Testing - Overview

- **Motivation** for testing
  - ◇ fault models
  - ◇ testing method
- Test generation **algorithms**
  - ◇ D-algorithm
  - ◇ PODEM
  - ◇ Socrates
  - ◇ SAT

Testing: WHY?

- Testing is **manufacture verification**

"Is what I manufactured what I designed?"

- Incorrect operation (fault) occurs due to physical defects
  - ◇ Logical faults
    shorts, missing transistors, ...
  - ◇ Parametric faults
    process variations, die anomalies, ...
- Faults may be intermittent or permanent
- Permanent faults may be created during life of the circuit
  physical/thermal stress, radiation, ...

Testing: WHY?

- No manufacturing process can guarantee 100% defect free IC's
- Larger the circuit, greater the probability of defects occurring
- Economics: Cost of detecting a faulty component is lowest before it is packaged and embedded in a system

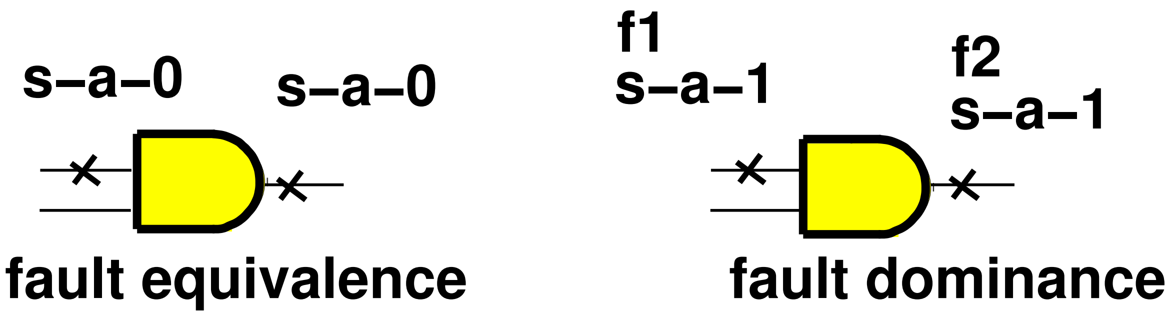
Detection of intermittent and permanent faults $\Rightarrow$ reliable circuit operation

Fault modeling and testing

- Logical faults
  - ◇ Single/multiple stuck-at (*most used*)
  - ◇ CMOS stuck-open
  - ◇ CMOS stuck-on
  - ◇ Bridging faults
- Parametric faults
  - ◇ Low/high voltage/current levels
  - ◇ Gate or path delay-faults

- Parametric (electrical) tests also detect stuck-on faults
- Logical tests detect stuck-at faults
- Transition tests detect stuck-open tests
- Timed transition tests detect delay faults

Fault Reductions



Dominated fault: If every test for fault $f1$ detects $f2$ then $f1$ dominates $f2$

$\Longrightarrow$ only have to generate test for f_1.

Set of faults required for testing is a minimal set with respect to fault equivalence and dominance

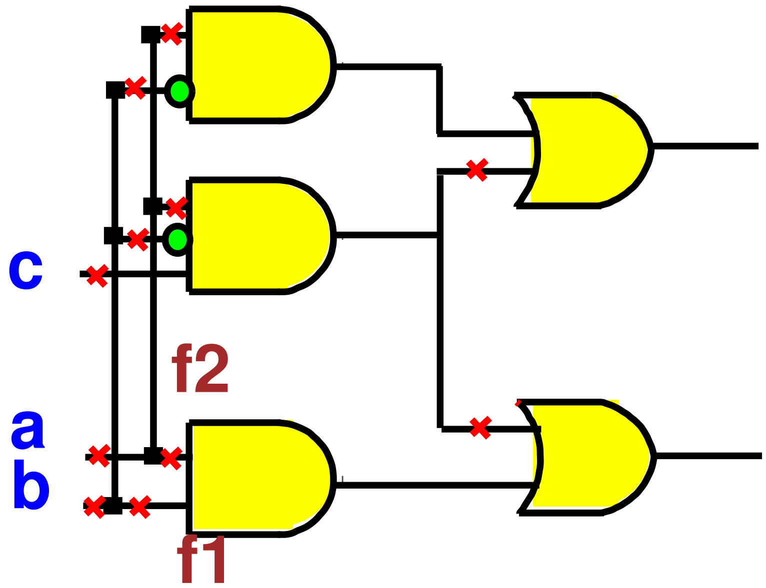
Stuck fault checkpoints: [Davidson] [Kohavi, Kohavi]

Theorem: In a combinational circuit, any set of tests which detects all single (multiple) stuck faults on all primary inputs and all **branches** of fanout points (*including primary inputs*), detects all single (multiple) stuck faults.

