Tutorial: Synthesis of control circuits from STG specifications

- Lecturers:
  - Jordi Cortadella (University Politècnica de Catalunya, Spain)
  - Michael Kishinevsky (The University of Aizu, Japan)

- in collaboration with:
  - Alex Kondratyev (The University of Aizu, Japan)
  - Luciano Lavagno
    (Cadence Berkeley Labs, USA and Politecnico di Torino, Italy)
  - Alex Yakovlev (University of Newcastle upon Tyne, UK)

Thanks to Peter Vanbekbergen for previous contributions to this tutorial and to Enric Pastor and Oriol Roig for the development of tools (petrify and versify)
Outline = Design Flow

- Specification
- Synthesis conditions
- State encoding
- Logic decomposition
- Technology mapping

- Verification
- Testing

Appendix 1: Hazards and races

Appendix 2: Other design styles

To be presented on Thursday 11:00-12:30
Part 1: Specification of control and synthesis overview

- Formalizing Timing Diagrams
- Signal Transition Graph (STG) model
- Signal Transition Graphs and Petri Nets
- Synthesis overview
- Example with concurrency (handshake communication)
- Example with choice (SR latch with completion)
From Timing Diagram to Formal model

TIMING DIAGRAM

input a
output c
input b

STG

\[ \begin{align*}
  & a^+ & b^+ \\
  & a^- & b^- \\
  & c^+ & c^- \\
\end{align*} \]
SIGNAL TRANSITION GRAPH(2)

- Vertices represent signal transitions.
- Arcs represent causal relations between transitions.
- Introduced by [Molnar 85] [Chu 87] [Rosenblum 85]
STG is an interpreted Petri net.
### CONSISTENCY

<table>
<thead>
<tr>
<th>INCONSISTENT</th>
<th>CONSISTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image_url" alt="Image" /></td>
<td><img src="image_url" alt="Image" /></td>
</tr>
<tr>
<td>A+ [---] A+ [---] A+</td>
<td>A+ [---] A- [---] A+</td>
</tr>
<tr>
<td>A- [---] A- [---] A-</td>
<td>Two up-transitions with a down-transition in between (and vice versa)</td>
</tr>
<tr>
<td>X+</td>
<td></td>
</tr>
<tr>
<td>A- [---] A+</td>
<td></td>
</tr>
<tr>
<td>Parallel transitions of the same signal</td>
<td></td>
</tr>
</tbody>
</table>

Two subsequent up-transitions without a down-transition in between

Two subsequent down-transitions without an up-transition in between
FIRING SEMANTICS

- \( \bullet \) = TOKEN
- \( \bigcirc \) = PLACE
- \( c^+ \) = TRANSITION

\( c^+ \) = enabled

\( c^+ \) has fired

\( c^+ \) fires

Synthesis of control circuits from STGs
**CONCURRENcy**

- c+ leads to a- concurrent with b-

- EITHER a- fires before b-
- OR b- fires before a-
- OR a- and b- fire simultaneously

**choicE**

- Choice made by environment

- EITHER a- fires
- OR b- fires
- But not both
Specification with choice

Set-dominant SR-latch

1. $Q$ goes high when $S$ goes high

2. $Q$ goes low when $R$ goes high
   while $S = 0$

Environment determines when

to apply $s_1+$ and $r_1-$
Synthesis overview

- Signal Transition Graph
  - Translation
  - Petri Net
    - Token flow simulation
  - Reachability Graph
    - Encoding
  - State Graph
  - Karnaugh Map
    - Logic synthesis
  - Logic equations

*direct method*

---

Summer school. Lyngby 1997

Synthesis of control circuits from STGs
REACHABILITY GRAPH

- Vertices represent states the circuit can be in.
- Arcs represent transition from one state to another.
- Can be derived from the Petri net by simulating the token flow.
- Easiest representation to do analysis on.
- Size may be exponential in the number of transitions.
- Similar to trace structures [Dill], transition diagrams [Muller].
Reachability Graph

- Vertices – circuit states
- Arcs – transitions
- Token flow simulation
- Easy for analysis
- Can be exponential
Every state is assigned a code.

This code is determined by signals in the Petri net.

Based on this code, logical equations can be derived.
Summerschool. Lyngby 1997 Synthesis of control circuits from STGs
Next State Functions for Signals

- State 000: $c$ is assigned 0 because $c$ stays low in this state.

- State 011: $c$ is assigned 1 because $c$ goes high in this state ($c^+$ is enabled).
EQUATIONS

STATE GRAPH

K-MAP

LOGIC EQUATIONS

c = ab + ac + bc

Summer school. Lyngby 1997 Synthesis of control circuits from STGs
Example: half-handshake circuit

A data processing structure [T. Meng 91] and partial and complete STG specifications of the "half-handshake" control.
Example: full-handshake circuit

Partial and complete two-phase STG specifications of the "full-handshake" control. A four-phase equivalent of the two-phase specification.
Example: SR-latch with completion detection

Specification and implementation
Limitations

- The advantages of STG:
  - Concurrency of signal transitions
  - Conditional behavior based on signal transitions

- The disadvantages of STG:
  - Conditional behavior based on signal levels
  - No timing information, just relative ordering of events
  - OR-type behavior is hard to model

- Different extensions have been proposed
Assignment of value to signal (with or without transition)
The behavior of the signals supporting boolean guards must be specified in the STG.
Part 2: Synthesis conditions

- Derivation of logic equations

- Synthesis conditions for implementability: boundedness, consistency, CSC

- Problems with delays (example: C-element)

- What is speed-independence? SI = QDI

- Synthesis conditions for SI (persistency, commutativity)
Synthesis by STG

- Route 0 – direct translation method

- Route 1 and 2 – synthesis based on transformations (adding signals, arcs, etc.)

- Which route to take? This tutorial is about route 2.
Next State Functions for Signals

- State 000: $c$ is assigned 0 because $c$ stays low in this state.

- State 011: $c$ is assigned 1 because $c$ goes high in this state ($c^+$ is enabled).
EQUATIONS

STATE GRAPH

K-MAP

LOGIC EQUATIONS

\[ c = ab + ac + bc \]

Summer school. Lyngby 1997 Synthesis of control circuits from STGs
Synthesis conditions

- A reachability graph can be infinite
- Binary encoding of a reachability graph not always possible
- A State graph can have conflicts for logic functions
- Logic can be hazardous (not speed-independent)

Checking synthesis conditions is necessary.
Boundedness of Petri Nets is decidable.

Boundedness $\iff$ finite reachability graph

Summer school. Lyngby 1997

Synthesis of control circuits from STGs
Consistent state assignment

<table>
<thead>
<tr>
<th>INCONSISTENT</th>
<th>CONSISTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+ \longrightarrow \rightarrow A+</td>
<td>A+ \longrightarrow \rightarrow A- \longrightarrow \rightarrow A+</td>
</tr>
<tr>
<td>A- \longrightarrow \rightarrow A-</td>
<td>Two up-transitions with a down-transition in between (and vice versa)</td>
</tr>
<tr>
<td>x+</td>
<td>Two subsequent up-transitions without a down-transition in between</td>
</tr>
<tr>
<td>A- \longrightarrow \rightarrow A+</td>
<td>Two subsequent down-transitions without an up-transition in between</td>
</tr>
<tr>
<td></td>
<td>Parallel transitions of the same signal</td>
</tr>
</tbody>
</table>

... → a+ → b+ → a+ → ...

