (LL, 15:30-16:00)
- Performance analysis (AY, 16:00-16:30)
- Petri net models for Quasi-Static-Scheduling (LL, 17:00-18:00)
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

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 soon allows putting 1 billion transistors on a chip
- Systems 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
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

- 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 correctnes (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 Parallel 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. 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 (cf. reference list)
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 (\(a \text{ causes } b\), directly or transitively)
- The order is partial if concurrency is present
- A class of async timed (yet not clocked!) circuits allows special timing order relations (\(a \text{ occurs before } b\), due to delay assumptions)

Simple circuit example

\[
\text{out}=(x+y)*(a+b)
\]

Data flow graph

Control flow graph – Petri net
Muller C-element

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

\[ y = x_1 \cdot x_2 + (x_1 + x_2) y \]

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

NMOS circuit implementation
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 datapath
- 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

Tutorial Outline

- Introduction
- Modeling Hardware with PNs
- Synthesis of Circuits from PN specifications
- Circuit verification with PNs
- Performance analysis using PNs
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
High-level modelling: Processor Example

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

- For use of Coloured Petri net models and use of Design/CPN in processor modelling:

Using Coloured Petri nets

Color Set:
- color Inst = with INT | FPADD | MUL | DIV | BRA | NULL
- color Line = int, color Dep = int, color Target = int, color Count = int timed
- color Value = w cond no Line*Inst+BDep+DTarget+Time

Var Set:
- var fetch, Value:
- var pc, Count
Low-level modelling: “lazy token” ring adaptor

Lazy ring adaptor

Circuit implementation will be shown in the synthesis part
Logic Circuit Modelling

Event-driven elements

- Muller C-element
- Toggle

Petri net equivalents

Logic Circuit Modelling

Level-driven elements

- NOT gate
- NAND gate

Petri net equivalents
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
Event-driven circuit

- Model of next stage: simple delay
- Model of previous stage: simple inverter
- This option results in non-safe behaviour

Level-driven circuit

- C1: \( y_1 = \text{Rin} (y_2) \cdot y_1 (\text{Rin} + \text{n_Aout} + y_2) \)
- C2: \( n_{y_2} = y_1 (n_{Aout} + n_{y_2}) \)
- I1: Rout = \( y_2 \cdot \) or Rout = delay \( n_{y_2} \)
- I1: \( n_{Ain} = y_1' \)
Level-driven circuit

\[ C_1: y_1 = \text{Rin} \cdot y_2 + y_1(\text{Rin} + \text{n_Aout} + y_2) \]
\[ C_2: \text{n_y2} = y_1(\text{n_Aout} + \text{n_y2}) \]
\[ I_1: \text{n_Ain} = y_1' \]
\[ I_1: \text{Rout} = y_2' \text{ or } \text{delay} \Rightarrow \text{n_y2} \]

Without \( y_2 \) in Set part of \( y_1 \) this trace can happen:

\[ I_2+ \]
\[ C1+ \]
\[ C2+ \]
\[ I2- \]
\[ C1- \]
\[ I2+ \]
\[ C2- \]
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

Race between x- and y+ causes nondeterministic behaviour on y:
(1) Either there is a 0-1-0 pulse
(2) Or nothing

Other modelling examples

- Examples with mixed event-based and level-based signalling:
  - Request-Grant-Done (RGD) arbiter with mutex element:
Request-Grant-Done (RGD) arbiter

Interface from event-based part to level-based part

Interface from level-based part to event-based part

(mutex)

(locally optimised model)
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
Tutorial Outline

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

Synthesis Outline

• Abstract synthesis of 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](image-url)
Causality Constraints: General Form(1)

Generic primitive causality constraint: 
\[ \forall t \in (A \cup B)^* : \sum_{i=1}^{m} #a(t) + \sum_{i=1}^{n} #b(t) \leq k \]

Specific cases:
- Simple causality
- 2-delayed causality
- Simple OR-causality
- Simple selection

Causality Constraints: General Form(2)

Same with weights:
\[ \forall t \in (A \cup B)^* : \sum_{i=1}^{m} #a(t)u_i + \sum_{i=1}^{n} #b(t)v_i \leq k \]

Specific cases (modelling counters):
- Simple frequency divider
- 2-goahead frequency multiplier
- OR-causality with divider
- Selector with multiplier
Composition of LPNs

Example: merge
Example: merge (refined)

\[ p_1: \#r_3 \leq \#r_1 + \#r_2 \]
\[ p_2: \#a_1 + \#a_2 \leq \#a_3 \]
\[ p_3 - p_5: \#a_i \leq \#r_i \]
\[ p_6 - p_8: \#r_i \leq \#a_i + 1 \]

More examples

n-place buffer

\[ P(\text{ut}) \quad \text{Buf}(n) \quad G(\text{et}) \]
\[ p_1: \#P \leq \#G + n \]
\[ p_2: \#G \leq \#P + n \]

Modulo-n counter (frequency divider)

\[ \text{in} \quad \text{Div}(n) \quad \text{out} \]
\[ p_1: \#\text{in} = \#\text{out}^2 + 2 \]
\[ p_2: \#\text{out}^2 \leq \#\text{in} \]
Decomposition of LPNs

\[ \text{P(ut)} \xrightarrow{} \text{Buf}(n) \xrightarrow{} \text{G(et)} \]
\[ \text{in} \xrightarrow{} \text{Div}(n) \xrightarrow{} \text{out} \]

Decomposition of LPNs

\[ \text{P(ut)} \xrightarrow{} \text{Buf}(n-1) \xrightarrow{} \text{x} \xrightarrow{} \text{Buf}(1) \xrightarrow{} \text{G(et)} \]
\[ \text{in} \xrightarrow{} \text{Div}(n/2) \xrightarrow{} \text{x} \xrightarrow{} \text{Div}(2) \xrightarrow{} \text{out} \]
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

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.)