The set of primary inputs and branches of fanout points are called the **checkpoints** of the circuit (*i.e.* checkpoints is the set of faults that are tested)

Checkpoints and fault collapsing

Fault collapsing is the process of reducing the size of the set of faults to test using **fault equivalence** and **fault dominance** on the checkpoints of the circuit



X shows checkpoints (s-a-0 and s-a-1 fault on each checkpoint)

$f1$ s-a-0 is equivalent to $f2$ s-a-0. Similarly for other faults.

Exercise: find minimal set of collapsed faults for this circuit. Have to test all sites for both s-a-1 and s-a-0.

Test effectiveness

Undetectable fault: No test exists for fault

Redundant fault: Undetectable fault whose occurrence does not affect circuit operation

- **Testability** $= \frac{\# detectable faults}{\# faults}$

- \# effective faults: #faults - redundant faults.

(These are the ones that we must detect if we want to completely test the chip. Since redundant faults cause no harm, they should not be counted against us.)

- **Fault coverage** $= \frac{\# detectable faults}{\# effective faults}$

(This is a better measure of how well a piece of logic is tested by a test set of vectors.)

- **Test set size** = # of test patterns

Goal:
100% fault coverage (not 100% testability) with a minimum sized test set.

Logic level test generation

- Test generation in combinational circuits
  - ◇ Satisfiability check on Boolean network
    Best solution is SAT based approach
    [Larrabee] [Stephan et al.]
    (*Berkeley experience -*
    not all test community agrees)

- Test generation in sequential circuits
  - ◇ Understood only for stuck-at faults
  - ◇ Tests are sequences of vectors
  - ◇ Still impractical on large circuits ($>$ *30 latches*)

- Convert testing of sequential circuits
  to testing on combinational circuits by
  allowing access to memory elements
  (*called a "scan design"*)
  - ◇ 10-20% area overhead
  - ◇ 10% performance penalty

All combinational test generation algorithms
reduce to SAT

$\Rightarrow$ solve SAT problem efficiently

Combinational test pattern generation - Heuristic

- **Structural search methods**
Perform search on Boolean space using **topology** of the circuit as guide.
(The following is in the order of improvements made over the years)
    - ◇ D-algorithm [Roth 66]
    - ◇ PODEM [Goel 81]
    - ◇ FAN [Fujiwara, Shimono 83]
    - ◇ Socrates [Schulz *et al.* 88]

- **Symbolic and algebraic methods**
Abstract formulation (conceptually elegant but practically infeasible)
    - ◇ ENF [Armstrong 66]
    - ◇ Boolean differences [Sellers *et al.* 68]

Structural techniques were superior to symbolic/algebraic techniques until 1990

- Perform SAT on Boolean network
  [Larrabee 89]

D-Algorithm

Use 5-valued logic for logic value in good (true) and faulty circuit. Implies only one circuit is necessary.

0 $\Rightarrow$ 0 in true circuit, 0 in faulty circuit
1 $\Rightarrow$ 1 in true circuit, 1 in faulty circuit
D $\Rightarrow$ 1 in true circuit, 0 in faulty circuit
$\overline{D}$ $\Rightarrow$ 0 in true circuit, 1 in faulty circuit
X $\Rightarrow$ unknown value exists in either true or faulty circuit

Goal: Find an assignment to primary inputs that causes a D or $\overline{D}$ at some primary output

D-Algorithm Calculus:

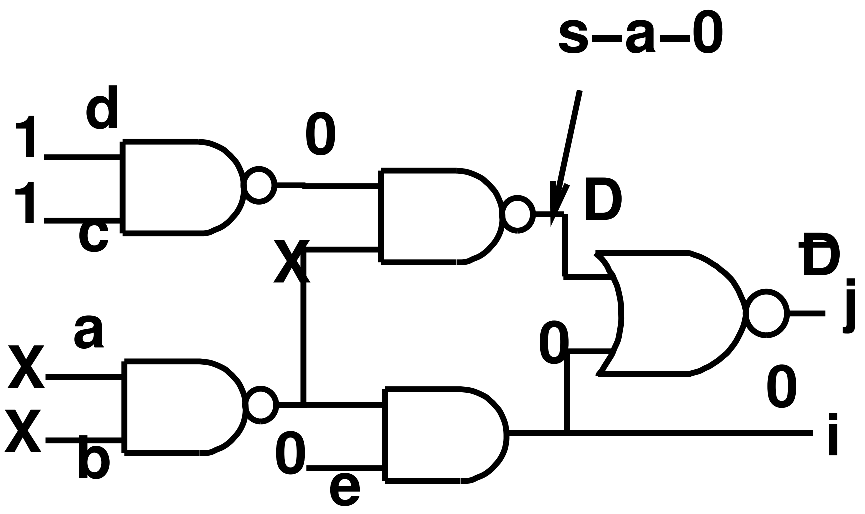
$B = \overline{A}$

A	B
0	1
1	0
X	X
D	$\overline{D}$
$\overline{D}$	D

$C = A\&B$

$A \setminus B$	0	1	X	D	$\overline{D}$
0	0	0	0	0	0
1	0	1	X	D	$\overline{D}$
X	0	X	X	X	X
D	0	D	X	D	0
$\overline{D}$	0	$\overline{D}$	X	0	$\overline{D}$