Example:

ab: 0*0 \rightarrow a+ 10* \rightarrow b+ 11 \rightarrow a+ ?1

Boundedness + CSA ⇔ finite binary state graph
Complete State Coding (CSC)

CSC-conflict: two states have the same binary code and different sets of enabled *non-input* signals
Solving CSC by concurrency reduction

Non-equivalent transformation – concurrency is reduced!

Summer school. Lyngby 1997 Synthesis of control circuits from STGs 32
Solving CSC by signal insertion

Equivalent transformation – all original traces are preserved
Preserving interface

On $I \cup O$ circuit shows the same language as STG.

- **I/O-implementability**
  Circuit and STG have exactly the same $I \cup O$ set. Interface is preserved.

- **Implementability**
  New inputs and outputs can be added to a circuit. Interface is not preserved.

Summer school. Lyngby 1997 Synthesis of control circuits from STGs
I/O implementability

- **Irreducible CSC conflicts:**
  cannot be solved without changing interface.
  In Petrify use \texttt{-timed} option

- **All CSC-conflicts are reducible if STG is:**
  - Deterministic
  - Commutative
  - No complementary input sequences

\[
\begin{align*}
(a_1 + a_j) & \quad (s_1 + s_3) \\
b+ & \quad a+ \\
s_2 & \quad s_3 \\
s_4 & \quad s_5
\end{align*}
\]

- must be different signals
- must be the same state

**Boundedness + CSA + CSC $\leftrightarrow$ Logic circuit**
Speed-independence (D. Muller’56)

There are no disabled transitions $1^* \rightarrow 1$ and $0^* \rightarrow 0$ in the state graph

Circuit is speed-independent

Muller’s model assumes the unbounded gate delay model
Checking speed-independence

Circuit is not speed-independent
(or, more precisely, not semi-modular)

Analysis for speed-independence: (D. Muller'56)
(1) traverse states and
(2) check that no SI-conflicts are reachable
C-element on basic gates

Is this implementation correct?

Answer: It depends on the delay model and the model of the environment.
Delay assumptions

- *Delay 1* is a propagation delay from node 1 via environment to node 3.
- *Delay 2* is a propagation delay from node 1 via internal feedback to node 3.
- *Delay 1* must be greater than *Delay 2*.

Delay assumptions influence correctness of asynchronous design.
CMOS implementation of a C-element

Are these implementations correct?

Careful design is required (layout, transistor sizing, charge sharing)
What influences the design methodology

- Environment model
- Delay model
- Signalling model
- Specification language
Delay models

- **Place:** gates, wires, gates + wires

- **Timing properties:** bounded and unbounded
  - **Bounded:** lower and upper delay bounds are known
  - **Unbounded:** delays are unknown but finite

- **Memory properties:** inertial and pure
  - Intermediate models are possible
Gate vs wire delay models

- Gate delay model = delays are in gates, no delays in wires
  More realistic:
  1. gates have arbitrary delays
  2. the skew of wire delays after fork is less than gate delay

- Wire delay model = gate+wire delay model:
  Wires (and gates) may have arbitrary delays
Delay models and correctness

- This design is correct for unbounded \textit{gate} delay model
- This design is incorrect for unbounded \textit{wire} delay model
  If $d_1 > d_3 + d_5$ or $d_2 > d_3 + d_4 \Rightarrow$ premature change on $c$
- For the bounded delay model correctness depends on delay constraints.
Design styles for asynchronous control logic

- **Bounded delays (BD):** realistic for gates and wires.
  Technology mapping is easy.
  Verification is difficult since time must be considered.

- **Speed-independent (SI):** pessimistic delay model for gates (unbounded) and often acceptable for wires (the skew of delays after fork is less than gate delay).
  Technology mapping is more difficult.
  Verification is easy since time is excluded.

- **Delay-insensitive (DI):** pessimistic delay model for gates and wires (both are unbounded).
  DI class (built out of basic gates) is almost empty.

- **Quasi-delay insensitive (QDI) =** delay-insensitive except critical wire forks, called *isochronic forks*.
  In practice this is the same as speed-independent.

\[ BD \quad DI \quad SI = QDI \]
Persistency and commutativity

Persistency:  
\[
\begin{align*}
s_1 & \xrightarrow{a} s_0 \xrightarrow{b} s_2 \\
& \quad \downarrow b \quad \downarrow a
\end{align*}
\]
Commutativity:  
\[
\begin{align*}
s_1 & \xrightarrow{a} s_0 \xrightarrow{b} s_2 \\
& \quad \downarrow b \quad \downarrow a \\
& \quad s_3 = s_4
\end{align*}
\]

In speed-independent design *hazard-free = no disabling*

**Signal Persistency:** (1) Non-input signals cannot be disabled and
(2) Inputs cannot be disabled by non-inputs

Bound. + CSA + CSC + Persist. + Comm. \iff SI-circuit
## Violations of implementability

<table>
<thead>
<tr>
<th>Violation of</th>
<th>Meaning</th>
<th>How to correct</th>
<th>Type of implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundedness</td>
<td>Infinite RG</td>
<td>Change spec</td>
<td>—</td>
</tr>
<tr>
<td>Consistency</td>
<td>No binary SG</td>
<td>Change spec</td>
<td>—</td>
</tr>
<tr>
<td>CSC (irreducible)</td>
<td>No logic</td>
<td>Add signals</td>
<td>new $I \cup O$</td>
</tr>
<tr>
<td>CSC (reducible)</td>
<td>No logic</td>
<td>Add signals</td>
<td>same $I \cup O$</td>
</tr>
<tr>
<td>Persistency</td>
<td>Hazards or arbiters</td>
<td>Change spec</td>
<td>not SI or non-determ.</td>
</tr>
</tbody>
</table>
Part 3: Complete state coding

- Signal insertion using regions of states
- SIP-sets (Speed-Independence Preserving sets)
- I-partition and search game
- Estimation of logic
- Examples
Solving CSC by signal insertion

Equivalent transformation – all original traces are preserved
Event insertion

- Choose a set of states = excitation region for a new signal $x$
- Delay all exit transitions until $x$ fires
- Preserve trace equivalence (or bisimilarity) and speed-independence
Insertion of state signals (I-partition)

- $S^+$ and $S^-$ must be SIP (speed independence)
- I-partition must not have illegal arcs (consistency)
  (illegal: $S0 \rightarrow S1$, $S^+ \rightarrow S0$, $S1 \rightarrow S^+$, etc.)
- The event insertion must preserve the behavior of the environment (no exit arcs of input events from $S^+$ and $S^-$)
SIP violations

Violations of persistency

\[ A: \begin{align*}
& s_1 \\
& s_2 \\
& s_3 \\
& s_4
\end{align*} \quad A': \begin{align*}
& s_1 \\
& s_2 \\
& s_3 \\
& s_4
\end{align*} \]

Violations of commutativity

\[ A: \begin{align*}
& s_1 \\
& s_2 \\
& s_3 \\
& s_4
\end{align*} \quad A': \begin{align*}
& s_1 \\
& s_2 \\
& s_3 \\
& s_4
\end{align*} \]
Legal SIP-sets for a state diamond

(a) (b) (c) (d) (e) (f)
Regions and excitation regions

- **Region** = set of states: each event $a$ can either enter, or exit, or non-cross a region [Ehrenfeucht90, Nielsen92]

- **Pre-region** = $a$ exits $r$. **Post-region** = $a$ enters $r$.

