Embedded Systems 26

Testing: Scope

Testing includes

- the application of test patterns to the inputs of the device under test (DUT) and
- the observation of the results.

More precisely, testing requires the following steps:

1. test pattern generation,
2. test pattern application,
3. response observation, and
4. result comparison (okay, not okay, *inconclusive*).

Test pattern generation

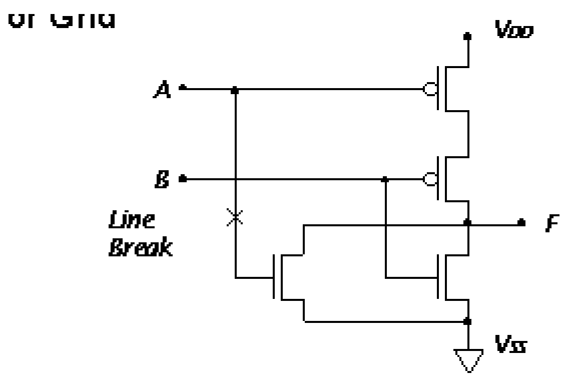
Test pattern generation typically

- considers certain fault models and
- generates patterns that enable a distinction between the faulty and the fault-free case.
- Coverage criteria shed light on the likeliness of instances of the fault type slipping through

Hardware Fault models

Hardware fault models include:

- stuck-at fault model (net permanently connected to ground or V_{dd})
- stuck-open faults: for CMOS, open transistors can behave like memories
- delay faults: circuit is functionally correct, but the delay is not.

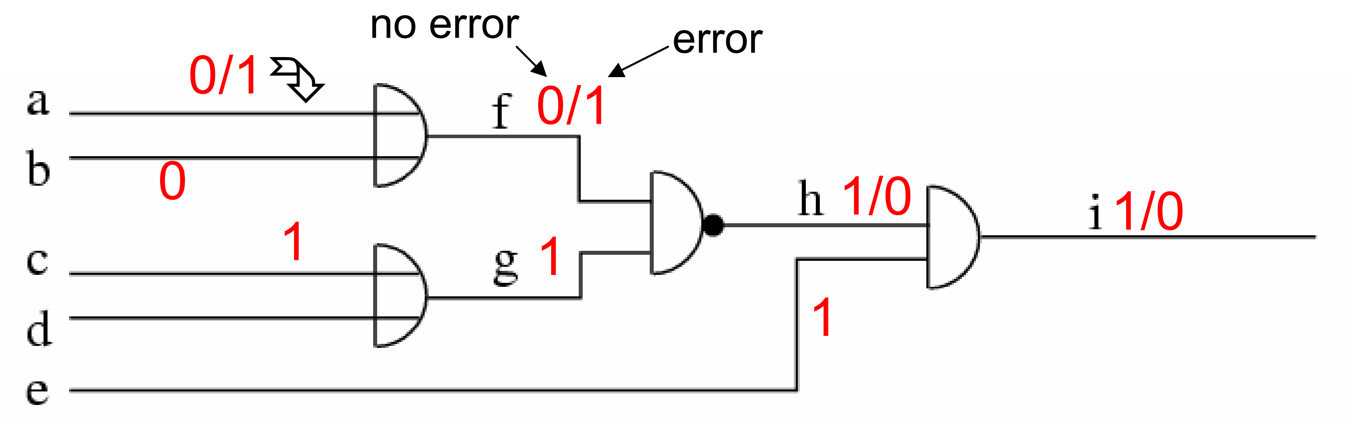


www.cedcc.psu.edu/ee497f

/rassp_43/sld022.htm

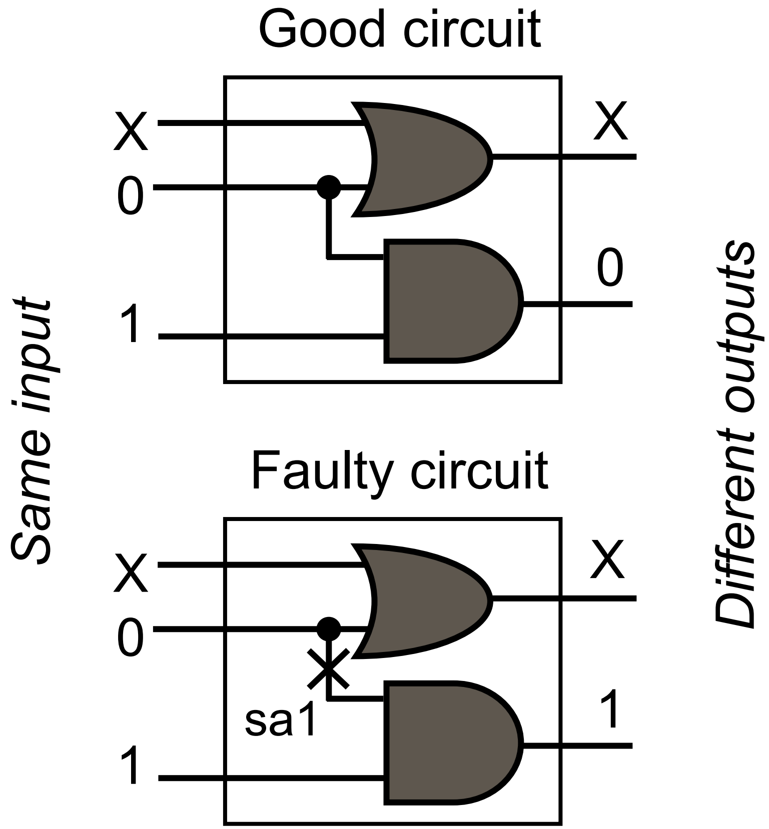
- *Break above results in a "memory-effect" in the behavior of the circuit*
- *With AB=10, there is not path from either VDD or VSS to the output*
  - *F retains the previous value for some undetermined discharge time*

Simple example



- Could we check for a stuck at one error at a (s-a-1(a)) ?
- Solution (just guessing):
  - f='1' if there is an error
  - ⇒ a='0', b='0' in order to have f='0' if there is no error
  - g='1' in order to propagate error
  - c='1' in order to have g='1' (or set d='1')
  - e='1' in order to propagate error
  - i='1' if there is no error & i='0' if there is

Need to Deal With Two Copies of the Circuit



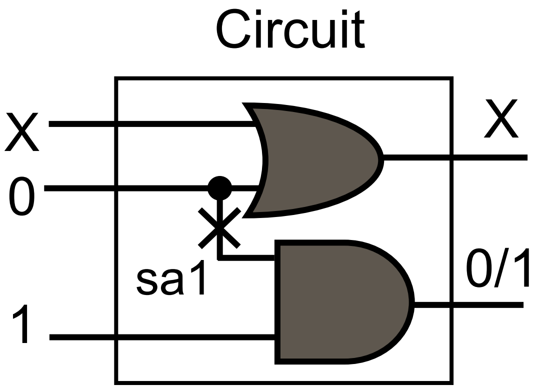
Good circuit

Faulty circuit

Same input

Different outputs

Alternatively, use a multi-valued algebra of signal values for both good and faulty circuits.



Circuit

Roth‘s 5-valued algebra (1966)

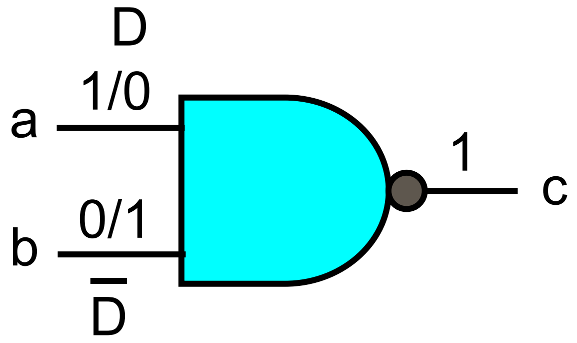
Symbol	Alternative Representation	Fault-free circuit	Faulty Circuit
D	1/0	1	0
$\overline{D}$	0/1	0	1
0	0/0	0	0
1	1/1	1	1
X	X/X	X	X

Pseudo - value :

$$D : \begin{cases} 1 & \text{if there is no error} \\ 0 & \text{if there is an error} \end{cases}$$

$$\overline{D} : \begin{cases} 0 \\ 1 \end{cases}$$

Function of NAND Gate



c		Input a				
		0	1	X	D	$\overline{D}$
Input b	0	1	1	1	1	1
	1	1	0	X	$\overline{D}$	D
	X	1	X	X	X	X
	D	1	$\overline{D}$	X	$\overline{D}$	1
	$\overline{D}$	1	D	X	1	D

D-Algorithm

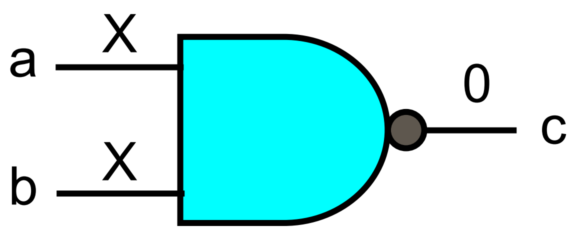
- Use D-algebra
- Activate fault
    - Place a D or $\overline{D}$ at fault site
    - Do justification, forward implication and consistency check for all signals
- Repeatedly propagate D-chain toward POs through a gate
    - Do justification, forward implication and consistency check for all signals
- Backtrack if
    - A conflict occurs, or
    - D-frontier becomes empty
- Stop when
    - D or $\overline{D}$ at a PO, i.e., test found, or
    - If search exhausted without a test, then no test possible

Definitions

- **Justification:** Changing inputs of a gate if the present input values do not justify the output value.
- **Forward implication:** Determination of the gate output value, which is X, according to the input values.
- **Consistency check:** Verifying that the gate output is justifiable from the values of inputs, which may have changed since the output was determined.
- **D-frontier:** Set of gates whose inputs have a D or $\overline{D}$, and the output is X.