D-Algorithm example



test is xx110

D-Algorithm

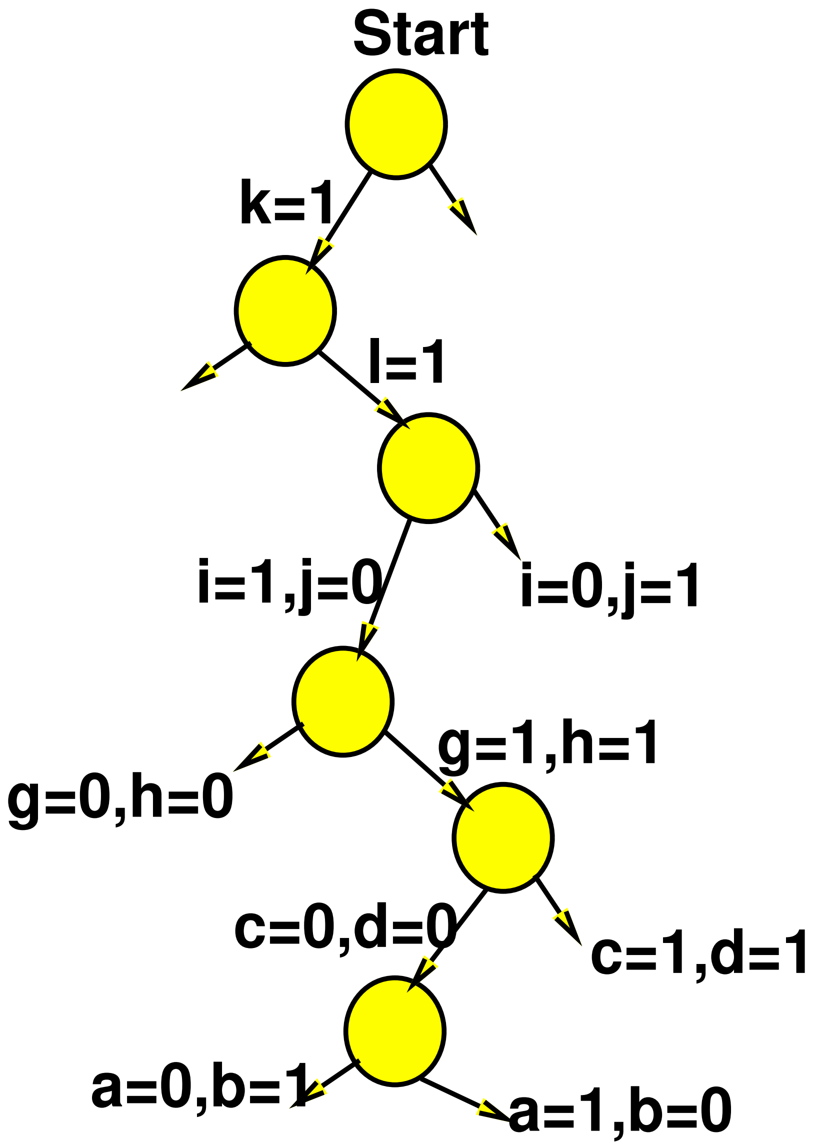
- At fault location determine what value must exist (*e.g.* for a s-a-0 fault, a D is required)

- Assign each internal line of the circuit a value (0, 1, D, $\overline{D}$, X) which is consistent under some primary input vector. A test exists is such a vector is found (with at least one D or $\overline{D}$ at an output), otherwise the fault cannot be tested (redundant).

Note:

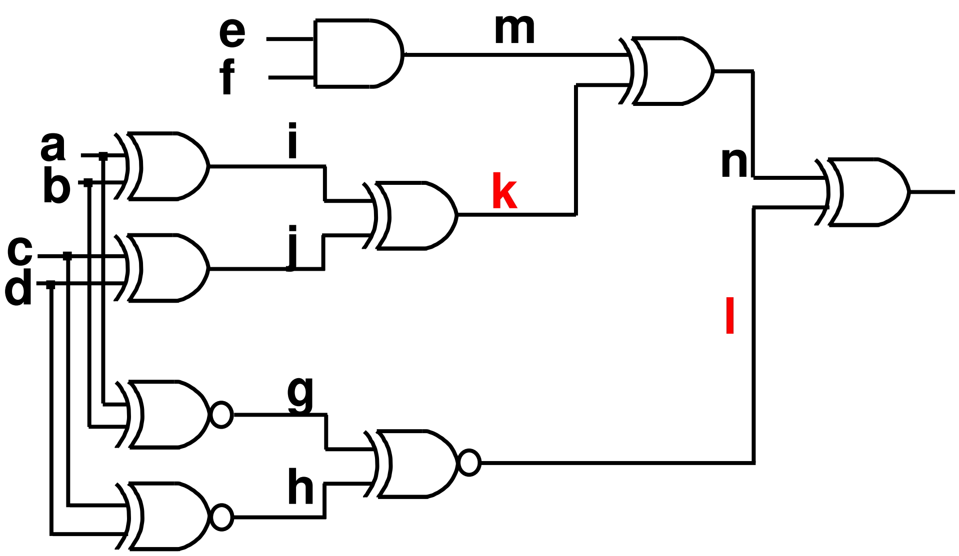
- Given n (internal) lines, 2^n values need to be enumerated in the worst case