- **Excitation region** $ER(a)$: all states that are exited by $a$; $ER(a)$ is an intersection of pre-regions of $a$

Any region is a union of minimal regions
Selecting SIP-sets

- SIP-sets must be used as excitation regions for new events

- The number of SIP-sets is exponential from the number of states

- Solution: For building SIP-sets use regions and their intersections instead of individual states

- Why this is sound?
USING REGIONS TO SELECT SIP-SETS

Legal SIP-set:

- Region

- Persistent excitation region

- Intersection of pre-regions (if forward connected)

PN-interpretation:
1. Find a bi-partition \( \{b, \bar{b}\} \)

2. Calculate \( EB(b) \) and \( EB(\bar{b}) \) (similarly for \( IBs \))

3. Extend \( EBs \) by backward closure

4. Verify SIP and no exit arcs for input events
Manipulating regions for signal insertion

(a) \[ p_1 \ p_2 \ p_3 \ p_4 \ p_5 \ p_6 \]

(b) \[ \text{ER}(x) = \text{MWFE}\text{B}(p_1) \]

(c) \[ \text{ER}(x) = p_1 \cap p_2 \]

(d) \[ \text{ER}(x) = \text{MWFE}\text{B}(p_4 \cap p_5 \cap p_6) \]

(e) \[ \text{ER}(x) = \text{MWFE}\text{B}(p_4 \cap p_5) \]
The bricks of a block

bricks:
- minimal regions
- intersection of pre-regions of the same event
- intersection of post-regions of the same event

block = \bigcup \text{bricks}
Heuristic search: a game-playing strategy

bricks = calculate_all_bricks ()
frontier = good_blocks = {the best FW bricks}
repeat /* heuristic search */
  new_frontier = ∅
  for each bl ∈ frontier do
    for each br ∈ bricks adjacent to bl do
      new_bl = bl ∪ br
      if cost(new_bl) < cost(bl) then
        good_blocks = good_blocks ∪ {new_bl}
        new_frontier = new_frontier ∪ {new_bl}
        frontier = select the best FW blocks from new_frontier
  until new_frontier = ∅
return the best block in good_blocks
Example: bipartition for CSC using regions

Summer school. Lyngby 1997 Synthesis of control circuits from STGs
Example: I-partition for CSC using regions

1. Find bi-partition \( \{b, \overline{b}\} \)

2. Find by expansion \( EB(b) \) and \( EB(\overline{b}) \) which are well-formed and SIP
Cost function

Comparison of two I-partitions:

- Correctness (SIP, well-formedness, behavior of the environment)
- Number of CSC conflicts solved
- Estimation of logic

⇒ Trade-off between CSC conflicts and estimated logic
Estimation of logic

ER(x+)

ER(a+) \Rightarrow \overline{b}
ER(a-) \Rightarrow b

locked signals
Solving examples with CSC-conflicts

---

Summer school. Lyngby 1997
Synthesis of control circuits from STGs
Solving examples with large state space

par16: 83 places, 68 trans., 34 (50) signals
1.5 \times 10^{11} \ (2.8 \times 10^{12}) \ states \Rightarrow < 4h. \ CPU \ time
Manipulating concurrency of inserted signals

(a)
(b)

ram-read-sbuf: (a) sequential solution (area = 406)
(b) after increasing concurrency (area = 456)

Summer school. Lyngby 1997 Synthesis of control circuits from STGs 68
# Features of region-based method

- Computationally efficient \((\text{search is guided by regions})\)

- Allows symbolic manipulation with SG \((\text{BDD-based})\)

- Complete \((\text{All CSC conflicts are solvable for safe STG})\)

- Represents the result in STG \((\text{STG} \rightarrow \text{SG} \rightarrow \text{STG})\)

- Implemented in **petrify** tool
Part 4: Logic decomposition

- Problem definition
- Monotonous cover
- Decomposition based on algebraic division
- Decomposition based on Boolean relations
- Synthesis for bounded gate and wire delays
Hazard-free synthesis

Each gate firing is acknowledged

STG with CSC

Unbounded delays

Bounded delays

Delay Padding

Hazard-free circuit

Hazards removal
History: synthesis with unbounded gate delays

- Muller (59): formal analysis method, but only design hints

- Martin, Burns (86, ...): syntax-directed translation, then QDI-preserving optimization

- Varshavsky et al. (90): syntax-directed translation from STG

- Beerel (92) and Kondratyev et al. (93, 95): necessary and sufficient conditions for monotonous switching, STG- and SG-level synthesis techniques based on signal insertion.
History: logic decomposition for SI design

- Kishinevsky, Varshavsky (82) – basis of 2-input gates (area overhead – only theoretical interest)

- Beerel et al. (ICCAD’ 92) – general acknowledgement (based on heuristics, no guarantee to find a solution even when it exists)

- Siegel et al. (ICCAD ’94) – only local decomposition (simple cases, no completeness)

- Myers et al. (London ’95) – decomposition + resynthesis (local acknowledgement, no effective solution)

- Burns (Async ’96) – local two-level decomposition (exact and efficient solution, single ack. restriction)

- Cortadella et al. (Async’97, ICCAD’97) – multilevel decomposition with multiple acknowledgement based on factoring and Boolean relations
Problem formulation for logic decomposition

**Problem:** hazards may appear =

some transitions may not be acknowledged ⇒

the result is not speed-independent
Sometimes classical decomposition works ...

**STG**

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z

STG

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z

STG

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z

STG

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z

STG

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z

STG

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z

STG

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z

STG

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z

STG

c- $\Rightarrow$ d-

d+ b+
z- d+
b- y+
z+ a-
c+ y-
a+ c+
c- $\Leftarrow$ d-

global ack.

db

cba

local ack.

x

y

z
Driving and supporting gates

- All internal signal transitions must propagate to the output (acknowledgement principle is applied to individual gates)

- Driving gates: “good”, supporting gates: “bad”
Implementation structures

Signal implementation

C-latch

Each signal transition corresponds to an AND gate

- $a+$ sets the latch
- $a-$ resets the latch

BASIC ARCHITECTURES

RS flip-flop

with input inverters

without input inverters
Regions of a state graph

Synthesis idea:

transition ⇒ cover cube in R or S function ⇒ driving gate

- Excitation region (ER) must be covered by one cube
- Quiescent region (QR) is a don’t care set for a cover cube
Monotonous cover conditions

Cover cube $C$ is a **monotonous cover** for $ER(a^*)$ iff:

1. $C$ covers all states $ER(a^*)$
2. $C$ covers no states outside $ER(a^*) \cup QR(a^*)$
3. $C$ changes only once inside $QR(a^*)$

- Monotonous cover $\Rightarrow$ SI-implementation on simple gates

- Restricted (but useful and realistic) architecture
Monotonous cover example

If a cover cube is not MC ⇒ reduce it by adding literal

If it does not help ⇒ add signals
Reduction to monotonous cover form

- $ER(d^+) \text{ cannot be covered by one cube}$
  (minimal cover for $ER(d^+) \text{ is } b^+ a\overline{c}$)

- After inserting signal $x$:
  minimal cover for $ER(d^+) \text{ is } x\overline{c}$

- Any MC-conflicts can be removed by adding signals

Trace equivalence is preserved
Extended MC conditions

Basic MC conditions are limited:

- Unique entry condition:
  (No OR-causality is allowed)

- No complex gates in SOP logic

- Sufficient but not necessary
  (one literal cubes, don’t cares for C-elements)

Monotonous cover ⇒ Extended monotonous cover
Global acknowledgement is important

STG

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

\[ \text{STG} \]

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

Local ack.