Definition: Singular Cover

- A singular cover defines the least restrictive inputs for a deterministic output value.
- Used for:
  - Line justification: determine gate inputs for specified output.
  - Forward implication: determine gate output.



Examples: XX0 ∩ 110 = 110
0XX ∩ 0X1 = 0X1

Singular covers	a	b	c
SC-1	0	X	1
SC-2	X	0	1
SC-3	1	1	0

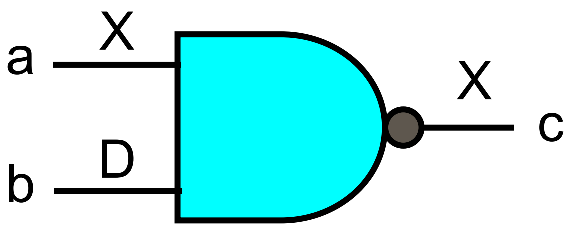
D-Intersection

∩	0	1	X	D	$\overline{D}$
0	0	■	0	■	■
1	■	1	1	■	■
X	0	1	X	D	$\overline{D}$
D	■	■	D	D	■
$\overline{D}$	■	■	$\overline{D}$	■	$\overline{D}$

Undefined State (conflict)

Definition: D-Cubes

- D-cubes are singular covers with five-valued signals
- Used for D-drive (propagation of D through gates) and forward implication.



Examples:

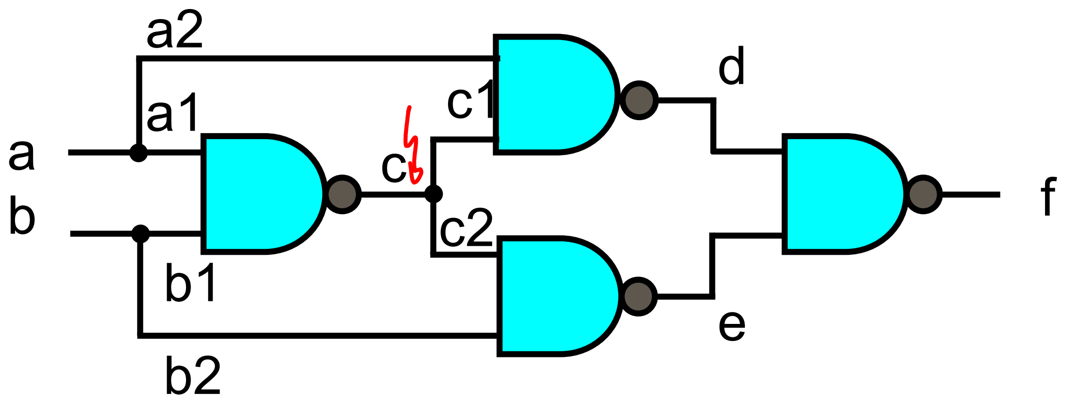
$XDX \cap 1D\overline{D} = 1D\overline{D}$

$0DX \cap 0D1 = 0D1$

$D\overline{D}X \cap D\overline{D}1 = D\overline{D}1$

D-cube	a	b	c
D-1	D	1	$\overline{D}$
D-2	1	D	$\overline{D}$
D-3	$\overline{D}$	1	D
D-4	1	$\overline{D}$	D
D-5	D	D	$\overline{D}$
D-6	$\overline{D}$	$\overline{D}$	D
D-7	D	0	1
D-8	0	D	1
D-9	D	$\overline{D}$	1
D-10	$\overline{D}$	D	1

Example: Test for c sa0



① Activate fault: set $c_1, c_2 = D$; $c = 1$

② Justify $c = 1$: $XX1 \cap 0X1 = 0X1$
$a = a1 = a2 = 0$

③ Forward implication $a2 = 0$: $0DX \cap 0D1 = 0D1$
$d = 1$

④ Forward implication $d = 1$: $1XX \cap XXX = 1XX$
→ no implication possible

⑤ D-drive $c2 \to e$: $DXX \cap D1\overline{D} = D1\overline{D}$
$b2 = b = b1 = 1$, $e = \overline{D}$

D-frontier
d, e
d, e
e
e
f

⑥ Forward impl. $u\ 1 = 1$ f

$011 \cap 0X1 = 011$

consistency checked.

⑦ D-drive $e \rightarrow f$ $1\bar{D}X \cap 1\bar{D}D$ PO

$f = D$

$\Rightarrow$ STOP, test found:

$$\boxed{\text{Test } (a,b) = (0,1),\ f = 1}$$

Complexity of D-Algorithm

- Signal values on all lines (PIs and internal lines) are manipulated using 5-valued algebra.
- Worst-case combinations of signals that may be tried is $5^{\#\text{lines}}$
  - For XOR circuit, 5^{12} = 244,140,625.
- Podem: A reduced-complexity ATPG algorithm
  - Recognizes that internal signals depend on PIs.
  - Only PIs are independent variables and should be manipulated.
  - Because faults are internal, a PI can assume only 3 values (0, 1, X).
  - Worst-case combinations = $3^{\#\text{PI}}$; for XOR circuit, 3^2 = 8.

Fault coverage

A certain set of test patterns will not always detect all faults that are possible within a fault model

$$coverage = \frac{\text{Number of detectable faults for a given test pattern set}}{\text{Number of faults possible due to the fault model}}$$

For actual designs, the coverage should be at least in the order of 98 to 99%

Fault simulation

- Coverage can be computed with **fault simulation**:
- ∀ faults ∈ fault model: check if distinction between faulty and the fault-free case can be made:

```
Simulate fault-free system;
 ∀ faults ∈ fault model  DO
  ∀ test patterns DO
   Simulate faulty system;
   Can the fault be observed for ≥1 pattern?
```

  Faults are called **redundant** if they do not affect the observable behavior of the system,

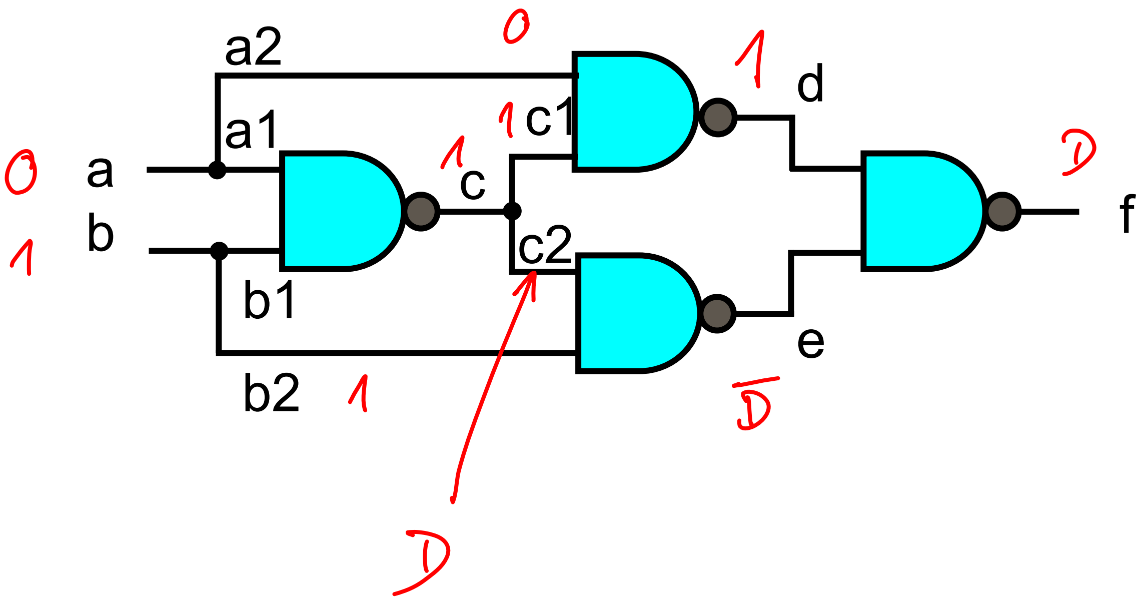
- Fault simulation checks whether mechanisms for improving fault tolerance actually help.