D-algorithm: Decision tree



Search performed level by level from PO's to PI's

D-algorithm problem



k and l are complementary signals. Assume $k = 1, l = 1$ is chosen as assignment by D-algorithm. It takes several other assignments before D-algorithm determines this is inconsistent.

Solution: Backtrack only on PI values to determine consistency of signals

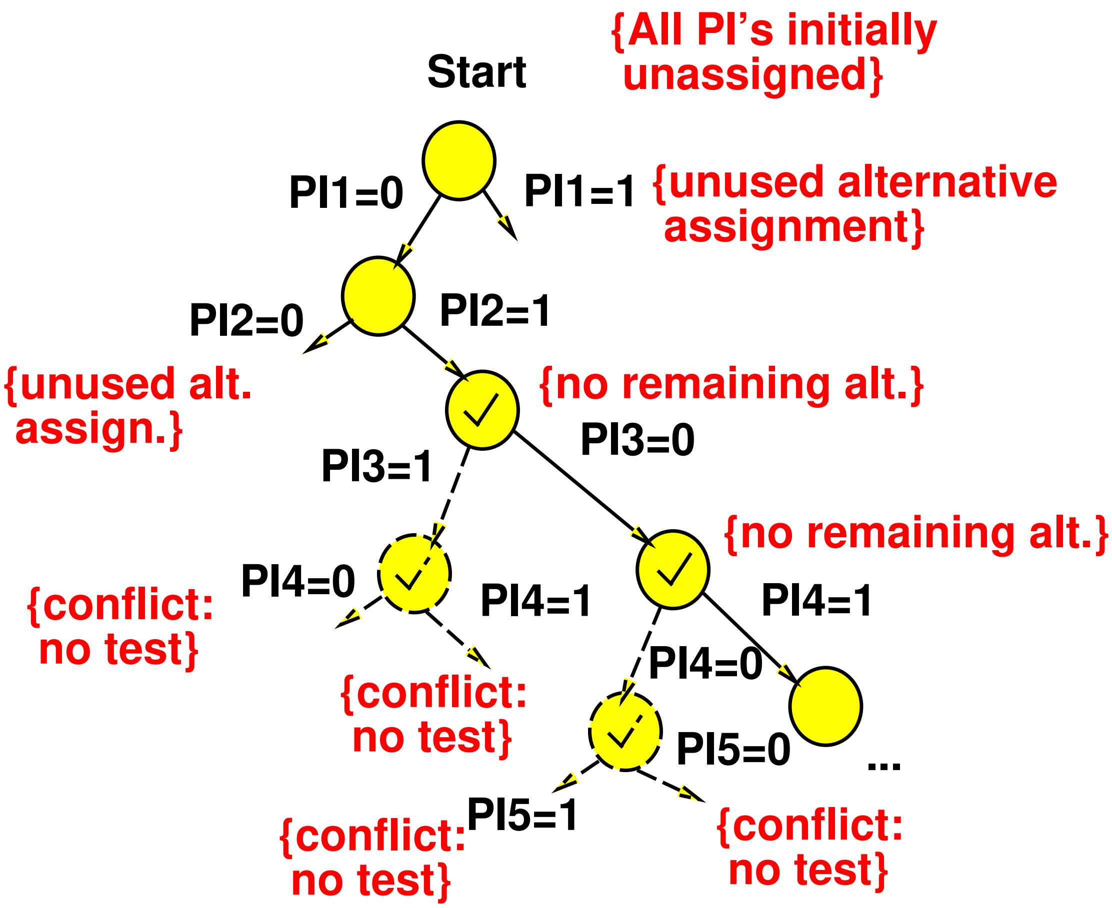
Implicit search enumeration: PODEM:

- Actual space of consistent assignments is only 2^m, where there are m primary inputs.

- Hence, can greatly reduce search space (compared to D-algorithm) by enumerating over primary inputs only.