Original TS specification

No non-trivial regions!
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)
Region r2: exit(a) enter(x) no-cross(b)
Region r3: exit(x) enter(b) no-cross(a)
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

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)
Molnar’s 5-state TS:

Examples of regions:

\( r_1 = \{E, R\} \) – pre-region(AI), post-region(PI)

\( r_2 = \{I, F, C\} \) – 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 \( \{F\} \)

Solution:

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

Minimal set of regions:
\[ r_1 = \{E_1, E_2, I\} \leftarrow \text{pre}(AR), \text{post}(PR) \]
\[ r_2 = \{E_1, E_2, R\} \leftarrow \text{pre}(AI), \text{post}(PI), \text{co}(d) \]
\[ r_3 = \{R, F, C\} \leftarrow \text{pre}(PR), \text{post}(AR), \text{co}(G) \]
\[ r_4 = \{I, F, C\} \leftarrow \text{pre}(PI), \text{post}(AI) \]
\[ r_5 = \{E_2, I, C\} \leftarrow \text{pre}(PI, AR), \text{post}(G, d) \]
\[ r_6 = \{E_2, R, C\} \leftarrow \text{pre}(PR, AI), \text{post}(G, d) \]
\[ r_7 = \{E_1, I, F\} \leftarrow \text{pre}(G, d), \text{post}(PR, AI) \]
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)
- More about this later (Luciano’s part on HDL and Petri nets)

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

Handshake refinement

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

E.g.

a= P(ut) → Buf(n)  

This may lead to the following refinements at the circuit level
Handshake refinement

Passive handshake

ar → circuit
ak ← Circuit produces (first) request

Environment produces (first) request

ar

ak

Passive handshake

ar

ak

Active handshake

ar → circuit
ak ← Circuit produces (first) request

ar

ak

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

Handshake refinement

Passive handshake

ar → circuit
ak ← Circuit produces (first) request

Environment produces (first) request

ar

ak

Passive handshake

ar

ak

Active handshake

ar → circuit
ak ← Circuit produces (first) request

ar

ak

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

Handshake refinement

Passive handshake

ar → circuit
ak ← Circuit produces (first) request

Environment produces (first) request

ar

ak

Passive handshake

ar

ak

Active handshake

ar → circuit
ak ← Circuit produces (first) request

ar

ak

Four phase, return-to-zero (RTZ) protocol
Handshake refinement example

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
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

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 for 1-safe net
- "Pragmatic" extensions to Patil's set (for non-simple PNs)
- Examples: modulo-N up/down counter, Lazy ring adapter

Patil's set of modules

<table>
<thead>
<tr>
<th>Petri net fragment:</th>
<th>Circuit equivalent:</th>
</tr>
</thead>
<tbody>
<tr>
<td>place</td>
<td>wire</td>
</tr>
<tr>
<td>marked place</td>
<td>inverter</td>
</tr>
<tr>
<td>join</td>
<td>C-element</td>
</tr>
<tr>
<td>merge</td>
<td>XOR</td>
</tr>
<tr>
<td>fork</td>
<td>fan-out</td>
</tr>
<tr>
<td>shared (conflict) place</td>
<td>switch Effectively RGD arbiter</td>
</tr>
</tbody>
</table>
Example

Two phase (NRZ) protocol:

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

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

Problems with C-elements

x1 and x2 are mutually exclusive – so no need for a 5-switch (RGD arbiter)

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)
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)

structure

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

(a)

Direct synthesis example
(lazy token ring adapter)

(b)

(c)

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

CUSC element for storing state b
Hollaar’s approach

Fragment of a flow-chart (allows parallelism)

Varshavsky’s Approach

One-hot circuit cell
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).
- Fundamental modelling mismatch (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
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:

![Diagram of a Petri net for a mod-6 counter]

“Flat” mod-6 counter

Result of direct translation (optimised by hand):
“Butterfly” circuit

Initial Specification:

\[ a^+ \rightarrow \text{dummy} \rightarrow b^- \]

\[ a^- \rightarrow \text{dummy} \rightarrow b^+ \]

STG after CSC resolution:

\[ a^+ \xrightarrow{y^+} x^+ \xrightarrow{z^+} b^+ \]

\[ a^- \xrightarrow{y^-} x^- \xrightarrow{z^-} b^- \]

“Butterfly” circuit

Speed-independent logic synthesis solution:
“Butterfly” circuit

Speed-independent DC-circuit:

Conclusion on direct synthesis

- Direct synthesis is a powerful way 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 their fast prototypes (remember: productivity is the king!) 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

Synthesis of asynchronous circuits from Signal Transition Graphs

Outline

- Overview of the synthesis flow
- Specification
- State graph and next-state functions
- State encoding
- Implementability conditions
- Speed-independent circuit
- Logic decomposition
Design flow

Signal Transition Graph (STG)
**Next-state functions**

\[ x = \bar{z} \cdot (x + \bar{y}) \]
\[ y = z + x \]
\[ z = x + \bar{y} \cdot z \]
Simple examples

A input
B output

Simple examples
Simple examples

A FIFO controller
Design flow

Specification (STG) → Reachability analysis
State Graph → State encoding
SG with CSC → Boolean minimization
Next-state functions → Logic decomposition
Decomposed functions → Technology mapping
Gate netlist

VME bus

Bus
Data Transceiver
DSr
DSw
DTACK

Device
VME Bus Controller
LDS
LDTACK

Read Cycle

DSr
LDS
LDTACK
D
DTACK
**STG for the READ cycle**

**Choice: Read and Write cycles**
**Choice: Read and Write cycles**

**Circuit synthesis**

- **Goal:**
  - Derive a hazard-free circuit under a given delay model and mode of operation
**Speed independence**

