ATPG Algorithms

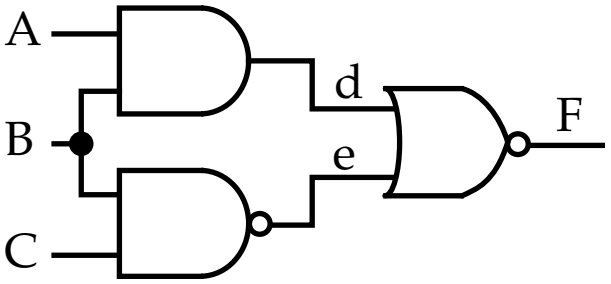
Characteristics of the three main algorithms:

- Roth's D-Algorithm (D-ALG) defined the calculus and algorithms for ATPG using D-cubes.
- Goel's PODEM used path propagation constraints to limit the ATPG search space and introduced *backtrace*.
- Fujiwara's FAN efficiently constrained the backtrace to speed up search and further limited the search space.

D-Calculus and D-Algorithm

Definitions:

- *Singular cover*: Defined to be the minimal set of input signal assignments needed to represent **essential prime implicants** in Karnaugh map.



AND	a	b	d	NOR	d	e	F
1	0	X	0	*4*	1	X	0
2	X	0	0	*5*	X	1	0
3	1	1	1	*6*	0	0	1

D-Calculus and D-Algorithm

- *D-cube*: A collapsed truth table entry.

  For example, combine rows 3 and 1 of the AND gate singular cover, and express it in Roth's 5-valued algebra (row 3 is good machine).

  $D\ 1\ D$

  Rows 3 and 2 yield the *propagation D-cube*:

  $1\ D\ D$

  A third is $D\ D\ D$.

  Inverting D to $\overline{D}$ in each of these yields the 6 *D-cubes* for the AND gate.

  3 of the NOR gate *D-cubes* are:

  $D\ 0\ \overline{D}$

  $0\ D\ \overline{D}$

  $D\ D\ \overline{D}$

- *D-intersection*: Define how different *D-cubes* can coexist for different gates in a logic circuit.

  $0 \cap 0 = 0 \cap X = X \cap 0 = 0$

  $1 \cap 1 = 1 \cap X = X \cap 1 = 1$

  $X \cap X = X$

  Rule: If one cube assigns a specific signal value, the other cubes must assign either the same signal or X

D-Calculus and D-Algorithm

- *D-intersection* (cont.):

  For example, "*0 X X*" intersect "*1 X X*" is the empty cube (incompatible).

D-intersection	**0**	**1**	**X**	**D**	$\overline{\mathbf{D}}$
0	0	ϕ	0	ψ	ψ
1	ϕ	1	1	ψ	ψ
X	0	1	X	D	$\overline{D}$
D	ψ	ψ	D	μ	λ
$\overline{D}$	ψ	ψ	$\overline{D}$	λ	μ

The greek symbols ϕ and ψ represent incompatible assignments.

If the values are incompatible during propagation or implications, the assignment is called *inconsistent* and *backtracking* is necessary.

Greek symbols μ and λ indicate incompatibilities if **both** are present in *D-cubes* with multiple input *D* and $\overline{D}$.

For example, if only λ occurs, invert the *D*s in the second cube and perform intersection.

D-Calculus and D-Algorithm

- *D-contains*: A cube *A D-contains* cube *B* if the set of *A* cube vertices contains (is a superset of) the *B* cube vertices.

- *Primitive D-cubes of failure (PDF)*: These model faults including:
  (a) *SA0* (represented by D)
  (b) *SA1* (represented by $\overline{D}$)
  (c) Bridging faults (short circuits)
  (d) Arbitrary change in logic gate function (e.g., from AND to OR).

  For the AND gate, the PDF for output *SA0* is "*1 1 D*"
  Here the good machine generates a *1* when both inputs are *1*, while the bad machine generates a *0*.
  The PDFs for the AND gate output *SA1* are "*0 X* $\overline{D}$" and "*X 0* $\overline{D}$".