- PODEM (Path oriented decision making) uses such an algorithm

PODEM: Decision Tree



✓ indicates no remaining alternative at node

PODEM: Algorithm outline

Start with given fault, empty decision tree, all PI's set to X

3 types of operations performed:

- Check if current PI assignment is consistent. If so, choose an unassigned PI and set it to 0 or 1
- If not, if alternative value of currently assigned PI has not been tried, try this alternative and mark this PI as having no remaining alternative
- If no remaining alternative on this PI, backup to previous PI that was assigned, deleting the decision tree below

Algorithm either terminates with a test (all PI's assigned) or proves fault is redundant

PODEM: Heuristics

Choosing which PI to assign next:

This depends on how the fault could propagate to primary output

- Choose closest PO to which fault can propagate and determine which PI affects the progation "the most".

  This is done by computing approximate node controllabilities and observabilities

Heuristic is quite ad-hoc and PODEM is ineffective on large networks with lots of reconvergence.

Socrates:

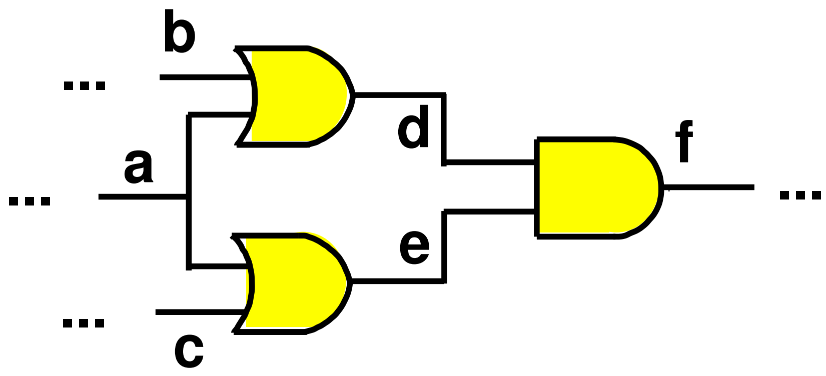
- Provides improvements to PODEM
  - ⋄ implications
  - ⋄ static and dynamic learning

Basic idea:

When a value is set on a node (due to current partial PI assignment) what conclusions can be made?

- ⋄ values on other nodes/ PI's
- ⋄ detection of inconsistencies causing early backtrack

Implications

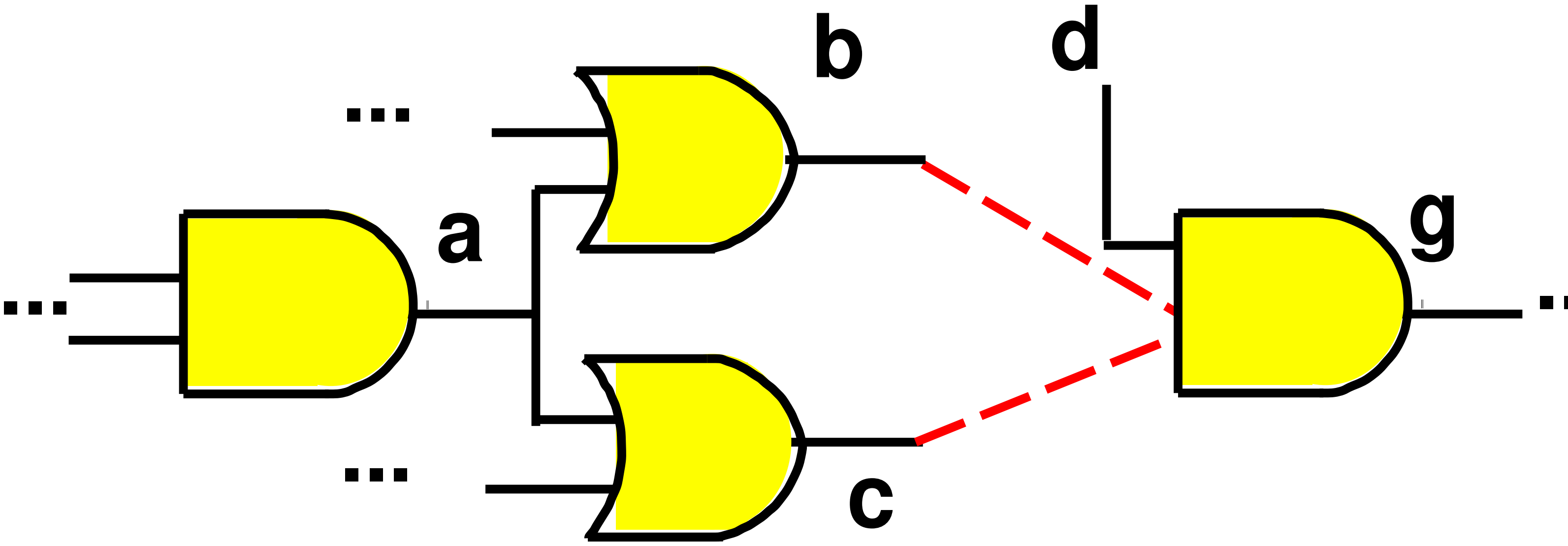


a=1 => d=1 & e=1 => f=1

Hence f=0 => a=0

Implications computed in pre-processing phase and stored for use during backtracking and assignment phase of algorithm