Finding Other Detected Faults by the Generated Test

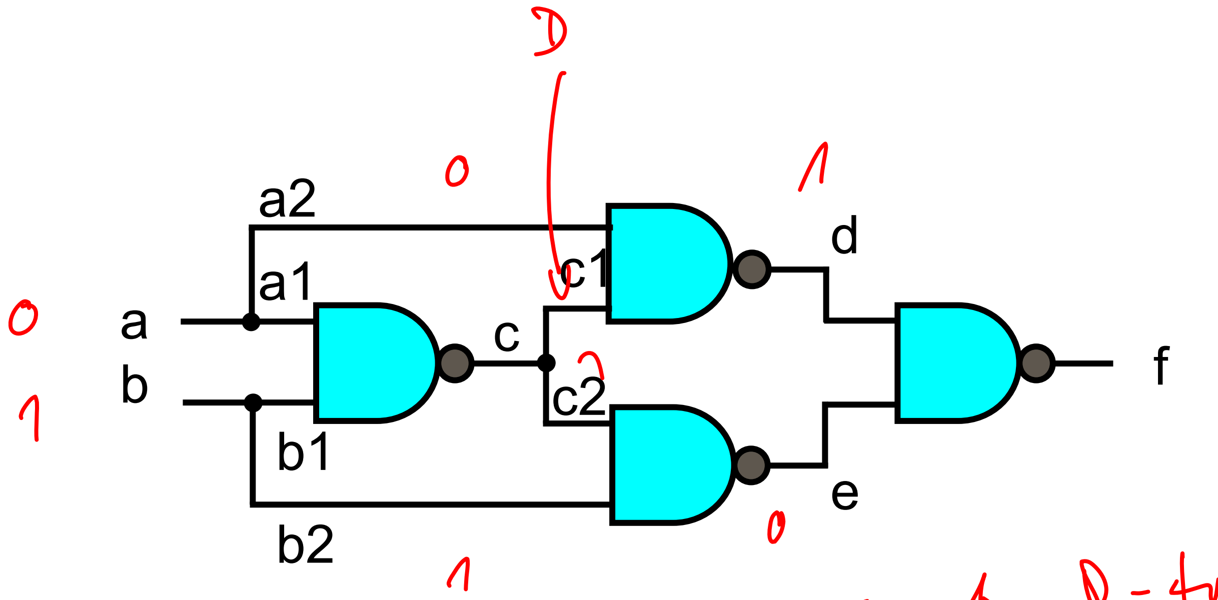
- Determine good circuit signal values.

- For each fault
  - Place a D or $\overline{D}$ at the fault site
  - Perform forward implications
  - Fault is detected if any PO assumes a D or $\overline{D}$ value

Example: Detect c2 sa0 with Test(0,1)?



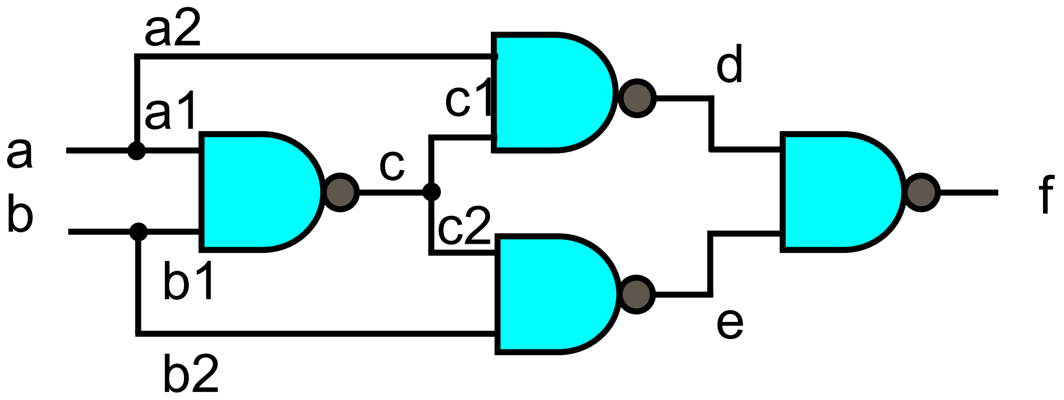
Example: Detect c1 sa0 with Test(0,1)?



empty D-frontier!

⇒ c1 sa 0 is not detected!

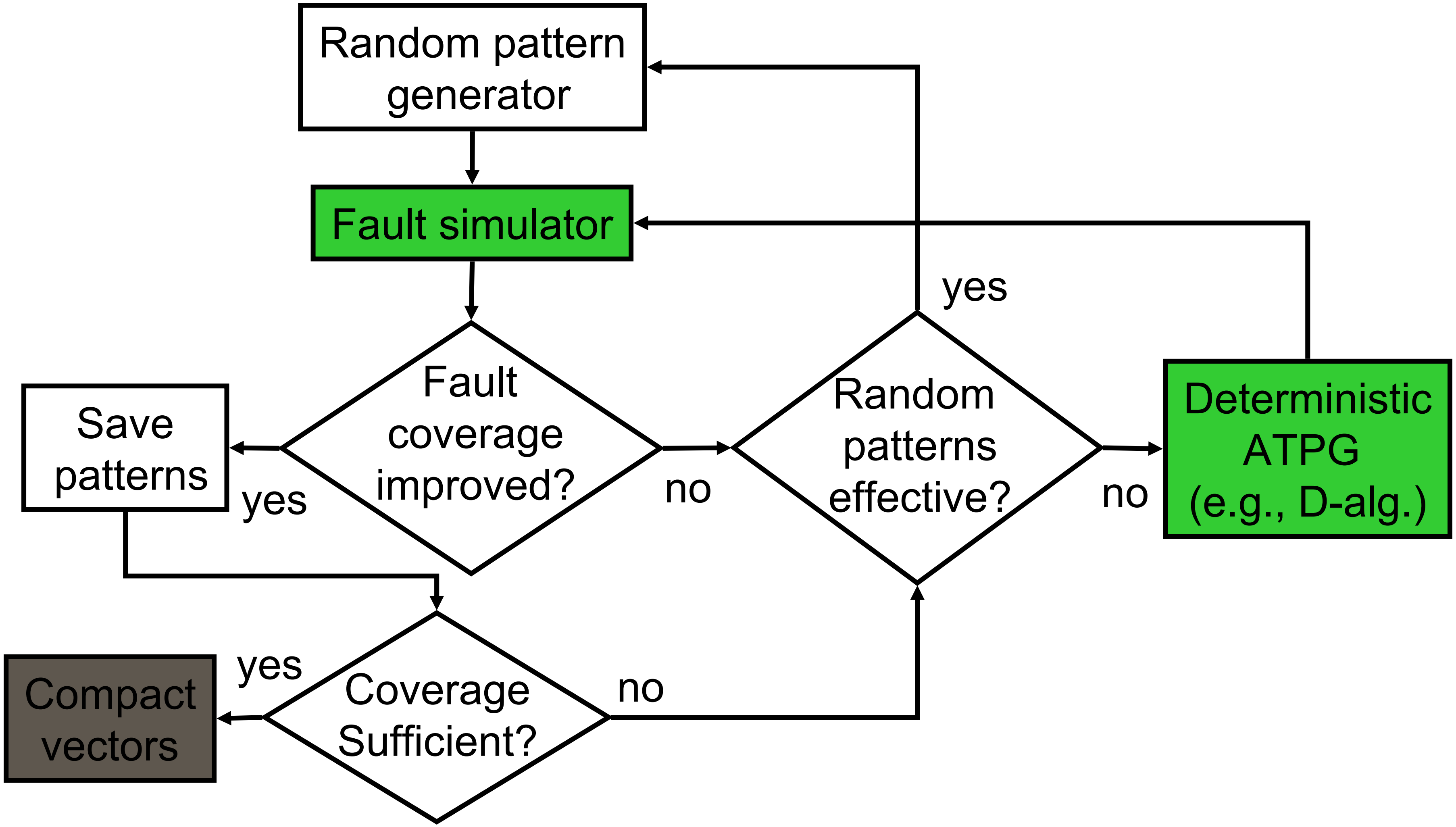
Test for c1 sa0



↗ Exercise!