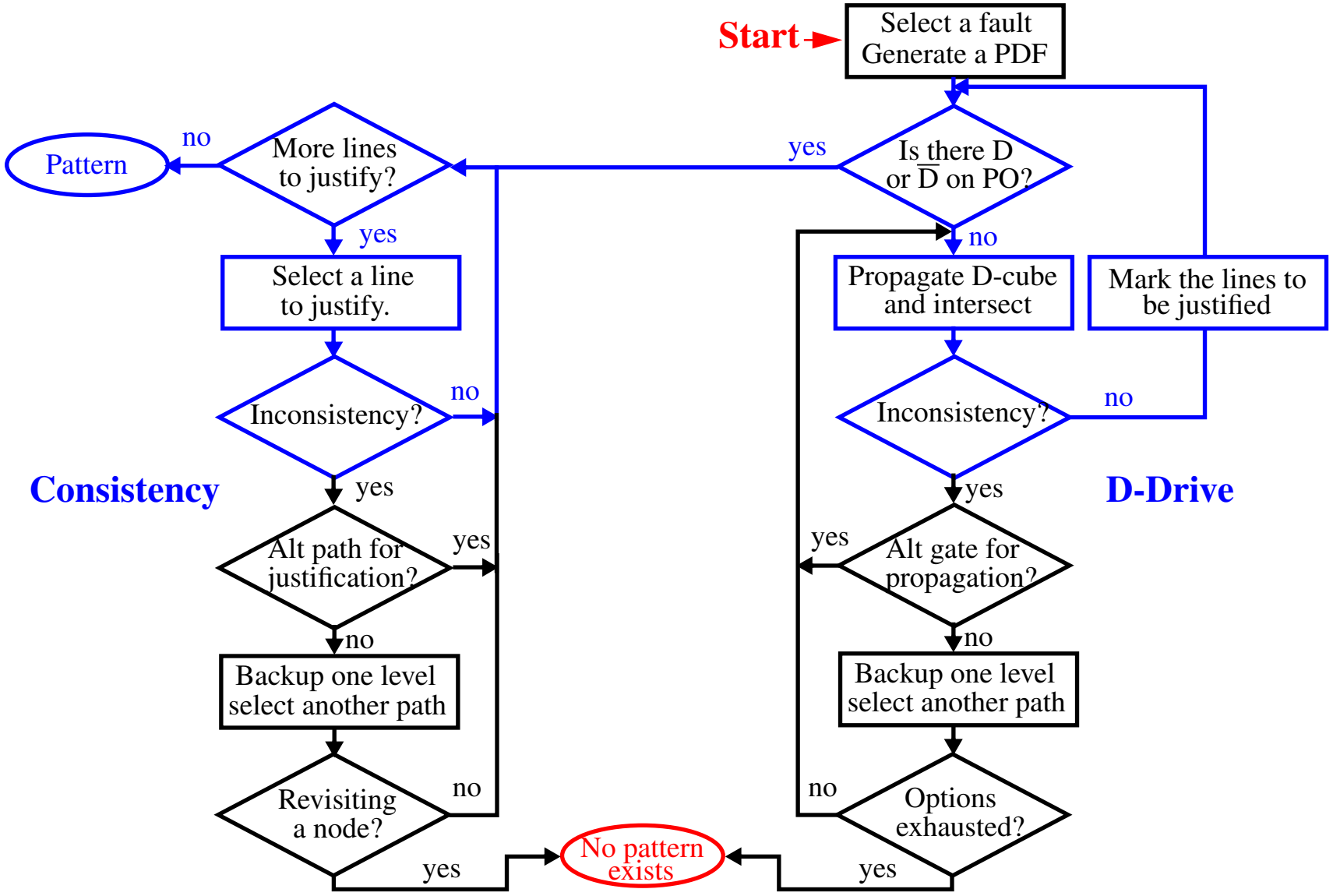
  Note the PDF are distinct from the *propagation D-cubes*.
  The former models a failure at the gate.
  The latter models the conditions for fault effect propagation.

D-Calculus and D-Algorithm

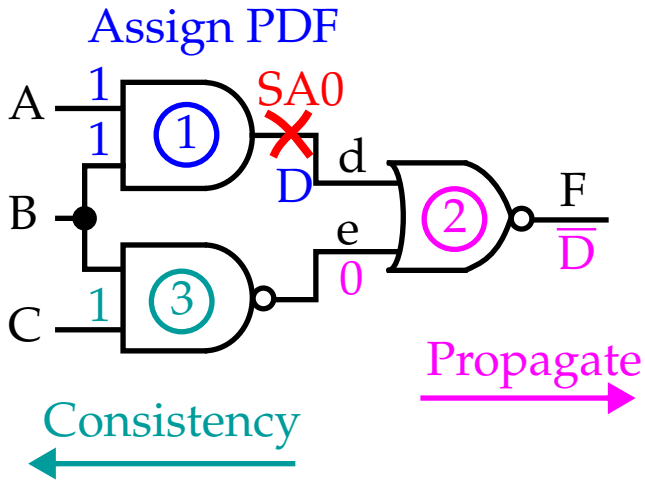
- *Implication procedure*: Consists of the following steps:
  (a) Model the fault with the appropriate PDF.
  (b) Select *propagation D-cubes* to propagate fault-effect to PO(s)(**D-drive**).
  (c) Select singular cover cubes to justify internal circuit signals (**consistency procedure**).

The D-algorithm's main problem is that it selects cubes and singular covers arbitrarily during test generation.