- **Delay model**
  - Unbounded gate / environment delays
  - Certain wire delays shorter than certain paths in the circuit

- **Conditions for implementability:**
  - Consistency
  - Complete State Coding
  - Persistency

---

**Design flow**

1. Specification (STG)
   - Reachability analysis
2. State Graph
   - State encoding
3. SG with CSC
   - Boolean minimization
4. Next-state functions
   - Logic decomposition
5. Decomposed functions
   - Technology mapping
6. Gate netlist
**STG for the READ cycle**

- LDS+ \(\rightarrow\) LDTACK+ \(\rightarrow\) D+ \(\rightarrow\) DTACK+ \(\rightarrow\) DSr- \(\rightarrow\) D-

- DTACK- \(\rightarrow\) LDTACK- \(\rightarrow\) LDS-

- DSr+ \(\rightarrow\) DSr-

**Binary encoding of signals**

- LDS+ \(\rightarrow\) LDS+

- LDS- \(\rightarrow\) LDS-

- LDS = 0 \(\rightarrow\) LDS = 1
Binary encoding of signals

(DSr, DTACK, LDTACK, LDS, D)

Excitation / Quiescent Regions

(ER, QR)

10000

10010

10110

01100

00110

01110

10110

D+

DSr+

DTACK+

LDS+

LDTACK+

ER (LDS+)
Next-state function

Karnaugh map for LDS

<table>
<thead>
<tr>
<th>D</th>
<th>DTACK</th>
<th>DSR</th>
<th>LDS = 0</th>
<th>LDS = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDTACK</td>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>DTACK</th>
<th>DSR</th>
<th>LDS = 0</th>
<th>LDS = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDTACK</td>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0/1?</td>
</tr>
</tbody>
</table>
Design flow

1. Specification (STG)
2. Reachability analysis
3. State Graph
4. State encoding
5. SG with CSC
6. Boolean minimization
7. Next-state functions
8. Logic decomposition
9. Decomposed functions
10. Technology mapping
11. Gate netlist

Concurrency reduction
Concurrency reduction

State encoding conflicts
Signal Insertion

Design flow

Specification (STG) -> Reachability analysis
State Graph -> State encoding
SG with CSC -> Boolean minimization
Next-state functions -> Logic decomposition
Decomposed functions -> Technology mapping
Gate netlist
Complex-gate implementation

\[
LDS = D + \text{csc} \\
DTACK = D \\
D = \overline{LDTACK} \cdot \text{csc} \\
\text{csc} = DSr \cdot (\text{csc} + \overline{LDTACK})
\]

Implementability conditions

- Consistency
  - Rising and falling transitions of each signal alternate in any trace

- Complete state coding (CSC)
  - Next-state functions correctly defined

- Persistency
  - No event can be disabled by another event (unless they are both inputs)
Implementability conditions

- Consistency + CSC + persistency

- There exists a speed-independent circuit that implements the behavior of the STG

(under the assumption that any Boolean function can be implemented with one complex gate)

Persistency

\[
\begin{align*}
100 & \xrightarrow{a^-} 000 & 001 & \xrightarrow{b^-} \\
\downarrow & b+ & \downarrow & b+
\end{align*}
\]

Speed independence \(\Rightarrow\) glitch-free output behavior under any delay
### Complex gate

\[
d = ad + \overline{ac}
\]

Complex gate

### Implementation with C elements

\[
\text{\textbullet\textbullet\textbullet} \rightarrow S+ \rightarrow z+ \rightarrow S- \rightarrow R+ \rightarrow z- \rightarrow R- \rightarrow \text{\textbullet\textbullet\textbullet}
\]

- $S$ (set) and $R$ (reset) must be mutually exclusive
- $S$ must cover $ER(z+)$ and must not intersect $ER(z-) \cup QR(z-)$
- $R$ must cover $ER(z-)$ and must not intersect $ER(z+) \cup QR(z+)$
but ...
Assume that $R=\overline{ac}$ has an unbounded delay.

Starting from state 0000 (R=1 and S=0):

\[ a^+ ; R^- ; b^+ ; a^- ; c^+ ; S^+ ; d^+ ; \]

R+ disabled (potential glitch)

Monotonic covers
**C-based implementations**

- Generalized C elements (gC)

**Speed-independent implementations**

- Implementability conditions
  - Consistency
  - Complete state coding
  - Persistency

- Circuit architectures
  - Complex (hazard-free) gates
  - C elements with monotonic covers
  - ...
Derive circuits for signals $x$ and $z$ (complex gates and monotonic covers)
Synthesis exercise

<table>
<thead>
<tr>
<th>y \ z</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Signal z

Design flow

- Specification (STG)
  - Reachability analysis
- State Graph
  - State encoding
- SG with CSC
  - Boolean minimization
- Next-state functions
  - Logic decomposition
- Decomposed functions
  - Technology mapping
- Gate netlist
No Hazards

Decomposition May Lead to Hazards
How about 2-input gates?

\[ \text{How about 2-input gates?} \]

\[ \begin{align*}
    &d^- \rightarrow b^+ \rightarrow d^+ \rightarrow y^+ \rightarrow a^- \rightarrow y^- \rightarrow c^+ \rightarrow d^- \\
    &c^- \leftarrow d^+ \rightarrow z^- \rightarrow b^- \leftarrow z^+ \rightarrow c^+ \leftarrow a^+ \leftarrow c^- \\
\end{align*} \]

\[ \begin{align*}
    c & \rightarrow z \\
    b & \rightarrow z \\
    a & \rightarrow z \\
\end{align*} \]

\[ \begin{align*}
    a & \rightarrow y \\
    b & \rightarrow y \\
    d & \rightarrow y \\
\end{align*} \]

\[ \text{How about 2-input gates?} \]

\[ \begin{align*}
    &d^- \rightarrow b^+ \rightarrow d^+ \rightarrow y^+ \rightarrow a^- \rightarrow y^- \rightarrow c^+ \rightarrow d^- \\
    &c^- \leftarrow d^+ \rightarrow z^- \rightarrow b^- \leftarrow z^+ \rightarrow c^+ \leftarrow a^+ \leftarrow c^- \\
\end{align*} \]

\[ \begin{align*}
    d^- & \rightarrow \text{XOR} \rightarrow d^+ \\
    b^- & \rightarrow \text{XOR} \rightarrow b^+ \\
    c^- & \rightarrow \text{XOR} \rightarrow c^+ \\
\end{align*} \]
How about 2-input gates?