Static and Dynamic Learning



If a has a D value and it must propagate through g, if d is not in transitive fanin of a, d must be set to 1

This is an implication learned from the topology of the network

Static learning: Implications learned in pre-processing phase

Dynamic learning: Implications learned under partial PI assignments

Socrates: Algorithm

- Perform implication of each single lead value and store information
- Given a fault, determine implied values by static learning
- Use PODEM and determine implied values using dynamic learning

Heuristic used to determine when to "learn"
(e.g. don't learn implications which are trivial or already known from previous steps)

Socrates completely subsumed by SAT procedure

P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," UC Berkeley ERL Memo UCB/ERL M92/112 October 1992 and submitted to IEEE Trans. CAD, November 1992

Boolean Network Satisfiability

Problem:

Given a Boolean network find a satisfying assignment to the primary outputs that makes at least one primary output 1

Applications:

- Combinational test pattern generation
- Sequential test pattern generation
- Delay fault test pattern generation
- Timing analysis
- Hazard detection
- Low power

In general, SAT is a good alternative to BDD's if only one solution is needed and a canonical form is not useful.

SAT Outline

- Method of Boolean Differences.
- Formulation as SAT problem.
- Solving SAT problems.
- Greedy, Dynamic Variable Ordering.
- Experimental Results
- SAT Conclusions.

Combinational Test Generation

Structural Search Methods: (*Most widely used.*)

- D-algorithm, Roth '66
- PODEM, Goel '81
- FAN, Fujiwara & Shimono '83
- Socrates, Shulz & Auth '88

Symbolic and Algebraic Methods: *Not considered practical.*

- Poage's Method, Poage '63
- ENF, Armstrong '66
- Boolean Differences, Sellers '68

Hybrid Approach:

- SAT-based, Larrabee '89, Stephan '92

Boolean Differences

General Case:
Let $F(X)$ be a single output function, $X = \{x_1, \ldots, x_n\}$, and $F_z(X)$ be F modified by a fault z. Then

$$T_z = F_z \oplus F$$

characterizes **all** tests for z. In some cases, we can build this as a BDD.

Stuck-at Faults:
Let $G(X)$ be any intermediate node. Rewrite F as $F(X, G)$. Then

$$\begin{aligned} T_0 &= F(X,0) \oplus F(X,1)G(X) \\ T_1 &= F(X,1) \oplus F(X,0)\overline{G}(X) \end{aligned}$$

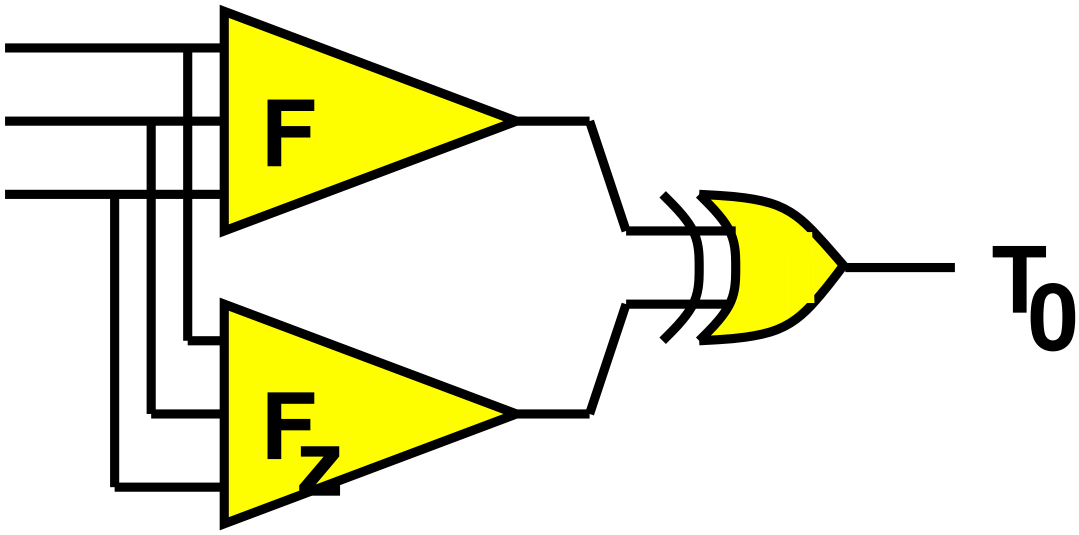
characterize all tests for G s-a-0, G s-a-1 respectively.