D-ALG


Start
Select a fault
Generate a PDF
Is there D
or D̄ on PO?
yes
no
More lines
to justify?
no
Pattern
yes
Select a line
to justify.
Inconsistency?
no
yes
Consistency
Alt path for
justification?
yes
no
Backup one level
select another path
Revisiting
a node?
no
yes
Propagate D-cube
and intersect
Mark the lines to
be justified
Inconsistency?
no
yes
D-Drive
Alt gate for
propagation?
yes
no
Backup one level
select another path
Options
exhausted?
no
yes
No pattern
exists

D-ALG Examples



Truth Table

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Singular Cover

A	B	C	d	e	F
1	1		1		
0			0		
	0		0		
	1	1		0	
	0			1	
		0		1	
				1	0
			1		0
			0	0	1

Propagation *D-cubes*

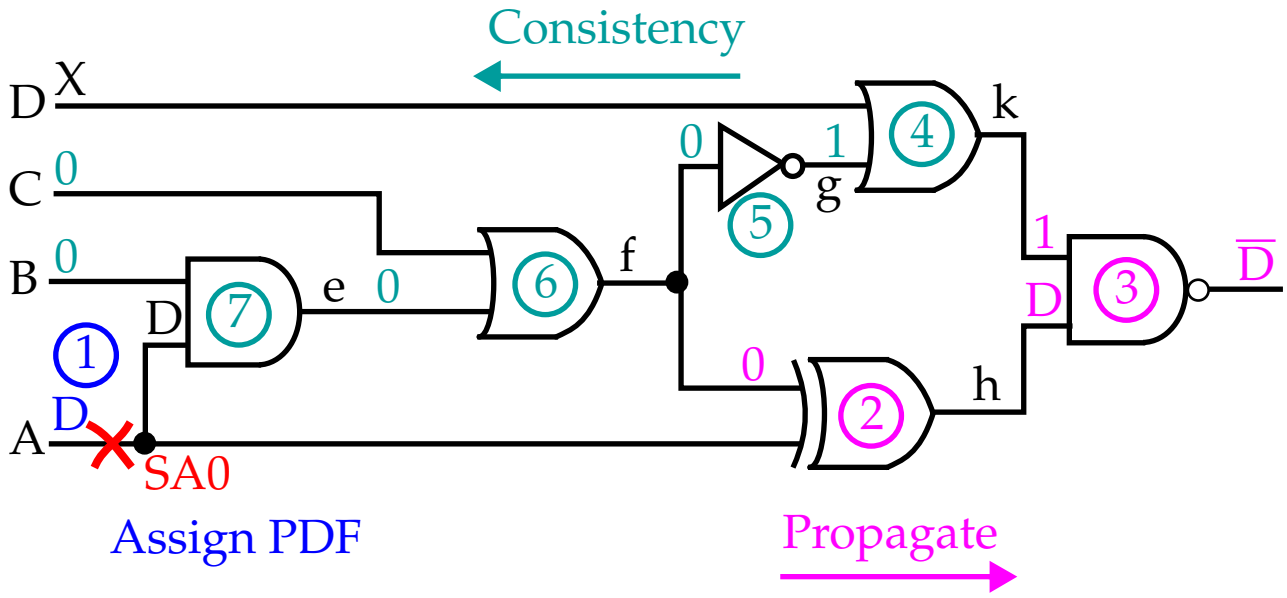
A	B	C	d	e	F
D	1		D		
1	D		D		
D	D		D		
	D	1		$\overline{D}$	
	1	D		$\overline{D}$	
	D	D		$\overline{D}$	
			D	0	$\overline{D}$
			0	D	$\overline{D}$
			D	D	$\overline{D}$

D-ALG Examples

The following procedure is carried out for *d SA0* in the previous circuit:

Step	A	B	C	d	e	F	Type of cube
1	1	1		D			PDF for AND gate
2				D	0	$\overline{D}$	*Propagation D-cube* for NOR gate
3		1	1		0		Singular cover of NAND gate

Example #2:



D-ALG Examples

Steps followed to generate *test cube* (tc):

	Step		A	B	C	D	e	f	g	h	k	L
D-drive	1		D									
	2		**D**					**0**		**D**		
	3		D					0		**D**	**1**	$\overline{\mathbf{D}}$
Consistency	4	or							1		1	
	5	not						0	1			
	6	or			0		0	0				
	7	and		0			0					
		tc	D	0	0		0	0	1	D	1	$\overline{\text{D}}$
D-chain dies												

This example and table is given in Roth's paper.