Global ack.

\[ \text{STG} \]

\[ \begin{align*}
\text{c} &\rightarrow \text{d} \\
\text{d} &\rightarrow \text{b} \\
\text{b} &\rightarrow \text{z} \\
\text{z} &\rightarrow \text{d} \\
\text{d} &\rightarrow \text{a} \\
\text{a} &\rightarrow \text{c} \\
\text{c} &\rightarrow \text{d}
\end{align*} \]
Types of decomposition

Sequential

Combinational

Complex gate

C(a)

C(b)

Complex gate

C(a)

C(b)

Complex gate

C(a)
Algebraic factorization

Algebraic division $C' = F \times G + R$

- Cover: $C = abc + abd$
- Kernel, cokernel: $C = ab(c + d)$
- OR decomposition: $C' = (abc) + abd$
- AND decomposition: $C' = (ab)c + abd$
I-partition defined by a function

- Divisor ⇒ State space partition
  \[ S = \{F^0, IB(F^+), F^1, IB(F^-)\} \]

- SIP conditions to insert events (speed-independence)

- Input border closure
  - Preserve consistency
    \((x^+ \text{ and } x^- \text{ alternate})\)
  - Preserve I/O interface
    \((\text{input signals are not delayed})\)
Example: construction of SIP sets

- a, d - outputs
- b, c, e - inputs

**IB closure**

\[ \text{ER}(x^+) = \text{IB}(F^+) \]

**SIP closure**

- \( F = a \)

**I/O closure**

- \( \text{ER}(x^+) = \text{IB}(F^+) \)
Progress in decomposition

\[ C'(a) = F \ast G + R \Rightarrow \]
\[ x = F; \quad C''(a) = x \ast G + R \]
\[ x - \text{inserted signal} \]

- SIP guarantees speed-independence for signal \( x \)
- Does not guarantee speed-independence of \( C''(a) \) (progress conditions)
- To acknowledge \( x \) other functions can be changed (estimation of decomposition cost)

---

Summer school. Lyngby 1997 Synthesis of control circuits from STGs 88
Sequential decomposition

Shortcomings of combinational decomposition:

- Only $x+$ is used for simplification (no useful job by $x-$)
- No freedom for choosing $ER(x-)$ (defined by divisor $F$)

Correctness conditions:
1. $D_1 \cap D_2 = \emptyset$
2. $Succ(D1) \subset IB(P)$
3. $Succ(D2) \subset IB(F)$
Example of decomposition. I
Example of decomposition. II
Decomposition based on Boolean Relations: example (I)

\[ y = \overline{R} (S+y) \]
Decomposition based on Boolean Relations: example (II)

\[ y = acd' + y(c' + d') \]

\[
\begin{array}{c|cccc}
\text{ac} & 00 & 01 & 11 & 10 \\
\hline
\text{dy} & 00 & 1 & 0 & 1 \\
00 & 0 & 0 & 1 & 0 \\
01 & 1 & 1 & 1 & 1 \\
11 & 1 & 0 & - & 1 \\
10 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cccc}
\text{ac} & 00 & 01 & 11 & 10 \\
\hline
\text{dy} & 00 & 1- & -0 & -0 \\
00 & 1- & -0 & 01 & -0 \\
01 & 0- & 0- & 0- & 0- \\
11 & 0- & 1- & -- & 0- \\
10 & 1- & -0 & 1- & -0 \\
\end{array}
\]

\[
\begin{align*}
0 \to 0: & \quad RS=1-,0- \\
0 \to 1: & \quad RS=01 \\
1 \to 0: & \quad RS=1- \\
1 \to 1: & \quad RS=0- \\
\end{align*}
\]

R=cd; S=acd' 
R=cd+c'y'; S=a

---

Summer school. Lyngby 1997 
Synthesis of control circuits from STGs
**Boolean Relations for different latches and gates.**

<table>
<thead>
<tr>
<th>Region</th>
<th>Trans.</th>
<th>C-element $H_1, H_2$</th>
<th>D-latch $C, D$</th>
<th>Rs $R, S$</th>
<th>Sr $S, R$</th>
<th>AND $H_1, H_2$</th>
<th>OR $H_1, H_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER($y+$)</td>
<td>0 → 1</td>
<td>11</td>
<td>11</td>
<td>01</td>
<td>1−</td>
<td>11</td>
<td>{1−,−1}</td>
</tr>
<tr>
<td>QR($y+$)</td>
<td>1 → 1</td>
<td>{1−,−1}</td>
<td>{0−,−1}</td>
<td>0−</td>
<td>{1−,−0}</td>
<td>11</td>
<td>{1−,−1}</td>
</tr>
<tr>
<td>ER($y−$)</td>
<td>1 → 0</td>
<td>00</td>
<td>10</td>
<td>1−</td>
<td>01</td>
<td>{0−,−0}</td>
<td>00</td>
</tr>
<tr>
<td>QR($y−$)</td>
<td>0 → 0</td>
<td>{0−,−0}</td>
<td>{0−,−0}</td>
<td>{1−,−0}</td>
<td>0−</td>
<td>{0−,−0}</td>
<td>00</td>
</tr>
<tr>
<td>unreach.</td>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

C element: \( y = H_1H_2 + y(H_1 + H_2) \)  
Sr latch: \( y = S + R_y \)  
D latch: \( y = CD + \bar{C}y \)  
AND gate: \( y = H_1H_2 \)  
Rs latch: \( y = \bar{R}(S + y) \)  
OR gate: \( y = H_1 + H_2 \)
Technology mapping

- Logic decomposition
- Graph covering based on
  - Tree matching
  - Boolean matching

References:
- Keutzer 87
- De Micheli 96.

Additional complexity for SI:
- SI-preserving decomposition
Library for technology mapping

Gate

DAG

Gate

DAG

Summer school. Lyngby 1997  Synthesis of control circuits from STGs
Technology mapping: graph covering
Synthesis of bounded-delay circuits

- Lavagno’91
- Nowick’92
- Myers’94

- One combinational logic block for each STG non-input signal
- Decouple hazards from races
Synthesis algorithm properties

- Two classes of hazards:
  - from concurrent transitions
  - from ordered transitions

- The initial synthesis algorithm guarantees no hazards from concurrent transitions (condition similar to FSM)
  
  (for concurrent \( z^+ \) and \( y^+ \) cube \( x \) is added to \( f_z \) e.g.)
Remaining hazards

- Ordered transitions: $y+ \rightarrow z-$  
  ($x$ should not be affected)

- Corresponding circuit:

- Hazard if $d_{y\Rightarrow x} > d_{y\Rightarrow z} + d_{z\Rightarrow x}$ (transition order reversed for $x$)
Hazard elimination idea

- For each case in which a hazard exists if \( d_y \Rightarrow x > d_y \Rightarrow z + d_z \Rightarrow x \)
  - if possible, balance \( d_y \Rightarrow x \) and \( d_z \Rightarrow x \)
  - otherwise, increase \( d_y \Rightarrow z \)

- Eliminating one hazard does not make any other hazard worse:
  - \( d_y \Rightarrow x \) and \( d_z \Rightarrow x \) are measured before the padded delay
  - \( d_y \Rightarrow z \) is measured after the padded delay
Analogy with synchronous circuits

- In classical synchronous circuit synthesis:
  - slow down *the clock* until *all events* complete propagation

- In bounded delay asynchronous circuit synthesis:
  - slow down *each signal* until *all events that caused its transition* complete propagation

- We can use similar timing analysis techniques for the combinational logic blocks...
Summary of logic synthesis