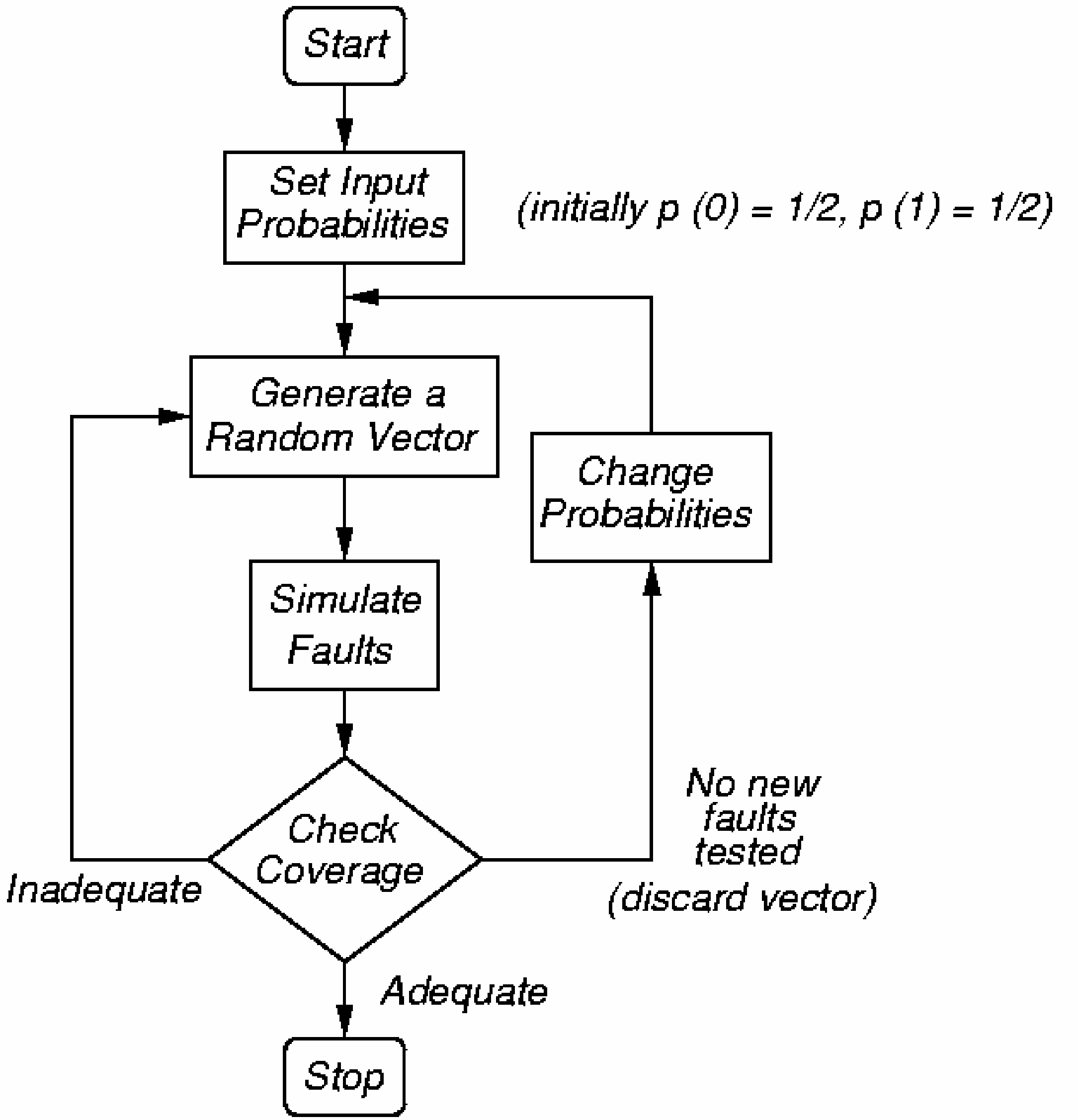
→ Test: $(a,b) = (1,0)$, $f = 1$

An ATPG System



Random Pattern Generation

- Typically gets tests for 60-80% of faults
- Then switch to D-algorithm or other ATPG method



Vector Compaction

- **Objective: Reduce the size of test vector set without reducing fault coverage.**
- Simulate faults with test vectors in reverse order of generation
  - ATPG patterns go first
  - Randomly-generated patterns go last (because they may have less coverage)
  - When coverage reaches 100% (or the original maximum value), drop remaining patterns
- Significantly shortens test sequence ⇒ testing cost reduction**.**

Static and Dynamic Compaction of Sequences

- **Static compaction**
  - ATPG should leave unassigned inputs as X
  - Two patterns *compatible* – if no conflicting values for any PI
  - Combine two tests t_a and t_b into one test $t_{ab} = t_a \cap t_b$ using intersection
  - Detects union of faults detected by t_a and t_b
- **Dynamic compaction**
  - Process every partially-done ATPG vector immediately
  - Assign 0 or 1 to PIs to test additional faults

Example

$t_1 = 0\ 1\ X$

$t_3 = 0\ X\ 0$

$t_2 = 0\ X\ 1$

$t_4 = X\ 0\ 1$

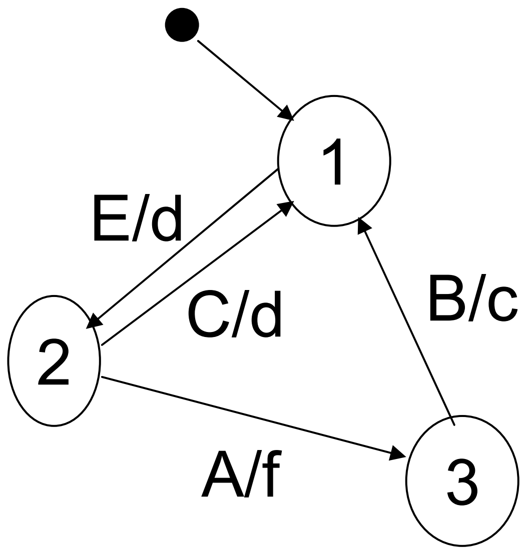
$t_{13} = 0\ 1\ 0$

$t_{24} = 0\ 0\ 1$

Design for testability

Testing finite state machines

Difficult to check states and transitions.