How about 2-input gates?
**Strategy for logic decomposition**

- Each decomposition defines a new internal signal

- **Method:** Insert new internal signals such that
  - After resynthesis, some large gates are decomposed
  - The new specification is hazard-free

- Generate candidates for decomposition using standard logic factorization techniques:
  - Algebraic factorization
  - Boolean factorization (boolean relations)

---

**Signal insertion for function $F$**

Insertion by input borders

- $F=0$
- $F=1$

State Graph
Event insertion

\[ a \quad b \quad c \]

\[ \text{ER}(x) \]

Event insertion

\[ a \quad b \quad c \]

\[ \text{SR}(x) \]

\[ \text{ER}(x) \]
Properties to preserve

- a is disabled by b = hazards
- a is persistent

Decomposition example
z- is delayed by the new transition s-!
**Conclusions**

- The synthesis of asynchronous control circuits from Petri net specifications can be totally automated.

- Existing tools at academia (http://www.lsi.upc.es/~jordic/petrify)

- An asynchronous circuit is a concurrent system with processes (gates) and communication (wires).

- The theory of concurrency is crucial to formalize automatic synthesis methods.
Hardware and Petri nets

Partial order methods for analysis and verification of asynchronous circuits

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)

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 …
Petri net and its unfolding

Petri net and its unfolding

PN transition and its instance in 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**

Prefix Construction Algorithm

**Proc** Build prefix \((N = \langle P, T, F, M_0 \rangle)\)

1. Initialise \(N'\) with instances of places in \(M_0\)
2. Initialise Queue with instances of \(t\) enabled at \(M_0\)
3. **while** Queue is not empty **do**
   1. Pull \(t'\) from Queue
   2. **if** \(t'\) is not cutoff **then** do
      1. Add \(t'\) and succ\((t')\) to \(N'\)
      2. **for each** \(t\) in \(T\) **do**
         1. **Find unused set of mutually concurrent instances of pred\((t)\)**
         2. **if** such set exists **then** do
            1. Add \(t'\) to Queue in order of its prehistory size
         **end do**
      **end do**
   **end do**
4. **end do**
5. **return** \(N'\)

**end proc**
**Cut-off definition**

- A newly built transition instance $t_1'$ in the unfolding is a cut-off point if there exists another instance $t_2'$ (of possibly another transition) whose:
  - Final cut maps to the same marking is the final cut of $t_1'$, and
  - The size of prehistory (local configuration) of $t_2'$ is strictly greater than that of $t_1'$

[McMillan, 1992]

- Initial marking and its min-cut are associated with an imaginary “bottom” instance (so we can cut-off on $t_7$ 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 $O(2^n)$.

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

- the structure and class of the net, and
- initial marking.
Size of Prefix

However this part is redundant

Non-1-safe net

However this part is redundant

Cut-offs

However this part is redundant
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**

[Diagram of STG Unfolding with nodes and transitions labeled with signals and states.]
**STG Unfolding**

- **STG Uninterpreted PN Reachability Graph**
- **Binary-coded STG Reach. Graph (State Graph)**
- **STG unfold. prefix**

**Consistency and Signal Deadlock**

- **STG PN Reach. Graph**
- **STG State Graph**

Signal deadlock wrt b+ (coding consistency violation)
Signal Deadlock and Autoconcurrency

STG

STG State Graph

STG Prefix

Signal deadlock wrt b+ (coding consistency violation)

Autoconcurrency wrt b+

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)

<table>
<thead>
<tr>
<th>Name</th>
<th>States</th>
<th>Verif. only</th>
<th>PUNT Ver.</th>
<th>PUNT Total</th>
<th>Places</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>c-elem</td>
<td>64</td>
<td>0.01</td>
<td>0.11</td>
<td>7</td>
<td>12</td>
<td>0.07</td>
</tr>
<tr>
<td>chu172</td>
<td>768</td>
<td>0.02</td>
<td>0.26</td>
<td>13</td>
<td>14</td>
<td>0.11</td>
</tr>
<tr>
<td>espinalt-ba</td>
<td>15360</td>
<td>0.07</td>
<td>0.74</td>
<td>13</td>
<td>17</td>
<td>0.12</td>
</tr>
<tr>
<td>espinalt-go</td>
<td>27648</td>
<td>0.1</td>
<td>0.83</td>
<td>25</td>
<td>30</td>
<td>0.17</td>
</tr>
<tr>
<td>fair-arb-sq</td>
<td>1280</td>
<td>0.09</td>
<td>0.8</td>
<td>32</td>
<td>33</td>
<td>0.51</td>
</tr>
<tr>
<td>josepm</td>
<td>45056</td>
<td>0.7</td>
<td>0.72</td>
<td>21</td>
<td>29</td>
<td>0.12</td>
</tr>
<tr>
<td>master-res</td>
<td>3.45E+07</td>
<td>0.39</td>
<td>7.4</td>
<td>51</td>
<td>78</td>
<td>0.37</td>
</tr>
<tr>
<td>t1</td>
<td>618496</td>
<td>2.65</td>
<td>8.97</td>
<td>67</td>
<td>104</td>
<td>2.87</td>
</tr>
<tr>
<td>irred.no1to</td>
<td>41472</td>
<td>0.19</td>
<td>0.93</td>
<td>6</td>
<td>10</td>
<td>0.07</td>
</tr>
<tr>
<td>TOTAL</td>
<td>4.37</td>
<td>26.98</td>
<td>6.13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