- Different algorithms for different delay models
  - neglecting wire delays, robust with respect to gate delays
  - considering wire delays but assuming bounds on them

- In any case, post-layout timing analysis is required

- Hierarchical decomposition:
  - bounded delays within blocks
  - unbounded wire delays between blocks

- Choosing the right granularity is important
Part 6: Verification and validation

- What is verification
- STG implementability
- Verification by state models
- Circuit analysis
- Fighting with state explosion
- Verification by event models
- Simulation
Verification

- Checking properties of a **specification**
  Example: Given an STG describing a bus protocol, check that
  - it does not deadlock (property verification)
  - it can be implemented by an asynchronous circuit (implementability)

- Checking properties of an **implementation**
  Example: Given a circuit check that
  - it is speed-independent (property verification)
  - it correctly implements a given STG specification (implementation verification)
Verification in design flow

Verification is needed both for top-down and bottom-up design
Verification of STG specifications

- **Property verifications**
  - Safety properties: deadlocks, mutual exclusion, concurrency, preceding between events etc.
  - Liveness properties: progress and fairness conditions
  - Structural invariants and traps based on theory of Petri Nets

- **Implementbility verifications**
  - Boundedness
  - Consistensy of state encoding
  - Completeness of state encoding, reducability of CSC-conflicts
  - Persistency
Fight with state explosion

- Efficient state representation
  - BDD-based techniques (Bryant, Cortadella et al.)
  - State space reduction (Kishinevsky)

- Use of event-based models
  - Unfoldings of Petri Nets and STGs
    (McMillan, Kondratyev et al., Esparza)
  - Stubborn sets (Valmari)
  - Partial orders (Probst, Godefroid)
  - Reduction of nets (Murata, Esparza)
State reduction

- Only transitions on gate inputs may cause an output transition
- Micropipeline control queue: run time reduced from 1 hour to 1 sec
- Efficient for fighting with concurrency. Choice is difficult to handle
BDD representation of reachable states

All reachable markings (reachability set):
\[ p_1 \overline{p_2} \overline{p_3} + \overline{p_1}p_2p_3 + p_1 \overline{p_2}p_3 \]

The characteristic function of the reachability set can be represented by a reduced ordered BDD.
Symbolic traversal of STG

Each property is represented by a characteristic function

Full state = marking + binary state of signals

\[ M_0 = p_1 \overline{p}_2 \overline{p}_3 \]
\[ S_0 = p_1 \overline{p}_2 \overline{p}_3 \overline{a} \overline{b} \]

\[ E_t = \prod_{p_i \in t} p_i \quad \text{(t enabled)} \]
\[ \text{ASM}_t = \prod_{p_i \in t^*} p_i \quad \text{(succ. marked)} \]
\[ \text{NPM}_t = \prod_{p_i \in t^*} \overline{p}_i \quad \text{(pred. unmarked)} \]
\[ \text{NSM}_t = \prod_{p_i \in t^*} \overline{p}_i \quad \text{(succ.unmarked)} \]
Checking properties via traversal

One-step reachability: \( \delta_N(M, t) = (M_{E_t} \ast NPM_{t})_{NSM_t} \ast ASM_t \)

Consistency: \( \text{Inconsistent}(a) = E(a+) \ast a + E(a-) \ast \bar{a} \)

Persistency: \( \delta_N( E_{a+}, b+) \ast \bar{E}_{a+} \neq 0 \)

Specs with \(10^{23}\) states were checked by BDD-based traversal!
Verification in trace theory

Dill, Rem, Udding, van de Snepscheut, Ebergen

- Specification of Join:
  \[ \text{pref} \ast [a?; c!] \parallel \text{pref} \ast [b?; c!] \]
  \( a?, b? \) are input; \( c! \) is output

- Mirror of the spec = environment:
  inputs ↔ outputs:
  \[ \text{pref} \ast [a!; c?] \parallel \text{pref} \ast [b!; c?] \]

- Does a C-element implement the Join?

---

Summer school. Lyngby 1997 Synthesis of control circuits from STGs 113
Failure states and hazardous behavior

Hazardous behavior (non speed-independence) = failure states are reachable in the composite state graph

It also shows that the circuit does not implement the spec
Incorrect implementation

OR-gate is not an implementation for the Join

Failure state is reachable \(\Rightarrow\) an oscillator OR-gate + two inverters is not speed-independent

Compare with speed-independence analysis

Parallel composition of state graphs is inefficient
Verification via composition environment/circuit

ENVIRONMENT (Petri net)

CIRCUIT

V_I

V_O

Summerschool. Lyngby 1997 Synthesis of control circuits from STGs
Verification: failure states

Failure state: circuit produces unexpected output transition (enabled in the circuit, but disabled in the environment)

Diagnosis: find a trace of events from initial to failure state (important for circuit redesign)
Correctness is not a property of a circuit but of the interaction of a circuit with an environment.
Unfolded Petri net

- Occurrence net = an infinite unfolded Petri Net
- All forward conflicts are split
Truncating occurrence nets

- **Unfolding** is a finite prefix of the occurrence net

- **Completeness**: no reachable marking of PN is missing

- **Configuration** of $t' = \{ \text{all transitions that must fire before } t' \} (\text{Conf. } d' = \{a'\})$

- **Basic marking** of $t' = \{ \text{marking after all transitions from the local configuration of } t' \text{ have fired} \} (BM(d') = p5'p2')$

- **Cutoff** is a transition whose successors can be removed:
  
  if basic-marking($t'$) = basic-marking($t''$) and
  size-of-configuration($t'$) < size-of-configuration($t''$) then $t''$ – cutoff

  (More efficient criteria exist)
Unfoldings

- **Ordering relations** are easy to derive:
  - \(a'\) and \(c'\) are concurrent, \(a'\) precedes \(f'\), \(a'\) and \(e'\) are in conflict
- All markings are in unfolding \(\iff\) all PN properties can be studied
- In many cases it is more efficient than BDD traversal

**Basic markings of** \(f'\) **and** \(f''\) **are equal**

Ordering relations in unfolding can be analyzed with \(O(N^3)\) complexity
STG verification by unfolding: consistency

- **Sign alternation (SA)**
  
  \[ ... \rightarrow a^+ \rightarrow b^+ \rightarrow a^+ \rightarrow ... \]

- **Non-autoconcurrency (NA)**
  
  \[ ... \rightarrow d^- \rightarrow b^+ \rightarrow a^- \rightarrow ... \]

- **Proper assignment (PA)**
  
  (same basic marking – same binary state)

\[
\begin{array}{c}
\text{SA + NA + PA} \iff \text{STG is consistent}
\end{array}
\]

- Autoconcurrency = particular case of concurrency
- Sign alternation = particular case of precedence
- Proper assignment reduces to precedence checking
SIMULATION ENVIRONMENT

[Vanbekbergen 95]

BEHAVIORAL VHDL

ASSERTION VHDL

TESTBENCH VHDL

GATE-LEVEL VHDL

STG

SYSTEM

USER

OR

OR

VHDL SIMULATION

RESULTS

ASSERTION VIOLATIONS

Summer school. Lyngby 1997 Synthesis of control circuits from STGs 123
ARCHITECTURE

TEST BENCH

BEHAVIORAL VHDL OR GATE-LEVEL VHDL

ASSERTION VHDL

Error messages

inputs

outputs
state signals

Summer school. Lyngby 1997
Synthesis of control circuits from STGs
VHDL MODEL

- State = set of tokens in places
- A vector of Booleans: places
  - $places(1) = 1$ if place $p_1$ contains a token.
  - $places(1) = 0$ if place $p_1$ does not contain a token.
VHDL MODEL (2)