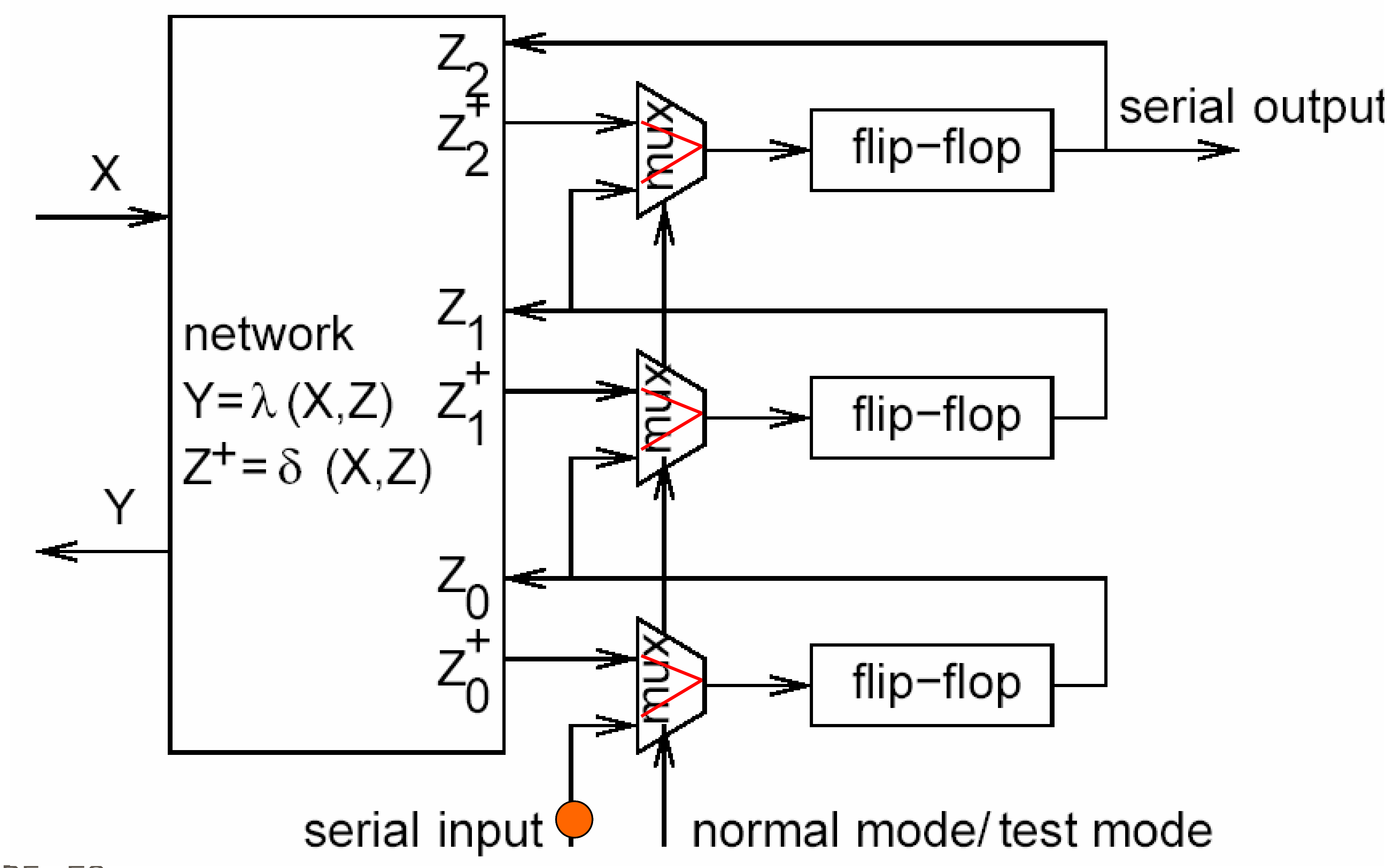


For example, verifying the transition from state 2 to 3 requires

- Getting into state 2
- Application of A
- Check if output is f
- Check if we have actually reached 3

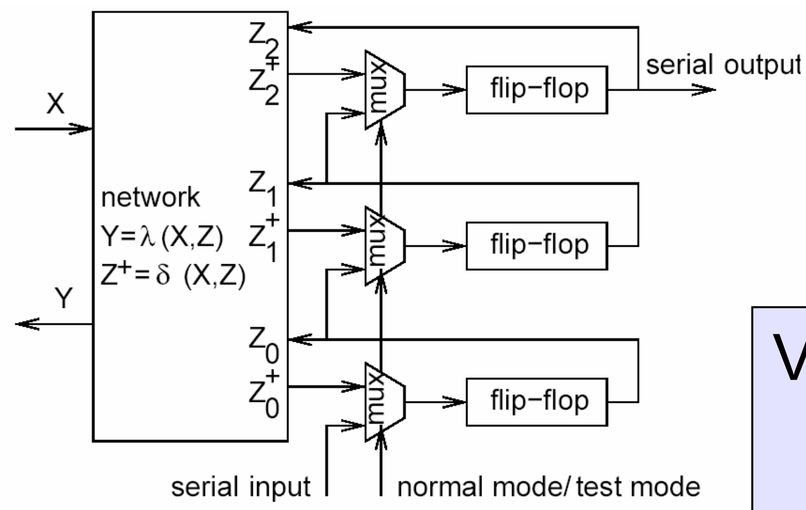
Can be simplified by “design for testability” → Scan design

Scan design



X
Y
network
Y=λ (X,Z)
Z⁺=δ (X,Z)
Z2
Z2+
Z1
Z1+
Z0
Z0+
mux
mux
mux
flip−flop
flip−flop
flip−flop
serial output
serial input
normal mode/ test mode

Scan design: usage



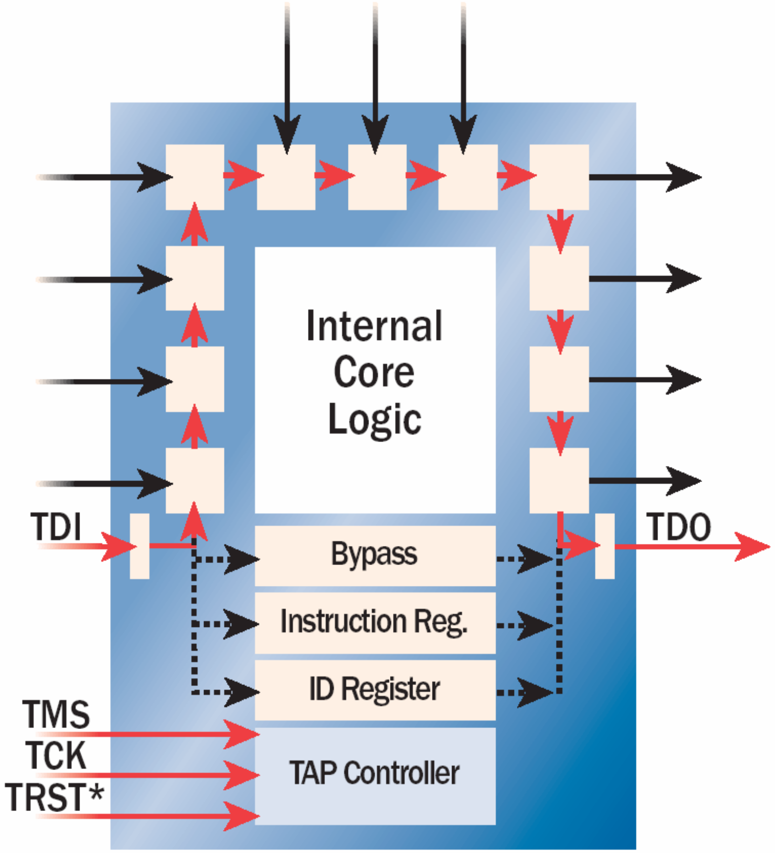
Verifying a transition requires

- Shifting-in the state to be tested
- Application of the input pattern
- Checking if output is correct
- Shifting-out the successor state and comparing it.

Essentially reduced to testing combinatorial logic

JTAG (Boundary scan)

- JTAG / IEEE 1149.1 defines a 4-5 wire serial interface to access complex ICs. Any compatible IC contains shift registers + *an FSM* to execute the JTAG functions.
- **TDI:** test data in; goes into instruction register or into one of the data registers.
  TDO: test data out
  TCK: clock
  TMS: controls the state of the test access port (TAP).
  Optional **TRST*** is reset signal.



IEEE 1149.1 Device Architecture

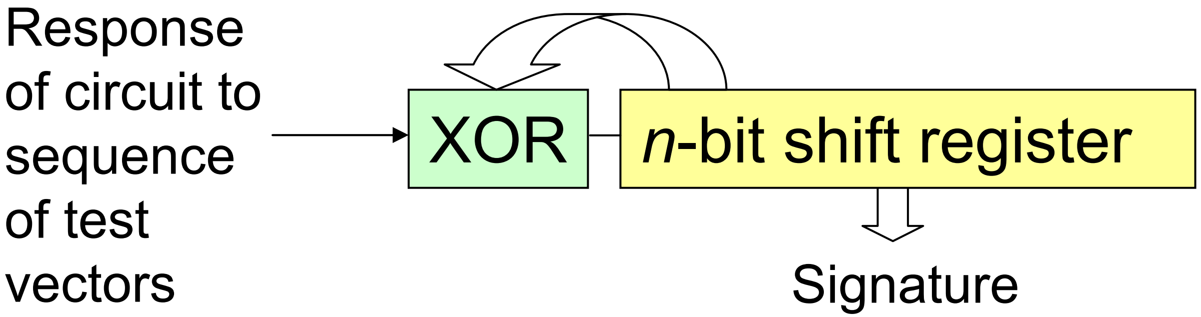
Source: http://www.jtag.com/brochure.php

Limitations of a single serial scan chain

- For chips with a large number of flip-flops, serial shifts can take a quite long time.
- Hence, it becomes necessary to provide several scan chains.
  - ☞ Trying to avoid serial shifts by generating test patterns internally and by also storing the results internally.
  - ☞ Compaction of circuit response in a **signature**. Shifting the entire result out becomes obsolete, we just shift out the signature.