### Experimental results (from Kondratyev)

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

**Level-driven elements**

- NOT gate: \( x(=1), y(=0) \)
- NAND gate: \( x(=1), y(=1), z(=0) \)

**Petri net equivalents**

**Self-loops in ordinary P/T nets**

**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.
Circuit Petri nets

C1: \( y_1 = R_{i_{2}} \cdot y_1 + y_1 (R_{i_{2}} + n_{A_{out}} + y_2) \)
C2: \( n_{y_2} = y_1 \cdot (n_{A_{out}} + n_{y_2}) \)
I1: \( n_{y_2} = y_1 \cdot Rout \)

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 semantical 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.)

Example: producer-consumer
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

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

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

---

Exper. results (from Alamsyah)

<table>
<thead>
<tr>
<th>Buffer</th>
<th>k-bounded net with Mcmillan's unfolding</th>
<th>safe nets using ERV's algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Original</td>
<td>Unfolding (t/p)</td>
</tr>
<tr>
<td>2</td>
<td>6/8</td>
<td>184/317</td>
</tr>
<tr>
<td>3</td>
<td>6/8</td>
<td>1095/1896</td>
</tr>
<tr>
<td>4</td>
<td>6/8</td>
<td>6944/11911</td>
</tr>
<tr>
<td>5</td>
<td>6/8</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>6/8</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>6/8</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>6/8</td>
<td>-</td>
</tr>
</tbody>
</table>
Exper. results (cont.)

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Safe nets using ERV's algorithm</th>
<th>FIFO-unfolding with McMillan's</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Unfolding (vp)</td>
</tr>
<tr>
<td>2</td>
<td>6/14</td>
<td>25/68</td>
</tr>
<tr>
<td>3</td>
<td>10/18</td>
<td>48/105</td>
</tr>
<tr>
<td>4</td>
<td>12/22</td>
<td>67/150</td>
</tr>
<tr>
<td>5</td>
<td>14/26</td>
<td>92/203</td>
</tr>
<tr>
<td>6</td>
<td>16/30</td>
<td>121/264</td>
</tr>
<tr>
<td>7</td>
<td>18/34</td>
<td>154/333</td>
</tr>
<tr>
<td>8</td>
<td>20/38</td>
<td>191/410</td>
</tr>
</tbody>
</table>

Conclusions

- 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., individualize tokens in k-safe nets or split self-loops) and thus spawn too many branches in history.
- See Javier Esparza's web page: http://wwwbrauer.informatik.tum.de/gruppen/theorie/ for more information and comparison between unfoldings and BDDs.
- Other forms of unfolding can be studied (e.g., non-aggressive unfolding of places—but 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—Hurray the lads!
Hardware and Petri nets

Symbolic methods for analysis and verification

Outline

- Representing Boolean functions with BDDs
- Symbolic traversal for reachability set calculation
- State encoding
- Structural methods for efficient encoding
Representing Boolean functions
**Representing Boolean functions**

![Binary Decision Diagram]

**Binary Decision Diagrams**

- All variables appear in the same order
- No isomorphic subgraphs
- Canonical form
- Efficient form for many functions
- [Bryant, ACM Comp. Surveys, September 1992]
Reachable markings

Boolean encoding
**Boolean encoding**

\[ \overline{p_1} \overline{p_2} \overline{p_3} (p_4 \oplus p_6) (p_5 \oplus p_7) \]

\[ \land \]

\[ \text{Enabled}(t_0) = p_6 p_7 \]

\[ \equiv \]

\[ \overline{p_1} \overline{p_2} \overline{p_3} \overline{p_4} \overline{p_5} \overline{p_6} \overline{p_7} \]

\[ \downarrow \]

\[ \text{(toggle } p_1, p_6, p_7) \]

\[ \downarrow \]

\[ p_1 \overline{p_2} \overline{p_3} \overline{p_4} \overline{p_5} \overline{p_6} \overline{p_7} \]

**Symbolic Traversal (BFS algorithm)**

\[ \text{Reached} = \text{From} = \{m_0\}; \]

\textbf{repeat}

\[ \text{From } \{ T \} \text{ To}; \]

\[ \text{New} = \text{To} \setminus \text{Reached}; \]

\[ \text{From} = \text{New}; \]

\[ \text{Reached} = \text{Reached} \cup \text{New}; \]

\textbf{until} \[ \text{New} = \emptyset; \]

\# iterations: sequential depth of the net
**Reachability Set computation**

- Based on BFS *Image* computation
  
  \[
  S_0 = M_0 \\
  S_{i+1} = S_i \cup \text{Image}(S_i)
  \]

  Monotonic increase until fix point \( S_{i+1} = S_i \)

**Boolean encoding**

Sparse encoding:

Seven variables:

\[ p1 \ p2 \ p3 \ p4 \ p5 \ p6 \ p7 \]

Very easy to derive and use

Less efficient in terms of BDDs

Optimal encoding:

Three variables \( \left\lceil \log_2 |RG| \right\rceil \):

\[ v1 \ v2 \ v3 \]

But the reachability graph should be known a priori ...

---

But the reachability graph should be known a priori ...
**Encoding for safe PNs**

### Observation:
- Not all combinations of tokens are possible.
- Find relations among places to reduce the number of variables!!