- Every transition possibly concurrent with any other transition
- ONE block per transition
- Token flow modeled inside the block
pcond controls
conditional execution

enables a different
transition for each
"pass"

BLOCK (places(1) = '1' AND pcond = '0')
BEGIN
  places(1) <= GUARDED '0';
  s1 <= GUARDED '1';
  pcond <= GUARDED '1';
END;

BLOCK (places(1) = '1' AND pcond = '1')
BEGIN
  places(1) <= GUARDED '0';
  s2 <= GUARDED '1';
  pcond <= GUARDED '0';
END;
assertion

- All blocks are possibly concurrent.
- In VHDL a resolution function is needed.
- STG is only correct if no two transitions of the same signal are enabled at the same time.
- Assertion in the resolution function that there are no two simultaneous assignments to the same signal.
ASSERTION(2)

\[
\text{ASSERT (s'\text{STABLE} = \text{TRUE OR places(1) = '1' OR ....)}}
\]

- Gives info about
  - where?
  - which signal?
  - which type of violation?

- Debugging circuit AND environment
TESTBENCH

- Testbench = behavioral VHDL of “inverse” STG (inputs and outputs switched).

- Use of same procedure as the one generating behavioral VHDL (not written by designer)

- Assumes that the environment is the fastest possible.

- Not complete (like verification), but many errors found this way.
Part 6: Testing

1. What do we know about asynchronous testing?

2. Type of faults and why do we need to test delay faults for "delay-insensitive" circuits

3. Robust path delay fault testing

4. How to achieve high testability: extra inputs, full scan, partial scan

5. Test generation in synchronous mode
Asynchronous circuit testing (1)

- Traditionally considered *difficult*

- Not much worse than synchronous case if a *
  systematic design methodology*
  is followed

- Delay-fault testing is generally required:
  *correctness only under specific delay hypotheses*
  (e.g., zero or bounded wire delay, isochronic forks, etc.)
Asynchronous circuit testing (2)

- Speed-independent circuits are self-checking (Armstrong 69, Varshavsky 76, Taubin 80, Beerel 91, ...), but only
  - for output stuck-at faults
  - if delay-faults do not invalidate the gate delay model
- Full-scan decomposition-based methodology (Keutzer 91)
- Full-scan phase splitting-based methodology (Lavagno 94)
- Partial-scan of sequential asynchronous nets (Kishinevsky 96)
- Synchronous ATPG for asynchronous circuits (Roig 97)
- Delay-fault testing for the Fundamental mode (Nowick 96)
- Fast test generation for handshake circuits (Roncken 94-96)
At the gate level: a FIFO latch is *input* stuck-at testable

After any technology mapping – the latch is not stuck-at testable
Problem 2: delay faults within a gate

D-element

Technology mapping

Delay fault: \( d(\text{inv}) > d(+u) \)

Signal Graph

\( +li \rightarrow +u \rightarrow +ro \rightarrow +ri \rightarrow -u \rightarrow -ro \rightarrow -ri \rightarrow -lo \)

(1) \( li \) stuck-at-0 is a premature fault (Hazewindus, 91):
\[ +li \rightarrow +u \rightarrow +r0 \]

(2) A delay-fault of the input inverter may cause hazards
Hazard-free testing vs. operation

- Hazard-free testing: for each fault:
  - there exists a sequence of vectors that do not cause hazards and test the fault under every timing condition
  - the vectors are produced by scan flip-flops:

- Hazard-free operation: for each vector sequence:
  - that conforms to the circuit specification
  - the sequence cannot cause a hazard under any timing condition
Definitions (1)

- Path: sequence of gates and wires

- Vector pair: pair of value assignments to inputs

- Event: $0 \rightarrow 1$ or $1 \rightarrow 0$ transition on a gate

- Event-sensitizable path: an event can propagate along it

- Robust propagation: independent of other delays
Definitions (2)

- **Path-delay fault**: an event is too slow or too fast propagating down the path.

- **Robust path-delay-fault test**: a pair of vectors that robustly propagates an event down the path. Hazards cannot force false positives.

- **Hazard-free robust path-delay-fault test**: a pair of vectors that robustly and without hazards propagates an event down the path.

---

Summer school. Lyngby 1997 Synthesis of control circuits from STGs 138
Path delay fault testing with extra test inputs to select paths

Luciano Lavagno and Kurt Keutzer, 1991
Redundant Boolean function $F$ with binate $x$: $F = xG + \bar{x}H + R$
Hazard-preserving transformations → minimize number of test points
Variable phase splitting

Signal $L$ is split into $L'$ and $L''$. The cover becomes prime and irredundant. Additional testing points are not needed.
Asynchronous scan latches are needed
Some of binary primary inputs may need to be split
Non-observable outputs may require standard scan D-latch
Partial scan based on asynchronous nets

- Asynchronous net: **feedback allowed only inside gates**
- Generalization of combinational circuit
- Can be obtained from asynchronous circuit by cutting global feedback (*partial scan*)
- Two-step procedure:
  1. Initialize all non-scan sequential elements, by means of a *vector sequence*
  2. Test the relevant path(s), by means of a *vector pair*
- Conditions for RPDFT of asynchronous nets can be formulated and checked efficiently by reduction to a combinational synchronous circuit
RPDFT testing of asynchronous nets

Replace each C element with *majority gate*
(additional input is never used for test generation)
Test pattern generation (1)

- Can be done by using *combinational stuck-at* ATPG techniques

- Transform the net into a *leaf-DAG* (fanout only from primary inputs)

- For rising transition, replace the initial lead of each reconvergent side-input to OR gate by 1

- Find a test for SA-0 on the initial lead

- Forces to find a test that keeps risky reconvergences that may course hazards “shut off”
Test pattern generation

Summer school. Lyngby 1997
Synthesis of control circuits from STGs
Initialization conditions

- (HF)RPDFT is valid only if initialization is consistent with test
- May not be trivial with *partial scan*
- Can always fall back to full scan...
- Classical procedure: unfolding and exhaustive enumeration
- *Monotonous initialization* can often be more efficient
Synchronous test patterns

Controlable inputs → Circuit → Observable outputs

Strategy for synchronous test:
- Apply input pattern
- Let the circuit stabilize
- Observe outputs

Good news:
- ATPG techniques for synchronous FSMs can be applied
- SI circuits are almost 100% input stuck-at testable

Bad news:
- Not all input patterns can be safely used
- Computationally expensive
Problems with synchronous test patterns

Non-confluence

Oscillation

Summer school. Lyngby 1997
Synthesis of control circuits from STGs
Finding synchronous test patterns

Non-confluence
Oscillation
Confluence

A+ C- 
A+ B+ 
A- B- 
D+ A- B-

s1
s2
s3
s4

Summerschool. Lyngby 1997 Synthesis of control circuits from STGs 149
Appendix 1: Hazards and races

- Types of hazards
- Combinational hazards
- Races and sequential hazards
- Hazard elimination
- Modern view on hazards and races
Hazards

- Hazard: signal transition not specified by the designer
- Hazards make asynchronous design difficult
- Eliminating hazards reduces power dissipation in synchronous design [Pedram]
- Theory of hazards: [Unger, Kung, Nowick]
Hazard analysis depends on the delay model

- Inertial gate delay model
  \[ \implies \text{no hazards} \]

- Wire delay model
  \[ \implies \text{hazards} \]
Taxonomy of hazards

- **Signal behavior:**
  - static (no transition is required)
  - or dynamic (monotonous transition is required)

- **Circuit type:**
  - combinational (circuits without feedbacks) or sequential (circuits with feedbacks)

- **Due to specification or due to implementation:**
  - function or logic; essential or not essential

- **Due to the environment behavior:**
  - next input transition is applied before the circuit settles
Combinational Hazards. Motivation

Clock arrives \textbf{after} combinational outputs settle

Clock period calculation: \textit{propagation + hold + set}

No clock $\Rightarrow$ wrong value can be written in a memory
Combinational hazards

- Function hazards: inherent in the specification of the logic function.
  