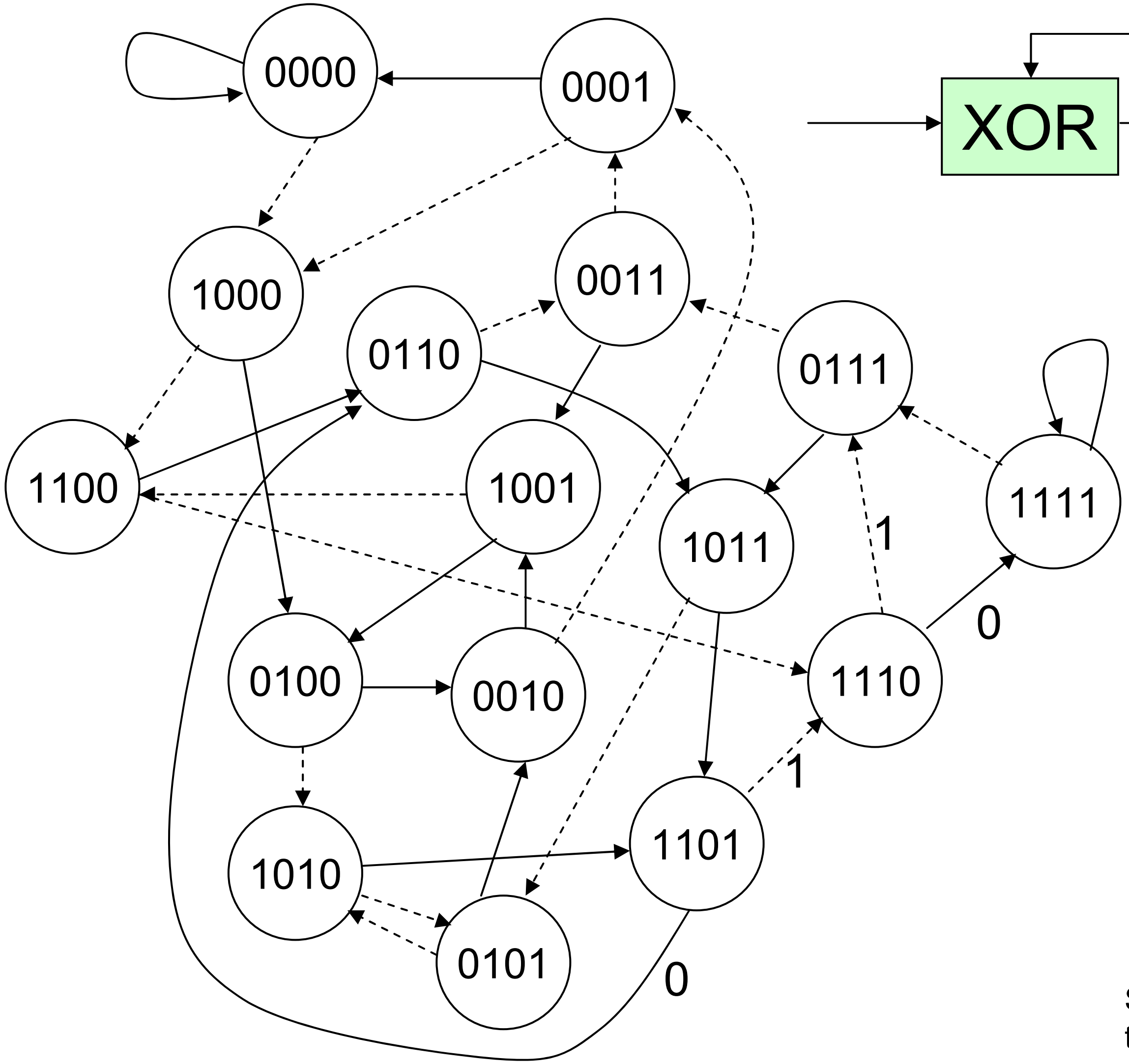
Signature analysis

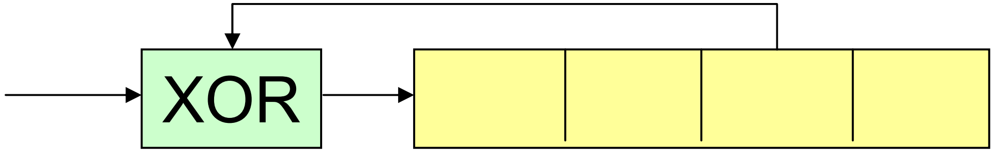
- Response of circuit to sequence of test patterns compacted in a signature. Only this signature is compared to the golden reference.
- In order to exploit an *n*-bit signature register as good as possible, we try to use all possible values.
- In practice, we use shift-registers with linear feedback:



- Using proper feedback bits, all possible values for the register can be generated.

Example: 4-bit signature generator





All 16 possible signatures are generated!

Source: P.K.Lala: Fault tolerant & fault testable hardware design, Prentice Hall, 1985

Aliasing for signatures

Consider aliasing for some current pattern

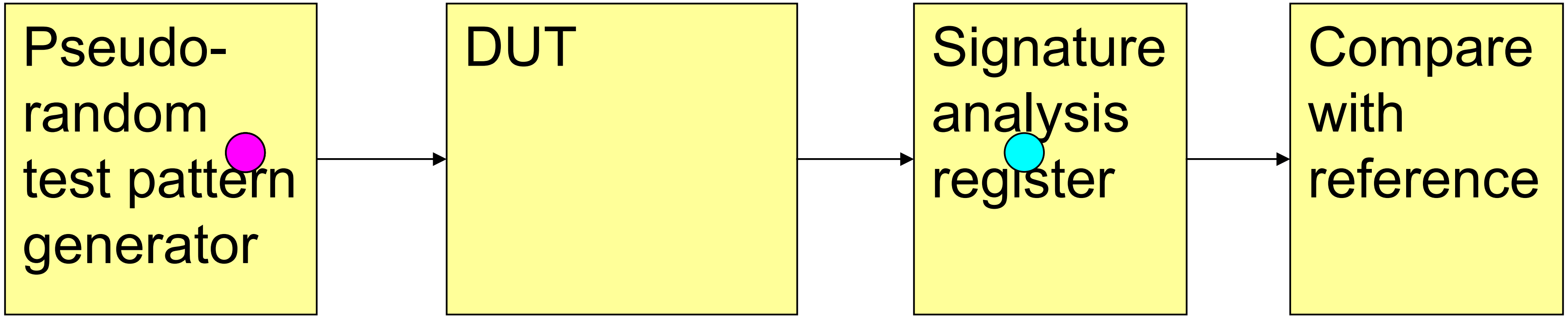
- An *n*-bit signature generator can generate 2^n signatures.
- For an *m*-bit input sequence, the best that we can get is to evenly map $2^{(m-n)}$ patterns to the same signature.
- Hence, there are $2^{(m-n)}$-1 sequences that map to the same signature as the pattern currently considered.
- In total, there are 2^m-1 sequences different from the current one.

☞ $$P = \text{Probability}\left(\frac{\text{other patterns map to same signature}}{\text{total number of other patterns}}\right) = \frac{2^{(m-n)} - 1}{2^m - 1}$$

$$P = \frac{1}{2^n} \text{ for } m >> n \text{ provided that we evenly map patterns to signatures}$$

Replacing serially shifted test pattern by pseudo-random test patterns

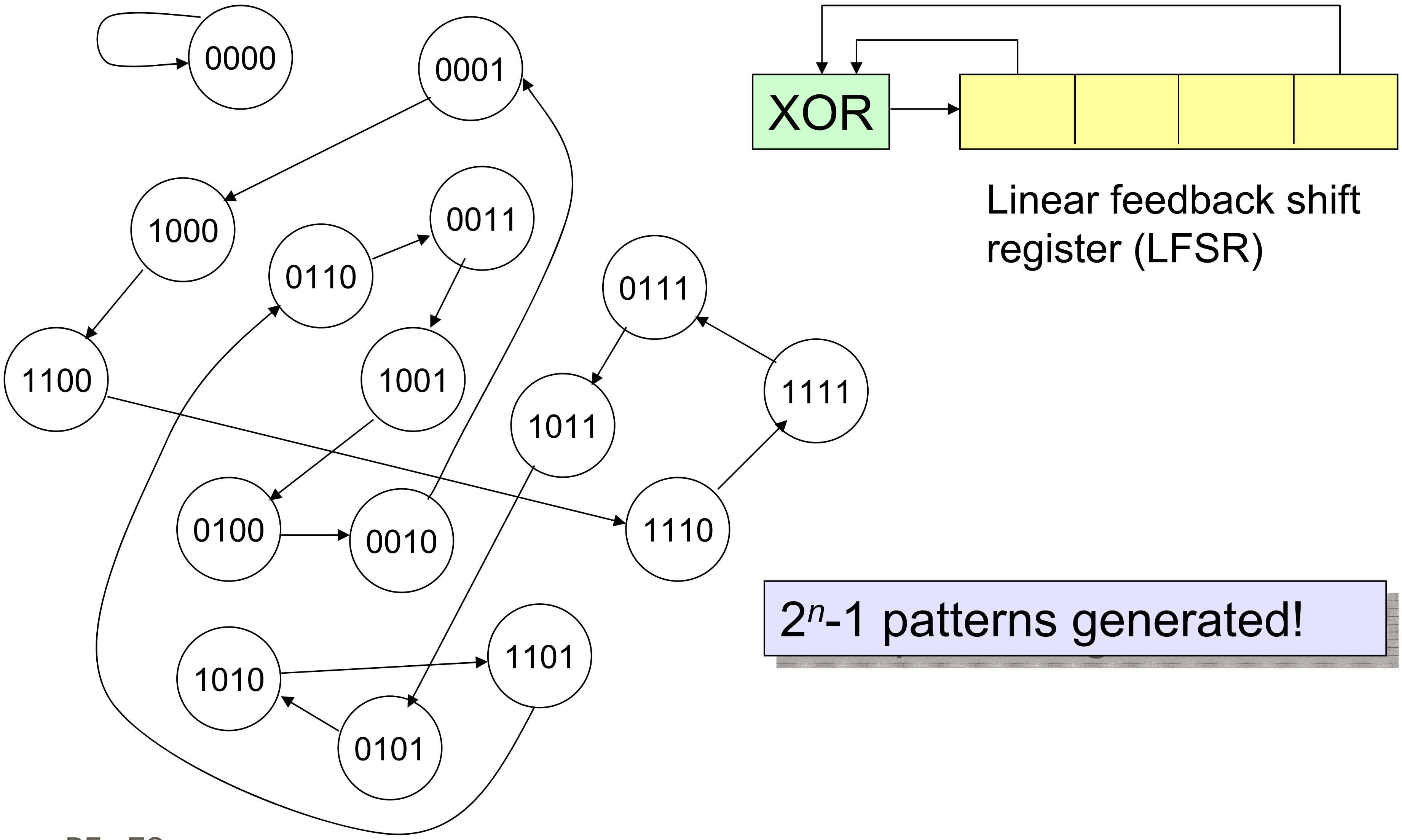
Shifting in test patterns can be avoided if we generate (more or less) all possible test patterns internally with a pseudo-random test pattern generator.