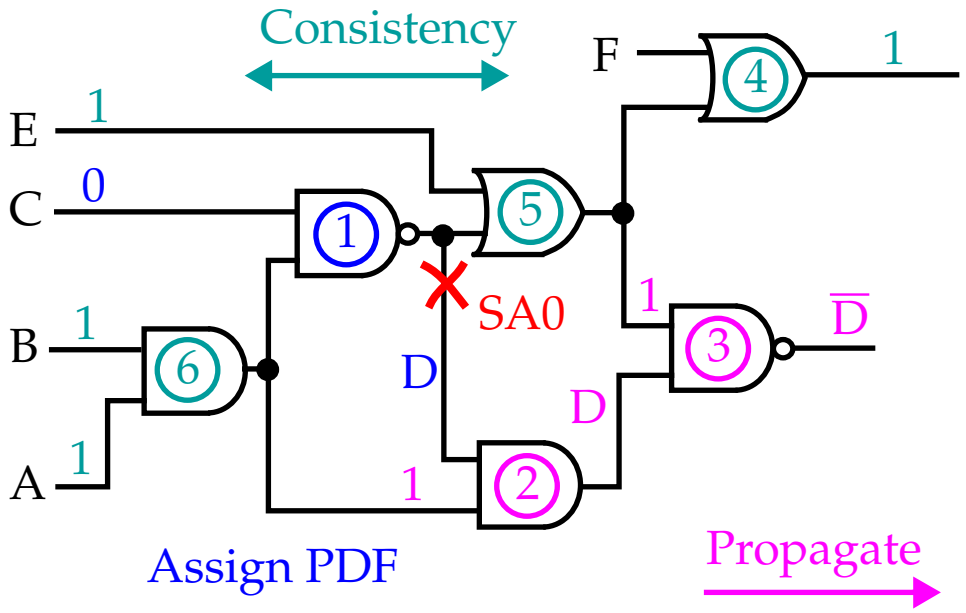
Several other examples are covered in the paper.

Note that all implications are performed in the *consistency* procedure here.

A later example by our authors indicates the *implications* are carried out after each *D propagation* step in the *D-drive?*

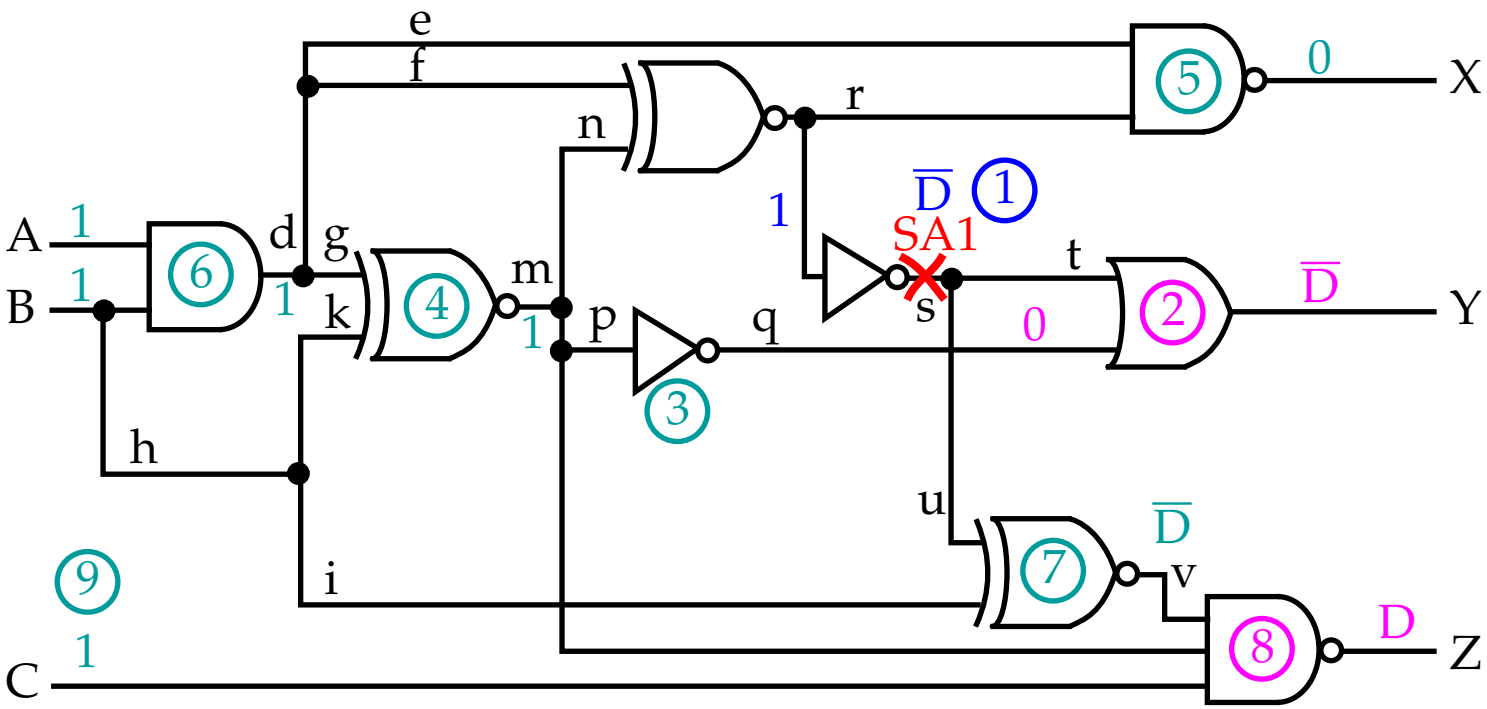
D-ALG Examples

Example #3:



D-ALG Examples

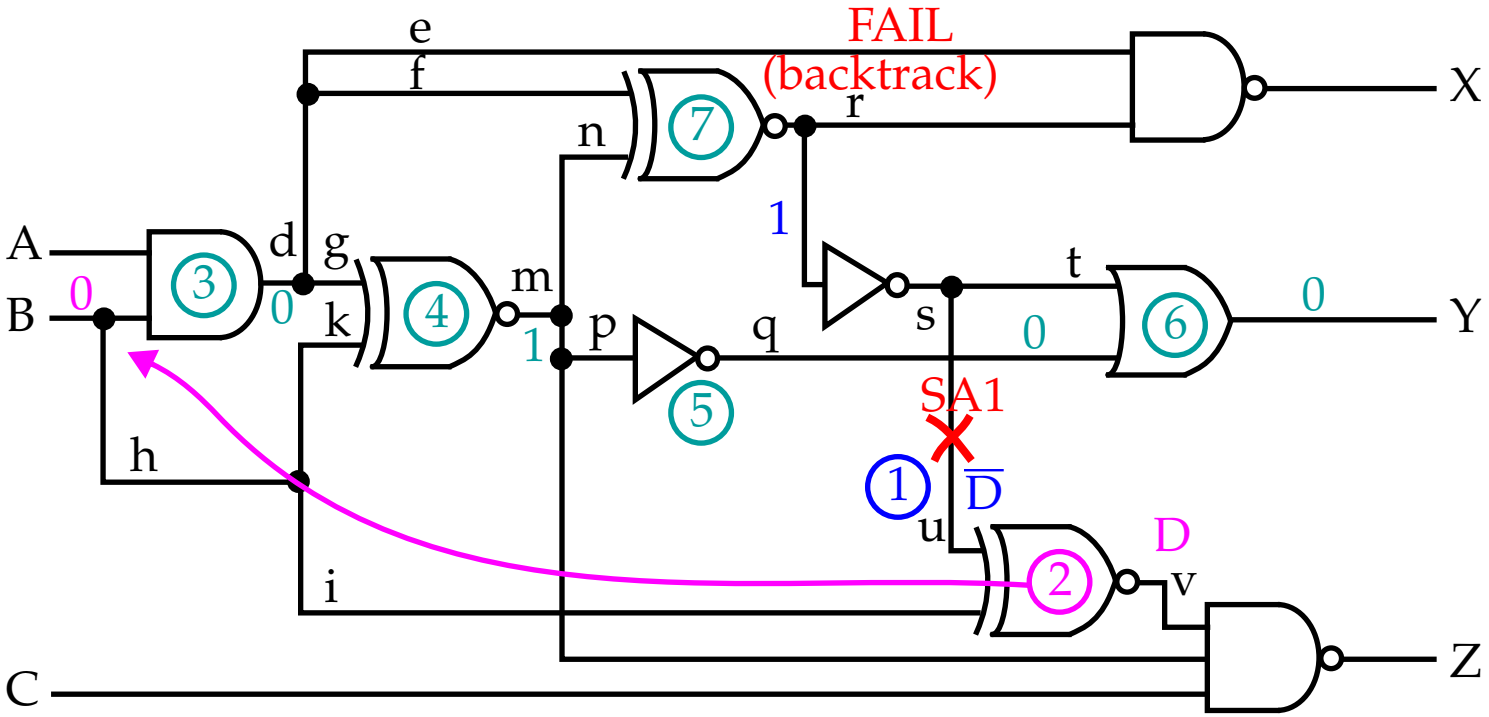
Example #4:



Assign PDF Propagate Consistency

D-ALG Examples

Example #5:



Assign PDF Propagate Implications

The $B = 0$ choice is eventually discovered as a bad choice.