$F(X,0) \oplus F(X,1)$ is the **Boolean Difference** of F with respect to G.

Solve T_0, T_1 algebraically (Sellers *et al.*).

Conjunctive Normal Form

- Consider T_0 in a circuit representation:



- Express network T_0 in CNF using characteristic function of each gate.
  Example: a = x y
  $$F = (\overline{x} + \overline{y} + a)(x + \overline{a})(y + \overline{a})$$
  Conjunction over all gates describes T_0.

- Add clause with only one literal (T_0) to the CNF for circuit. This says that we want to find an input where $T_0 = 1$, i.e. a test for the fault. Since entire formula is in CNF, problem is *SATISFIABILITY*. . . .

Satisfiability

- The *original* NP-complete problem.

SAT

Instance: A set U of variables and a collection C of clauses over U.
Question: Is there a satisfying truth assignment for C?

Cook's Theorem (1971): SAT is NP-complete. (See Garey & Johnson, '79).

- Still NP-complete if:
  $\forall c \in C \;\; |c| = 3$ (3SAT)
  $\forall u \in U$ at most 3 clauses contain u or $\overline{u}$.

- Polynomial if:
  $\forall c \in C \;\; |c| \leq 2$ (2SAT)

SAT vs. Tautology

- Duality:

SAT	non-Tautology
CNF, product of sums	DNF, sum of products
Is expression ever true?	Is expression ever false?*
n-clause	$\lvert$support$\rvert - n$ cube
null clause: always false	universal cube: always true
assign variable x	cofactor with respect to x
backtrack if null clause	backtrack if universal cube
$(a+b)(\overline{a}+c+d)$	$ab+\overline{a}cd$
unate: easy to satisfy	unate: fast tautology test

* Recall tautology asks "Is the expression always true?"

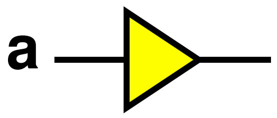
- Why not just use Espresso?

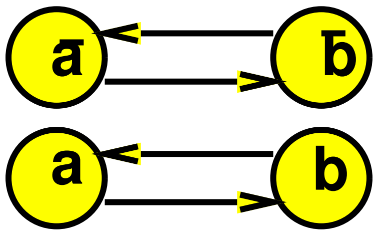
Unate recursive paradigm for tautology is inefficient for formulas derived from gate-level characteristic functions.

— Extremely binate cubes, few literals.
— Experiments with Espresso confirm this.

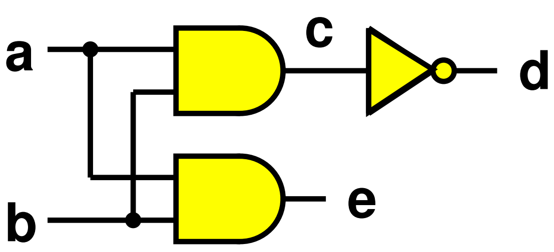
Implication Graph

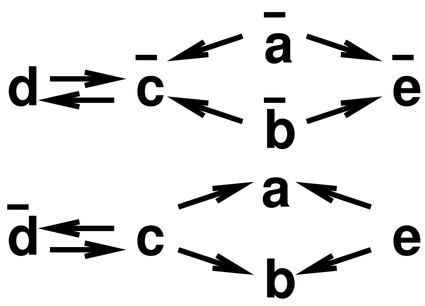
- View 2-clauses as pair of implications.
  — $(a + \overline{b}) \Leftrightarrow (\overline{a} \rightarrow \overline{b}) \wedge (b \rightarrow a)$
  — Forms an *implication graph*.
- Strongly-connected components (SCCs) are equivalent variables (inverters, buffers).





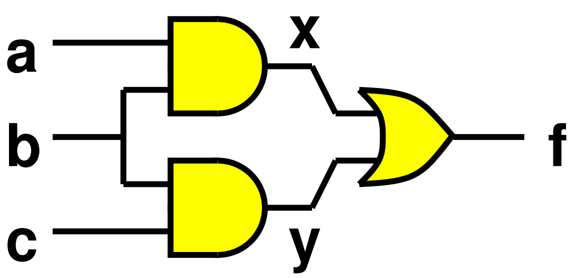
- More complex equivalence not detected.
  Example: symmetry versus SCC.





$$(\overline{a}+\overline{b}+c)(a+\overline{c})(b+\overline{c})(\overline{a}+\overline{b}+e)(a+\overline{e})(b+\overline{e})(\overline{c}+\overline{d})(c+d)$$