**PN structure: Place Invariant**
- Set of places with a constant weighted sum of tokens
  \[ k_1 p_1 + k_2 p_2 + \ldots + k_n p_n = B \]
- Specially efficient for safe PNs (State Machines)
  \[ p_1 + p_2 + \ldots + p_n = 1 \]
- Computed by linear programming techniques
State Machine Components

State Machine Components
Encoding for safe PNs

Two variables: \( v_1, v_2 \)

Two additional variables: \( v_3, v_4 \)

Encoding for safe PNs

Four variables: \( v_1, v_2, v_3, v_4 \)
**Encoding for bounded PNs**

Sparse encoding: 10 variables

Invariants of the PN:

- I1: $2p_1 + 4p_2 - p_4 = 4$
- I2: $p_1 + p_2 + p_3 = 3$

Dense encoding: 3 variables

Invariants of the PN:

- I1: $2p_1 + 4p_2 - p_4 = 4$
- I2: $p_1 + p_2 + p_3 = 3$
Invariants characterize the set of all potentially reachable markings (an overestimation of the reachable markings)

**Invariants of the PN:**

1. \(2p_1 + 4p_2 - p_4 = 4\)
2. \(p_1 + p_2 + p_3 = 3\)

**Dense encoding: 3 variables**

**Encoding for bounded PNs**

2p1 + 4p2 - p4 = 4 (8 vars)  
\[ \begin{align*}
\text{p}1 & \quad 2 \quad 3 \\
\text{p}2 & \quad 0 \quad 1 \quad 2 \\
\text{p}4 & \quad 0 \quad 4 \quad 8 \quad 2 \\
\end{align*} \]

p1 + p2 + p3 = 3

\[ \begin{align*}
\text{p}1 & \quad 2 \quad 3 \\
\text{p}2 & \quad 0 \quad 1 \quad 0 \\
\text{p}3 & \quad 1 \quad 0 \quad 0
\end{align*} \]
Encoding for bounded PNs

\[ p_1 + p_2 + p_3 = 3 \]

\[ M(p_1) + M(p_2) \]

Encoding for bounded PNs

\[ 2p_1 + 4p_2 - p_4 = 4 \]

\[ p_3 = 3 - p_1 - p_2 \]
Upper bounds for the state space

2p1 + 4p2 - p4 = 4

Characteristic function for potentially reachable states

Experimental results

| PN  | States    | Vars  | $|BDD|$ | Ni | Nn | CPU |
|-----|-----------|-------|-------|----|----|-----|
| muller10 | $4.2 \times 10^2$ | 40 / 20 | 770 / 189 | 10 | 40 | 1 1 |
| muller20 | $2.5 \times 10^5$ | 80 / 40 | 3188 / 668 | 20 | 80 | 9 3 |
| muller30 | $6.0 \times 10^7$ | 120 / 60 | 6694 / 1390 | 30 | 120 | 51 13 |
| phil5   | $8.5 \times 10^4$ | 65 / 25 | 639 / 158 | 15 | 125 | 2 2 |
| phil10  | $7.4 \times 10^3$ | 130 / 50 | 7805 / 433 | 30 | 250 | 40 24 |
| phil15  | $6.4 \times 10^{14}$ | 195 / 75 | 87419 / 708 | 45 | 375 | 700 124 |
| slot5   | $1.7 \times 10^5$ | 50 / 25 | 673 / 129 | 10 | 50 | 14 5 |
| slot10  | $3.8 \times 10^{11}$ | 100 / 50 | 2516 / 460 | 20 | 100 | 1006 309 |
Conclusions

- Formal verification and synthesis often suffer from the state explosion problem.

- Symbolic techniques can be used to efficiently represent the state space.

- Structural techniques are crucial to overestimate and encode the state space.

- Try to resist the temptation of using BDDs from the very beginning. Use them rationally and only if desperate.
Hardware and Petri nets

From Hardware Description Languages to Petri Nets

Joint work with:
Ivan Blunno

The goal of our work

HDL Specification

Asynchronous Control Unit

Synchronous Data Path
**Motivation**

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

---

**Control-data partitioning**

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

---

![Control-data partitioning diagram](image-url)
**Design flow**

- Behavioral HDL specification
- Control/data splitting
- STG specification
- Synthesis (petrify)
- Logic implementation
- Delay insertion
- HDL netlist implementation

---

**HDL choice: standard HDL**

- Several available tools, broad user basis
- BUT syntax and semantics oriented to gates (not STGs or BFSMs or process algebra)
- Need to define a subset for synthesis

- Choice
  - VHDL
  - Verilog
Asynchronous Verilog subset

- Declarations
- Initialization and main loop
- Asynchronous Verilog blocks
- wait statements
- Block assignments
- if - else statements

Initialization and main loop

- 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 synthesized
  - always loop surrounds live behavior
  - initial block defines initialization sequence
Asynchronous Verilog blocks

- Only structured mix of sequencing, concurrency and choice can be specified
  - begin-end for sequencing
  - fork-join for concurrency
  - if-else for input choice

```verilog
wait(start);
```
RES = 0;

RES = A + B;

always @(posedge RES_out)
begin
    if(cmd_RES_0_out == 0)
    begin
        RES = 0;
    end
    if(cmd_RES_0_out == 1)
    begin
        RES = A + B;
    end
end
**Assignments: STG specification**

load register req

REG control

load register ack

reg 4ph completion

**Assignments**

RES = 0;

......

......

RES = A + B;

C.U.