  Condition for 1-hazards: a transition cube contains 0
  Cannot be eliminated by changing the logic
  Each Boolean function of more than one variable has static hazards for some input transitions ⇒ restrict input changes

- Logic hazards:
  depend on the particular implementation of the logic function
  Can be eliminated by changing the logic (or sometimes the delays)
Function hazards

- Condition for 1-hazards: a transition cube contains 0
- Cannot be avoided by changing the logic

If input changes are not restricted, then any boolean function of more than one variable has static hazards

Restrict input changes ⇒
- single (SIC)
- multiple (MIC)
Static logic hazard elimination

Static hazard: a signal which should remain constant changes

Static logic 1-hazard elimination condition:
*each transition cube is contained inside a cube of the cover*

A sum of all prime implicants is free from static hazards
(for those transitions which are free from function hazards)

Static logic 0-hazards never occur in a two-level Sum-of-products
Dynamic function hazards

Dynamic hazard: a signal changes multiple times instead of once

Dynamic function hazard:
non-monotonous change of the function inside the transition cube
Dynamic logic hazard elimination

Dynamic logic 1to0-hazard elimination condition: if transition cube intersects cube $c$ of the cover, then $c$ contains the start point of the transition cube

SOP of all prime implicants has no dynamic hazards under SIC

0to1-hazards are symmetrical (the same for the end point)
Hazard-free combinational implementations

- Eichelberger
  (SOP of primes + SIC)

- Bredeson and Hulina
  (RS-flip-flop + factorization)

- Two-phase operation
  ($f$ and $\overline{f} + $ spacer)
Sequential circuits

Sequential circuits $\approx$ circuits with feedbacks

In synchronous circuits feedbacks are broken by:

- Edge-triggered flip-flops
- Two-phase nonoverlapping clocks
Sequential hazards

Sequential circuits \(\approx\) circuits with feedbacks

Sequential hazards occur because feedbacks propagate new values before the input net completes signal propagation to all inputs of the next state logic.

- **Essential hazard**: inherent in the FSM specification.
  
  cannot be eliminated by another state encoding \(\Rightarrow\)
  cannot be eliminated for unbounded delays
  
  can be masked out by controlling the delays

- **Non-essential hazards** (races):
  
  can be eliminated by a proper state encoding
Non-critical races

Race: more than one state signal changes

Non-critical race: all transient states drive the FSM into the same final state
Critical races

different transient states drive the FSM into different final states.

Final state depends on the race winner
Race-free encoding

Tracey encoding:

Two state transitions under the same input $s_1 \xrightarrow{b} s_2$ and $s_3 \xrightarrow{b} s_4$ must be distinguished by a stable state signal:

$011 \xrightarrow{10} 000$ and $110 \xrightarrow{10} 100$
Types of race-free encoding

- **Single bit change:** simple, bad throughput
  adjacent states differ by exactly one bit
  requires few FSM cycles to reach a stable state
  Example: one-hot encoding

- **Single transition:**
  better throughput, more complex
  adjacent states may differ by multiple bits
  requires one FSM cycle to reach a stable state
  Examples: equal distance encoding; Tracey encoding

Race-free FSMs may still have **essential hazards**
Mod-2 counter: essential hazards

Unger's condition for essential hazards:

state after **three** transitions of an input signal ≠
state after **one** transition of the same input signal
Why essential hazards occur

No essential hazards if and only if
\[ |delay_1 - delay_2| < mindelay_{next} + mindelay_{feedback} \]

Essential hazards can be removed by controlling the delays
Delay hazards

- I/O-mode: input \( b \) changes after output \( c \), but before gate 3 settles
- This is a delay hazard

Delay hazards occur in I/O-mode even in designs free from essential hazards
Hazard elimination methods

- Changing delays (delay padding).
  Works only for bounded delay model

- Logic transformations (circuit redundancy).
  Works both for bounded and unbounded delay models

- Encoding of inputs and outputs (code redundancy)
  and/or changing allowed input transitions.
  Works both for bounded and unbounded delay models

Fighting with hazards always requires some area and time overhead
Masking hazards by delay padding (1)

Controlling delays can eliminate hazards
Masking hazards by delay padding (2)

Contradictory constraints ⇒ controlling delays cannot eliminate hazards in general
Hazard elimination by redundant encoding

- Function hazards cannot be eliminated by changing the logic \( \Rightarrow \) change the specification.

- Hazards depend on the allowed input transitions: XOR-gate is hazard-free for single input changes.

- Redundant encoding of input and output signals and four phase signalling protocol allows to eliminate even functional hazards: XOR-gate is hazard-free. 
  \( Valid \rightarrow Empty \rightarrow Valid \)

Dual-rail four-phase XOR-gate
Modern view on hazards and races

Classical hazard/race taxonomy is complex for sequential circuits even in Fundamental Mode.

It is much more complex for the I/O-mode:

- Hazards may be viewed as signal transition disabling

- Critical races may be viewed as incomplete (ambiguous) specifications

Hazard avoidance: design a circuit in which transition disabling cannot occur

Critical race avoidance: make the specification complete and unambiguous
For unbounded gate delay model:
Disabling cannot occur \( \Rightarrow \) no hazards

No need for a complex theory of hazards for this delay model
Critical races as incomplete state coding

FSM spec is ambiguous, i.e. incomplete

State assignment conflict:

<table>
<thead>
<tr>
<th>$&lt; y_1, y_2, y_3 &gt;$</th>
<th>next state $y_2'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Complete state coding

Solving state assignment conflicts:
Add new state signals or
Remove one of the states (by re-encoding, reducing concurrency)
Appendix 2: Review of other methods of synthesis

- CSP-based approach [Martin, Burns, et al]

- Burst Mode finite state machines [Davis et al, Nowick, Yun, Dill]
Other specification models. CSP

- CSP = communicating sequential processes.

- Concurrent processes communicating by input and output commands on channels.

- Semantics are based on trace theory. [Udlding, van de Snepscheut, Rem]

- In general, techniques try to come up with quasi-delay-insensitive implementations

- Two main approaches: Production rule based [Martin, ...] and Syntax-directed decomposition [Ebergen, Berkel]
CSP PROGRAM

\[
*[[l_i]; r_o \uparrow; [r_i]; r_o \downarrow; [\neg r_i]; l_o \uparrow; [\neg l_i]; l_o \downarrow]
\]

- ";" = sequencing.

- \( r_0 \uparrow = r_0 \) goes high, \( r_0 \downarrow r_0 \) goes low.

- \([l_i] = \) wait until \( l_i \) is high, \([\neg l_i] \) wait until \( l_i \) is low.