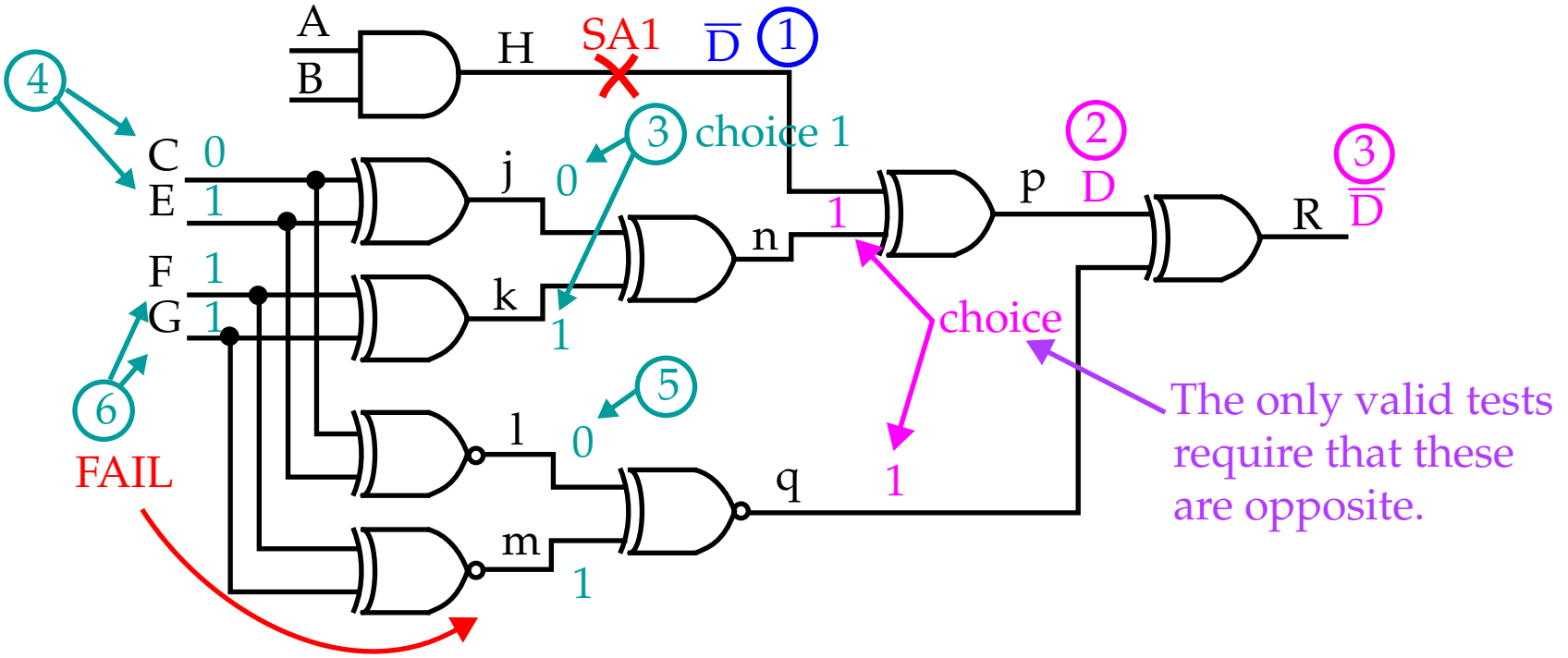
One backtrack to step 2, which sets $B = 1$ and leads to the successful generation of a test.

Note here that *implications* carried out before the *D-drive* is completed.

PODEM

In late '70s, IBM introduced *error correction and translation* (ECAT) to their DRAM to increase reliability.

D-ALG fails on attempts to generate tests for these circuits because the search is not directed.



D-ALG will eventually determine that $n = q$ is not realizable by this circuit.

PODEM

PODEM (**Path-Oriented Decision-Making**) introduced several standard ATPG concepts:

- PODEM expands the binary decision tree **around the PIs** and not around all circuit signals.

  This reduces the size of the tree from 2^n to 2^{num_PIs}.

- D-ALG tended to continue intersecting *D-cubes* even when the *D-frontier* disappeared.

  PODEM introduced a subroutine to test if *D-frontier* still existed.