Pseudo-random test pattern generator
DUT
Signature analysis register
Compare with reference

- Effect of pseudo random numbers on coverage to be analyzed.
- Signature analysis register shifted-out at the end of the test.

Comparison possible because test pattern generator is a deterministic source!

Pseudo random test pattern generation

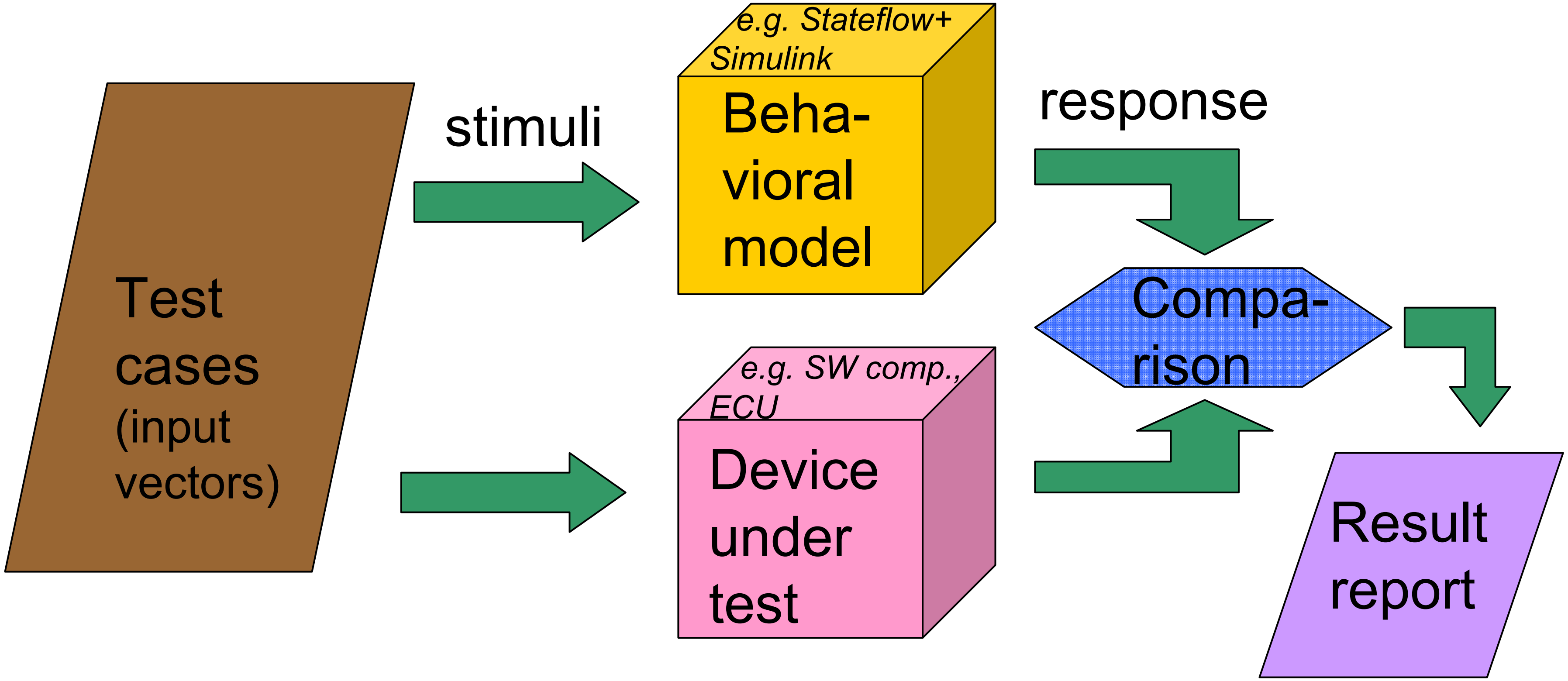


Linear feedback shift register (LFSR)

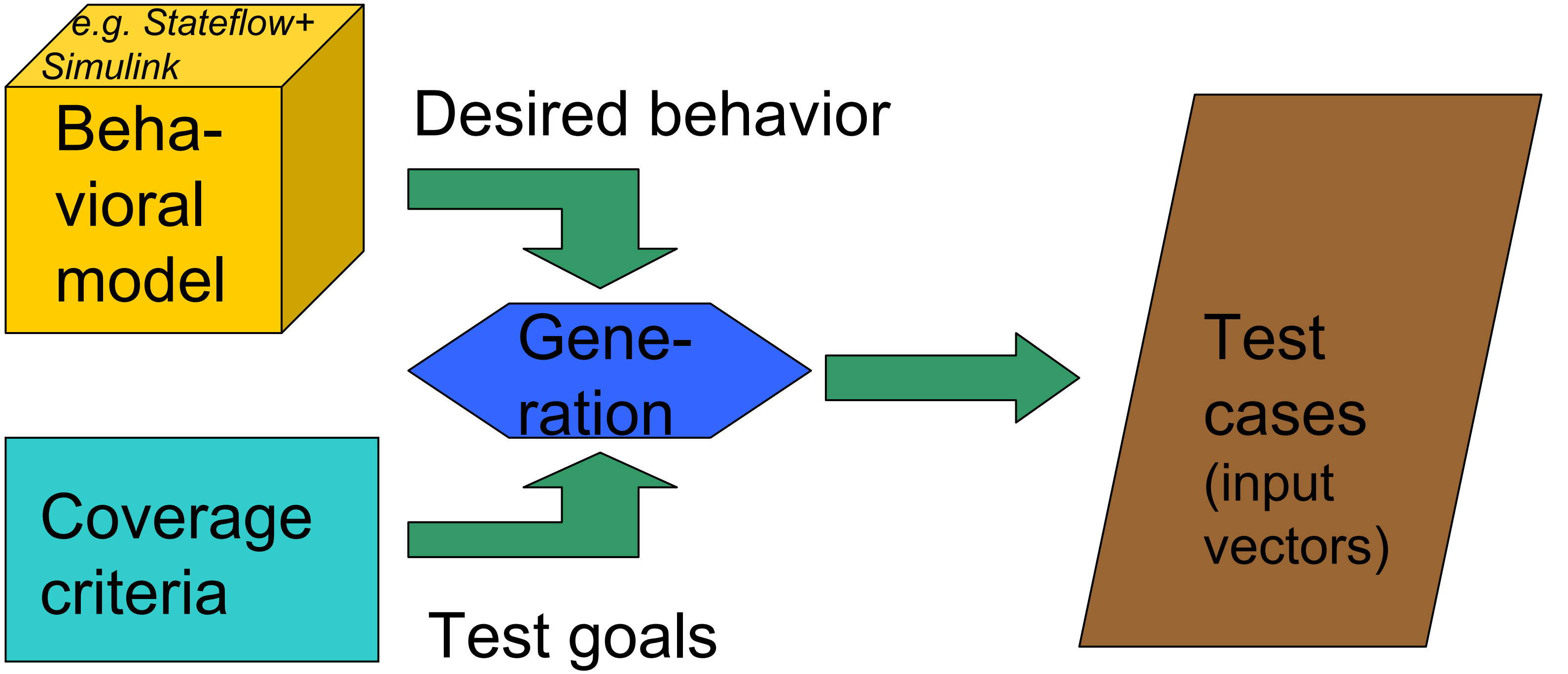
2^n-1 patterns generated!