- Program meaning: \( l_i \) has to be high before \( r_0 \) goes high. The circuit will wait until \( r_i \) goes high, before \( r_0 \) will go low again and so on ....
Production rules and CSP programs

**CSP program**

\([l_i]; r_o \uparrow; [r_i]; r_o \downarrow; [\neg r_i]; l_o \uparrow; [\neg l_i]; l_o \downarrow\)

**Production rules**

\(l_i \leftrightarrow r_o \uparrow\)
\(r_i \leftrightarrow r_o \downarrow\)
\(\neg r_i \leftrightarrow l_o \uparrow\)
\(\neg l_i \leftrightarrow l_o \downarrow\)

- ";" = sequencing.
- \([l_i]\) = wait
- production rule = boolean condition \(\Leftarrow\) transition.
- \(l_i \leftrightarrow r_o \uparrow\): when \(l_i\) is high, \(r_o\) will go high.
- \(\neg r_i \leftrightarrow l_o \uparrow\): when \(r_i\) is low, \(l_o\) will go high.
- conflict: \(r_o \uparrow\) and \(r_o \downarrow\) are not mutually exclusive.
Elimination of conflict

- Add a state signal to solve conflict:
  \[*[[l_i]; r_o \uparrow; [r_i]; x \uparrow; [x]; r_o \downarrow; [\neg r_i]; l_o \uparrow; [\neg l_i]; x \downarrow; [\neg x]; l_o \downarrow]*\]

- The new production rules:
  \[
  \neg x \land l_i \leftrightarrow r_o \uparrow \\
  r_i \leftrightarrow x \uparrow \\
  x \leftrightarrow r_o \downarrow \\
  x \land \neg r_i \leftrightarrow l_o \uparrow \\
  \neg l_i \leftrightarrow x \downarrow \\
  \neg x \leftrightarrow l_o \downarrow 
  \]

Summer school. Lyngby 1997

Synthesis of control circuits from STGs 182
The production rules can now be identified with operations.

- \( r_0 \) maps to an and-gate \( (r_0 = \overline{x} l_i) \).

\[
\begin{align*}
\neg x \land l_i & \mapsto r_o \uparrow \\
\neg x & \mapsto r_o \downarrow
\end{align*}
\]
FSM

s1

\overline{x}C + \overline{x}C/o

s2

\overline{C} + xC/o

s3

\overline{x}C/o

y1y2

00

xC/o

10

\overline{x}C/o

11

\overline{x}C/o
FSM(2)

- Nodes = states, arcs = state transition
- Each arc is labeled with a pair of input/output symbols.
  - *input* = condition for state transition to take place.
  - *output* = values for output signals after the state transition.
- Mostly limited to fundamental mode [Unger].
BURST MODE FSM

- Input transitions fire.
- Output transitions fire.
- State signals change.

Diagram:

- States: s1, s2, s3, s4
- Transitions:
  - s1 to s2: a+/y+
  - s1 to s4: a-b-/x-
  - s2 to s3: b+/x+y-
  - s2 to s4: c-/y+
  - s3 to s4: c+/y-
Main features of burst mode FSM

- Mostly limited to fundamental mode [Unger]
- Constrained state minimization [Nowick, Yun]
- Exact minimization for hazard-free logic [Nowick]
- Standard tools can be used under the new constraints
Extended Burst Mode FSM

- Directed don’t cares \((a*)\): some concurrency between inputs and outputs is allowed for input transitions which do not influence an output burst.

- Conditionals: control flow depends on the levels (not the edges) of conditional signals.

- Every XBM can be represented by an equivalent STG.

- See [Yun94] for details.
XBM and STG. Example

$D^*$ is a directed don’t care (only monotonous changes are allowed)
$<D->$ is a conditional guard = "if $D=0$ then ...."
Restrictions on XBM specification

1. Conditional signals cannot serve as edge signals

2. Compulsory edge transition must be present at every input burst
   Compulsory = must appear during the state transition it labels, not before.

3. Setup/hold constraints for changes of conditional signals:
   stay stable during \([First\_edge - setup, Last\_edge + hold]\)

4. Input bursts must be distinguishable

5. Each state satisfies unique entry point condition
Distinguishability of input bursts

Distinguishability = one burst is not a subset of another (including conditionals)
Unique entry condition

The set of input and output values is the same for any input arc of a state.

This condition can always be satisfied by splitting states.
Three types of cycles:
- input burst ⇒
  output burst || state burst
- input burst ⇒
  output burst ⇒ state burst
- input burst ⇒
  state burst ⇒ output burst
- choice depends on type of combinational logic used (to avoid hazards) and on the required level of concurrency
Reduction of XBM to combinational logic

- Specify logic functions free of *function* hazards
- Implement logic functions free of *logic* hazards
- Ensure that sequential feedbacks do not introduce new hazards
Sequential hazards in XBM

- **Constraints on the environment:**
  - Fundamental mode constraints: no compulsory transitions of the next input burst may arrive before the machine is stabilized
  - Halt/setup constraints for conditional signals

- **Constraints on feedbacks** = Fundamental mode for feedbacks:
  - To avoid essential hazards (recall the Mod2-counter example) check that feedbacks are slow enough. If not, insert delays.
  - If the structure of the combinational logic is fixed (2-level SOP), then use sufficient constraints like, e.g.:
    \[ t_{\text{min, comb.logic}} + t_{\text{min, feedback}} > t_{\text{max, inverter + input wire}} \]

- **Constraints on state encoding:** use race-free encoding for state signals
Function hazards in XBM

- Any single burst of transitions at the input of combinational logic is function hazard-free by construction of the next state table.

- Assigning the next state values for different bursts of transitions can produce conflicts in the next state table = CSC problem for XBM.

- To solve the conflicts use additional state signals and the race-free encoding.
### Logic hazards in XBM. Sum-of-products logic

#### Logical Functions

- **Static hazards elimination condition:**
  
  *Each transition cube is contained inside a cube of the cover*

- **Dynamic 1to0-hazard (0to1) elimination condition:**
  
  *If transition cube intersects cube c of the cover, then c contains the start (end) cube of the transition cube*

- **Conflicting requirements can be resolved by adding more state signals**

---

<table>
<thead>
<tr>
<th></th>
<th>ab</th>
<th>cd</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
 f = ac' + cb' + ab' \\
 f = ac' + cb' \\
 f = ac' + cb' + ab' \\
\]
$x = \overline{abq} + a\overline{c} + a\overline{b}$. Signal $a$ is a trigger for $x$, signal $b$ - is not

- Trigger signals must precede non-trigger signals in variable ordering for BDD: $a < b$ is valid, $b < a$ – is not.
- Conflicting requirements can be resolved by adding more state signals
3D synthesis procedure

1. Next state assignment: one state of the specification to one layer of the Karnaugh map

2. Layer minimization: compatible layers are merged

3. Layer encoding: use adjustments of standard race-free encoding (Tracey)

4. Synthesis of hazard-free combinational logic (as described above)
State assignment resolves conflicts in definition of the next state function similar to CSC-conflicts in STG synthesis.
Layer minimization

- Standard state minimization for \textit{incompletely} specified FSMs

- Compatibility of states (layers) defined differently

- State \( i \) and \( j \) are not compatible \( \equiv \) cannot be merged in one layer without state conflicts (CSC) or violating requirements for hazard-freedom of logic

- Compatibility relation depends on the implementation of the combinational logic (SOP, BDD-based)
State compatibility

\[ ab \cdot x = 01.1 \]

\[ a + b - / x - \]

\[ b + / x + \]

\[ \text{merged in one layer} \rightarrow \text{hazards} \]

\[ \text{cube required for state 1} \]
\[ \text{incorrectly intersects} \]
\[ \text{transition cube for state 0} \]
\[ \text{(the start point is not covered)} \]