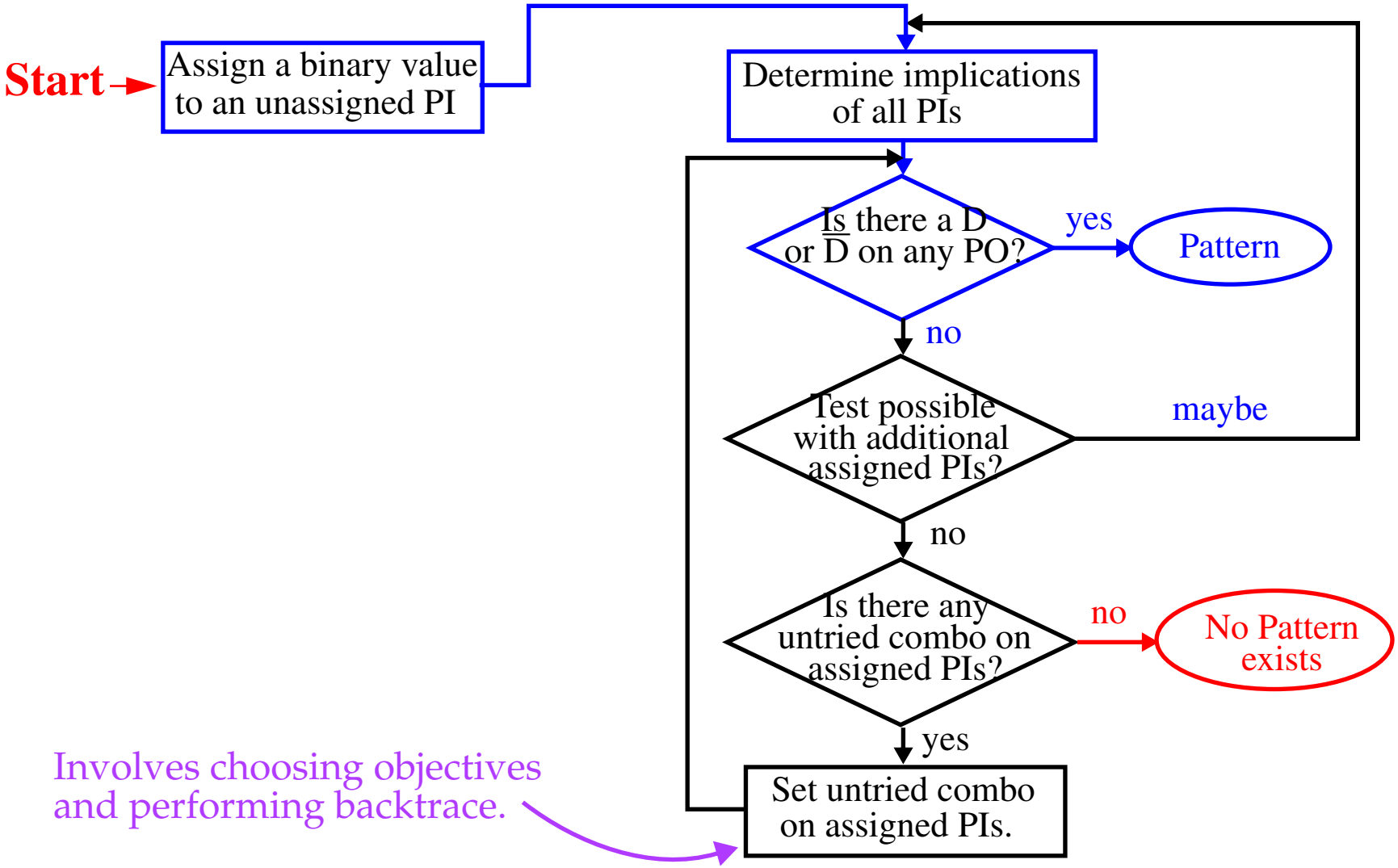
- PODEM introduced *objectives* and realized that choosing PIs to set was important in efficiently realizing objectives.

  Backtracing was used to obtain a PI assignment given an initial objective.

  PODEM considered the length of the path between the objective and the POs and used **controllability measures** to guide the ATPG algorithm.

PODEM

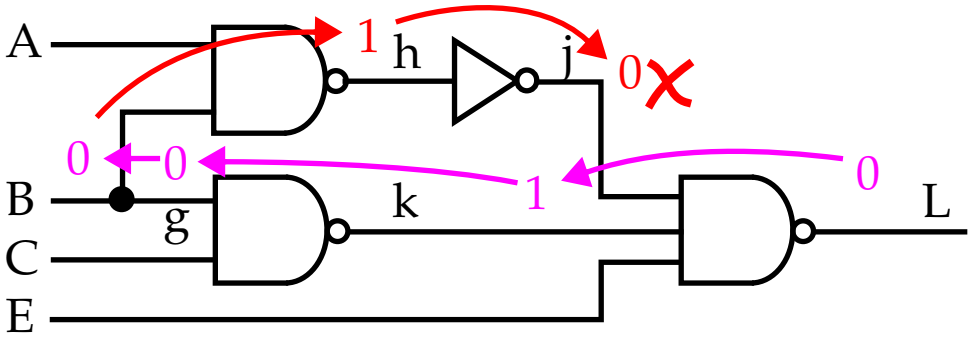
PODEM starts at the PIs instead of at **faulty line** like D-ALG.