Model-based testing

Model-based testing: model as “golden device”

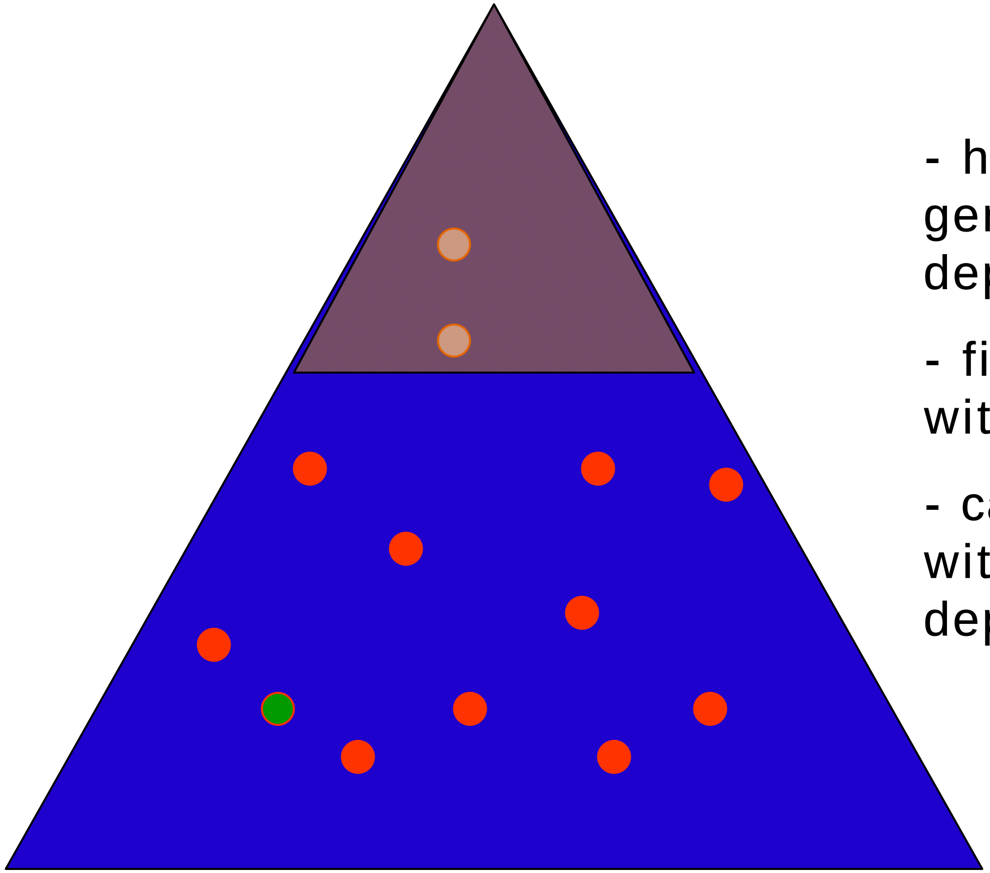


Model-based testing:
model-based test vector generation



Main operation is search of computation paths in the model which lead to states with certain properties.

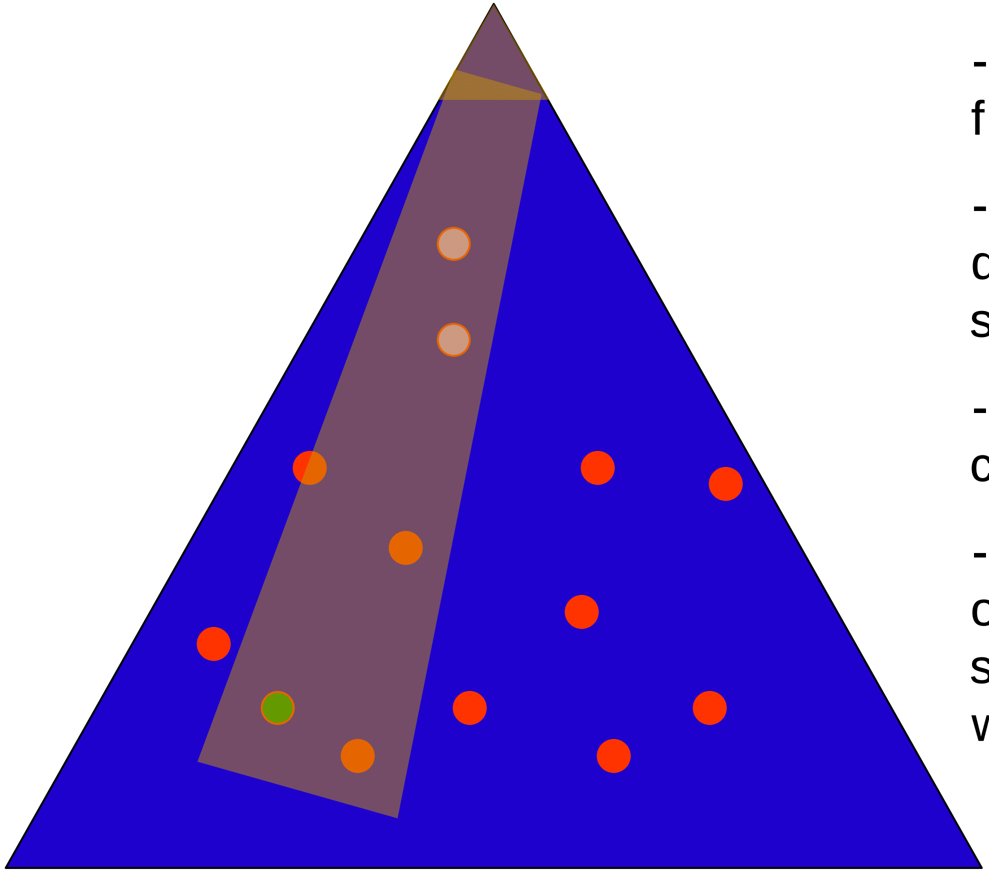
Breadth-first search (BFS)



- hard depth limit (#states generated exponential in depth)
- finds all fulfilling states within depth limit
- cannot find fulfilling states with path length longer than depth limit

(Copyright this and next slide: M. Lettrari, MbEES 05/06)

Heuristic search



- uses a heuristic function h for computing value h(s)

- h(s) approximates the real distance from s to a p-fulfilling state s‘

- Heuristic search can be combined with BFS

- if h is a good approximation of real distance, heuristic search can find fulfilling states with very long path length

Fault Injection

Fault injection

- Fault simulation may be too time-consuming

☞ If real systems are available, faults can be injected.

☞ Two types of fault injection:

1. local faults within the system, and
2. faults in the environment (behaviors which do not correspond to the specification).
   For example, we can check how the system behaves if it is operated outside the specified temperature or radiation ranges.

Fault injection

- Intentional activation of faults by HW or/and SW means
  - Establish faults in a predictable and reproducible way
  - Trigger error-handling routines
- Two purposes:
  - Testing and Debugging
    - During normal operation faults are *rare events*
    - May be much too rare to achieve meaningful data from std. testing
  - Dependability Forecasting
    - Used for deriving data about the likely dependability of the system
    - Need to know the types and frequencies of different faults in the intended operational environment

Software Fault Injection

- **Errors are seeded into memory by software**
  - Mimic errors originating in hardware faults by software
  - Either randomly or in specific location to provoke specific fault-management routine

- **Advantages over physical fault injection** include
  - Predictability and reproducibility: fault injection is independent of uncontrollable or statistical effects
  - Reachability of inner registers in VLSI chips
  - Simplicity of experiments: can be carried out with software tools

- **Coverage similar to physical fault injection**, if well-done
  - Statistical data gathered from physical injection experiments can be used to trim software fault injection

Typical forms of SW fault injection

- Random bit flips in memory
  - Simulates adverse operational conditions corrupting memory
- Boolean masking of words written to (some) memory
- Discard some message(s)
  - Simulates imperfect communication media
- Adding messages
  - Simulates presence of a babbling idiot
  - Checks consequences of certain fault tolerance mechanisms (resend…) if used when transient fault condition was no longer present (very hard to test by HW fault injection)
- Delaying messages / result delivery
- …and many more