**D.P. block diagram**

RES

0

A + B

cmd_RES.0_out

RES_out

RES_out

RES

C.U.

register - MUX
Assignments: STG specification

IF - ELSE statements

if (RES[7] == 1) ....
else .....
IF - ELSE statements

if(RES[7] == 1) .....
else .....

always @(posedge cmd_cmp_0_out)
begin
    if(RES[7] == 1) cmp_0 = 1;
    else cmp_0 = 0;
end

IF - ELSE statements: STG

PH
   trF
   Pzero
   cmp_0-
   cmp_0+
 ELSE branch
   PK
   trQ

IF branch
   trT
   Pone

D.P. Verilog specification

C.U.
STG
comparator - FF
IF - ELSE statements: data path

```plaintext
if (RES[7] == 1) ..... 
else ..... 
```

D.P. block diagram

IF - ELSE statements: final STG
Example: FIR filter

RES[z] = SMP[z] \times 4 + SMP[z - 1] \times 3

(saturating arithmetic)

always
begin
wait(start);
RES = SMP \times 4 + R;
R = SMP \times 3;
if(RES[7] == 1) RES = 0;
else begin
if(RES[6] == 1) RES = h3F;
end;
done = 1;
wait(!start);
done = 0;
end
Future work

- Extension of Verilog HDL subset
- Data Path and Control Unit optimizations
- Module interface design automation
- Syntax-directed state encoding to speed up control unit logic synthesis
Hardware and Petri nets

Hardware-software co-design applications
Joint work with:
Jordi Cortadella, Alex Kondratyev,
Marc Massot, Sandra Moral,
Claudio Passerone, Marco Sgroi,
Yosinori Watanabe

Outline

• Motivation
• Quasi-Static Scheduling of process networks and Petri nets
• Free-choice nets
• Non-free choice nets
• Conclusions
**Embedded Software Synthesis**

- System specification: set of concurrent functional blocks (DF actors, CFSMs, CSP, …)
- Software implementation: (smaller) set of concurrent software tasks
- Two sub-problems:
  - Generate code for each task (from code fragments of functional blocks)
  - Schedule tasks dynamically (to satisfy Real-time constraints)
- Goal: minimize real-time scheduling overhead

---

**Quasi-Static Scheduling**

- Sequentialize concurrent operations as much as possible
- better starting point for compilation:
  - straight-line code from function blocks
  - run-time task generation
  - Must handle data-dependent control, multi-rate communication
The problem

- Given:
  a network of Kahn processes
  - Kahn process: sequential function + ports
  - communication: point-to-point, uni-directional, multi-rate through ports

- Find:
  a single task (function block)
  - functionally equivalent to the original network (modulo concurrency)
The scheduling procedure

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

2. Translate to the computational model: Petri nets

3. Find a “schedule” on the Petri net

4. Translate the schedule to a task

From process network to Petri Net

D and E are in conflict.
(modeling data-dependent control)
### Existing approaches

- **Lee - Messerschmitt ‘86**
  - Static Data Flow: cannot specify data-dependent control
- **Buck - Lee ‘94**
  - Boolean Data Flow: scheduling problem is undecidable
- **Thoen - Goossens - De Man ‘96**
  - Event graph: no schedulability check, no minimization of number of tasks
- **Lin ‘97**
  - Safe Petri Net: no schedulability check, no multi-rate
- **Thiele - Teich ‘99**
  - Bounded Petri Net: partial schedulability check, complex (reachability-based) algorithm

---

### Scheduling Petri Nets

- Unified model for mixed control and dataflow
- Most properties are decidable
- A lot of theory is available

\[ \text{Static Data Flow network} \]

\[ \text{Petri net} \]
**Bounded scheduling**

- A finite complete cycle is a finite sequence of transition firings that returns the net to its initial state
  - Bounded memory
  - Infinite execution
- To find a finite complete cycle solve

\[
\begin{bmatrix}
1 & 0 \\
-2 & 1 \\
0 & 0 & -2
\end{bmatrix}
\begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} = 0
\]

T-invariant \( f(\sigma) = (4,2,1) \)

\[
f(\sigma) D = 0 \text{ has no solution}
\]

No schedule

**Outline**

- Motivation
- Quasi-Static Scheduling of process networks and Petri nets
- Free-choice nets
- Non-free choice nets
- Conclusions
**Free-Choice Petri Nets (FCPN)**

- **Marked Graph (MG)**
- **Free-Choice**
- **Confusion (not-Free-Choice)**

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

---

**Bounded scheduling**

- Can the “adversary” ever force token overflow?
Can the “adversary” ever force token overflow?
Can the “adversary” ever force token overflow?
Bounded scheduling

- Can the “adversary” ever force token overflow?

Schedulability of a Petri Net

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

$\Sigma=\{ (t_1 \ t_2 \ t_4), (t_1 \ t_3 \ t_5) \} \rightarrow \text{Schedulable}$
**How to check schedulability**

- Basic intuition: every resolution of data-dependent choices must be schedulable
- Algorithm:
  - Decompose the given Free-Choice Petri Net into as many Conflict-Free components as the number of possible resolutions of the non-deterministic choices.
  - Check if every component is statically schedulable
  - Derive a valid schedule, i.e. a set containing one static schedule for each component

**How to check schedulability**

- An Allocation is a control function that chooses which transition fires among several conflicting ones
- A Conflict-free component is the subnet generated from the corresponding allocation by applying the Reduction Algorithm
- A FCPN is allocatable if every Conflict-free component has a solution of the state equations
- Theorem: A FCPN is schedulable iff
  - it is allocatable and
  - every component is schedulable (using Lee’s algorithm)
Reduction Algorithm

T-allocation $A_1=\{t_1,t_2,t_4,t_5,t_6,t_7\}$

From schedule to C code

Task 1:
```c
{ t1;
    if (p1)
        t2;
        count(p2)++;
        if (count(p2) = 2)
            t4;
            count(p2) = count(p2) - 2;
        else
t           t3;
           t5;
    }
```