FAN

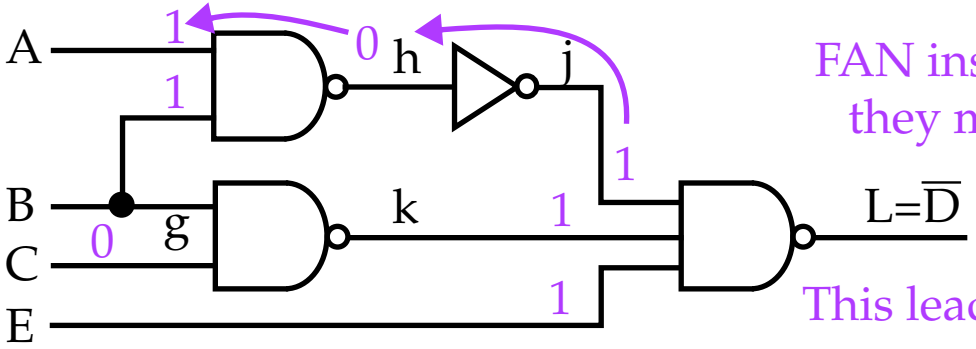
Fujiwara and Shimono introduced several novel concepts to further limit the ATPG search space and accelerate backtracing:

- *Immediate Implications*: PODEM misses opportunities to immediately assign values that are uniquely determined to signals.



Given objective $L = 0$, PODEM would backtrace and assign k=1, g=0 and B=0.

This is unfortunate since B=1 => h=1 and j=0 which prevents objective.



FAN instead sets j, k and E to 1 since they must all be set to justify $L=\overline{D}$.

This leads to unique A and B=1, C=0.

FAN and Other Advanced ATPG Algorithms

Test book describes other novel features of the FAN algorithm.

- *Dominator ATPG Programs*: TOPS (Kirkland and Mercer)
- *Learning ATPG Programs:*
  - SOCRATES (Schulz et al.)
  - EST (Giraldi and Bushnell)
  - Recursive Learning (Kunz and Pradhan)
- *Implication Graph ATPG Algorithms*:
  - NNATPG (Chakradhar et al.)
  - TRAN (Chakradhar et al.)
  - GRASP
  - NEMESIS
  - TEGUS
  - A program by Tafertshofer et al.
- *BDD-Based ATPG Algorithms* (performance poor on multipliers)*:*
  - CATAPULT (Gaede at al.)
  - TSUNAMI (Stanion Bhattacharya)

Test Generation Systems

An ATPG system may contain:

- Fault generator/collapsing program
- RPG program
- Fault simulator
- ATPG program
- Test compactor

Performance criteria include:

- Fault coverage

$$\text{Fault coverage} = \frac{\text{Number of detected faults}}{\text{Total number of faults}}$$

- Fault efficiency

$$\text{Fault efficiency} = \frac{\text{Number of detected faults}}{\text{Total number of faults} - \text{Number of undetectable faults}}$$

- Vector set size
- CPU time

Test Generation Systems

SOCRATES: Starts with RPG (optionally with weighted pattern probabilities), concurrent fault simulation and fault dropping.

32 random patterns are generated in parallel and one concurrent fault simulation is carried out.

The process terminates when no faults are detected after 64 random patterns have been tried.

This is followed with several passes of ATPG.

Pass one is done usually with only 10 backtracks allowed per fault.

Each pattern is then fault simulated against all remaining faults and detected faults are dropped.

Later passes increase the number of backtracks to 50, 100 and finally 10,000.

This process outputs a *test vector file*, a list of *undetected faults*, a list of *redundant faults*, a list of *aborted faults* and a *backtrack distribution file*.

Test Compaction

Many ATPG systems use RPG to get 60% fault coverage, followed by ATPG.

However, many of the RPG patterns may not be as effective at providing "high" fault coverage.

At the end of ATPG, all patterns are fault simulated in **reverse order** of their generation.

Once fault coverage reaches 100%, the remaining RPG patterns are discarded.

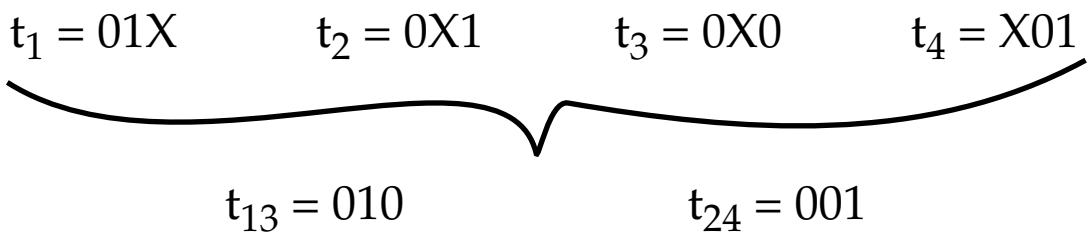
This type of **compaction** greatly reduces the size of the test set.

An additional **static compaction** method is suitable for the ATPG generated patterns, where unassigned inputs are left at *X*.

Two patterns can be combined if they are compatible, as defined by the *D-intersection* operator given earlier.

Test Compaction

The degree of compaction possible depends on the order in which the vectors are processed.



Optimal static compaction algorithms are impractical, so heuristic algorithms are used.

Dynamic compaction immediately assigns *1*'s and *0*'s to the unassigned PIs after the ATPG program generates them.

The *secondary* faults detected allows additional fault dropping.