Task 2:
```c
{ t6;
    t7;
    t5;
}
```

$\Sigma=\{(t_1 \ t_2 \ t_1 \ t_2 \ t_4 \ t_6 \ t_7 \ t_5) \ (t_1 \ t_3 \ t_5 \ t_6 \ t_7 \ t_5)\}$
Application example: ATM Switch

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

Functional Decomposition

Accept/discard cell

- Clock divider
- Output cell enabler

Output time selector

4 Tasks
(+ 1 arbiter)
Minimal (QSS) Decomposition

Input cell processing

Output cell processing

Real-time scheduling of tasks

Task 1

+ RTOS

Task 2

Shared Processor
**ATM: experimental results**

### Functional partitioning vs. QSS

<table>
<thead>
<tr>
<th>Sw Implementation</th>
<th>QSS</th>
<th>Functional partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tasks</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Lines of C code</td>
<td>1664</td>
<td>2187</td>
</tr>
<tr>
<td>Clock cycles</td>
<td>197526</td>
<td>249726</td>
</tr>
</tbody>
</table>

### Outline

- Motivation
- Quasi-Static Scheduling of process networks and Petri nets
- Free-choice nets
- Non-free choice nets
- Conclusions
**Extension beyond FCPNs**

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

**Finding a Schedule on the Petri Net**

- Distinguished node, r, associated with the initial marking.
- All and only transitions in conflict from each node.
- A path to node r from each node.
Finding a Schedule on the Petri Net

Finding a Schedule on the Petri Net
Generating a Function Block

Properties of the Algorithm

• Claim 1:
  If the algorithm terminates successfully, then a schedule is obtained.

• Claim 2:
  If the algorithm does NOT terminate successfully, then no schedule exists under given termination conditions.

• Decidability is an open question for non-free-choice Petri nets
Improving Efficiency

- Which transition should be chosen at each node?
  - Find sequences of transitions to create cycles.

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

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

  **T-invariants:**

  \[
  \begin{bmatrix}
  0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
  2 & 2 & 0 & 0 & 1 & 0 & 0 & 0 
  \end{bmatrix}
  \]

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

PFC Example

![PFC Example Diagram](image-url)
Experimental Results

![Graph showing # of clock cycles vs. size of channels for 1-task and 4-task implementations]

Conclusions

- Quasi-Static Scheduling minimizes run-time overhead with respect to Dynamic Scheduling by Automatic partitioning of the system functions into a minimum number of concurrent tasks
  - sequentialize concurrent operations
  - data-dependent controls, multi-rate operations
  - technology-independent preprocessor
- Open issues:
  - correlated data-dependent controls
  - heuristic evaluation on T-invariant selections
Selected (not complete) bibliography on
Hardware Design and Petri Nets

General overviews and monographs


Hardware Modelling with Petri Nets


Synthesis of Petri nets and direct implementation


STG-based synthesis of asynchronous circuits


Verification of asynchronous circuits (unfoldings)

- J. Espanza. Bibliography on unfolding techniques. (http://wwwbrauer.informatik.tu-muenchen.de/gruppen/theorie/pom/#biblio)
• A. Semenov, A. Yakovlev, E. Pastor, M. Peña, J. Cortadella, and L. Lavagno. Partial order approach
to synthesis of speed-independent circuits. Proc. 3rd Int. Symp. on Advanced Research in
254-265.

• W. Vogler, A. Semenov and A. Yakovlev. Unfolding and Finite Prefix for Nets with Read Arcs.

Verification of asynchronous circuits (symbolic and timed automata)

• E. Pastor, O. Roig, J. Cortadella, and Badia, R.M. Petri Net Analysis Using Boolean Manipulation.

• H. Hulgaard and S.M. Burns, Bounded delay timing analysis of a class of CSP programs with choice.
Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 2-11,
Nov. 1994.

• O. Maler and A. Pnueli, Timing analysis of asynchronous circuits using timed automata. In P.E.
Verlag, 1995.

• O. Roig, J. Cortadella and E. Pastor, Verification of Asynchronous Circuits by BDD-based Model
Checking of Petri Nets, Application and Theory of Petri Nets 1995, Lecture Notes in Computer

• E. Pastor, J. Cortadella, M.A. Penya. Structural Methods to Improve the Symbolic Analysis of
Petri Nets 1999, 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA, pages

• M.A. Peña, J. Cortadella, A. Kondratyev and E. Pastor, Formal verification of safety properties
in timed circuits, Proc. Sixth International Symposium on Advanced Research in Asynchronous

Petri nets and Hardware Description Languages

• I. Bluuno, L. Lavagno. Deriving Signal Transition Graphs from Behavioral Verilog HDL, in: A.
Yakovlev, L. Gomes and L. Lavagno (Eds.) Hardware Design and Petri Nets. Kluwer Academic

Int. Symp. on Adv. Research in Asynchronous Systems and Circuits (ASYNC’2000), Eliat, Israel,

Quasi-Static Scheduling in codesign

• M. Sgroi, L. Lavagno, Y. Watanabe and A. Sangiovanni-Vincentelli. Quasi-Static Scheduling of
Embedded Software Using Equal Conflict Nets, in: Donatelli, S. and Kleijn, J. (Ed.): Proc. of
20th International Conference on Application and Theory of Petri Nets, ICATPN ’99, Annual
Conference, ICATPN’99, Williamsburg, Virginia, USA, LNCS, Vol. 1630, pp. 208-227, Springer-
Verlag, June 1999.

• J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, A.
Sangiovanni-Vincentelli Task Generation and Compile-Time Scheduling for Mixed Data-Control
Embedded Software, Design Automation Conference, DAC’2000,
Performance analysis