Nonlocal Implications



$$(\overline{a}+\overline{b}+x)(a+\overline{x})(b+\overline{x})$$

$$(\overline{b}+\overline{c}+y)(b+\overline{y})(c+\overline{y})$$

$$(x+y+\overline{f})(\overline{x}+f)(\overline{y}+f)$$

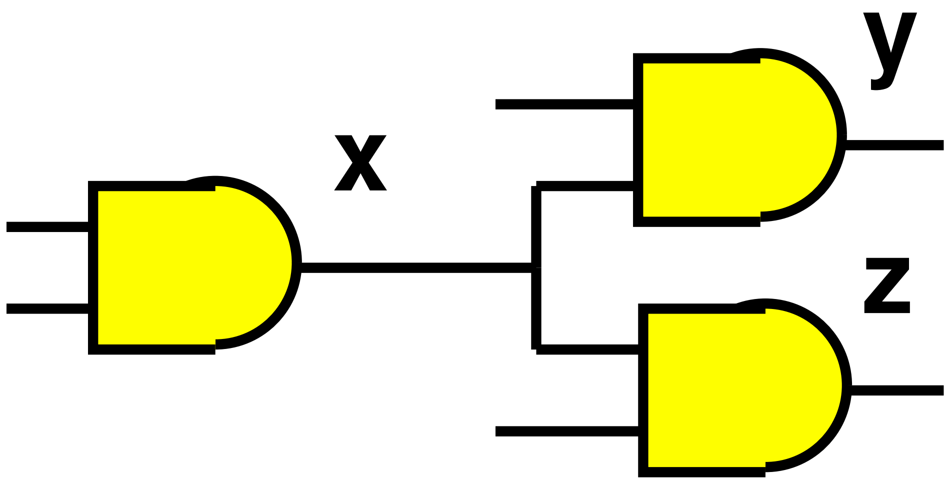
- Finding nonlocal implications for b:
  1. Try asserting $(\overline{b})$.
  2. $(b+\overline{x}) \Rightarrow (\overline{x})$, and $(b+\overline{y}) \Rightarrow (\overline{y})$
  3. $(x+y+\overline{f}) \Rightarrow (\overline{f})$
  4. Thus $(\overline{b}) \Rightarrow (\overline{f})$, so deduce $(f) \Rightarrow (b)$.
- Contradiction $\Rightarrow$ fix to constant.
- Repeat for every formula variable.
- Crucial for hard faults (esp. redundancies).

Active Clauses

- Problem: Good/faulty circuits related only at I/Os, slow to find contradictions.
- Solution: active clauses define relationships for internal nodes (Larrabee '90).
- Active variable x_a is true if net x differs in good and faulty network. Here x_g refers to signal x in the good circuit, and x_f to x in the faulty circuit:

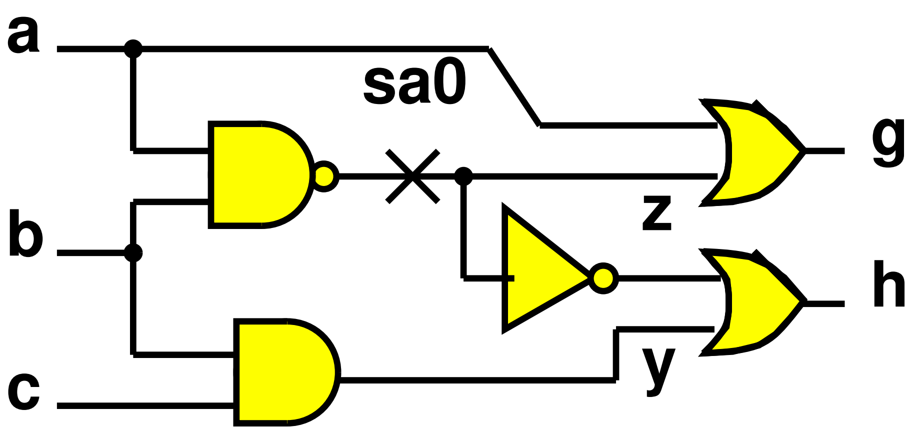
$$(\overline{x}_a + x_g + x_f)(\overline{x}_a + \overline{x}_g + \overline{x}_f)$$

- If gate is active, we require that some fanout must be.



$$(\overline{x}_a + y_a + z_a)$$

Example Formula



- Good Circuit (32 literals):
  $(a + x)(b + x)(\overline{x} + \overline{a} + \overline{b})(\overline{a} + g)(\overline{x} + g)(\overline{g} + a + x)$
  $(x + z)(\overline{x} + \overline{z})(\overline{z} + h)(\overline{y} + h)(\overline{h} + z + y)$
  $(b + \overline{y})(c + \overline{y})(y + \overline{b} + \overline{c})$

- Faulty Circuit (18 literals):
  $(\overline{a} + g_f)(\overline{x}_f + g_f)(\overline{g}_f + a + x_f)$
  $(x_f + z_f)(\overline{x_f} + \overline{z}_f)(\overline{z_f} + h_f)(\overline{y}_f + h_f)(\overline{h}_f + z_f + y_f)$

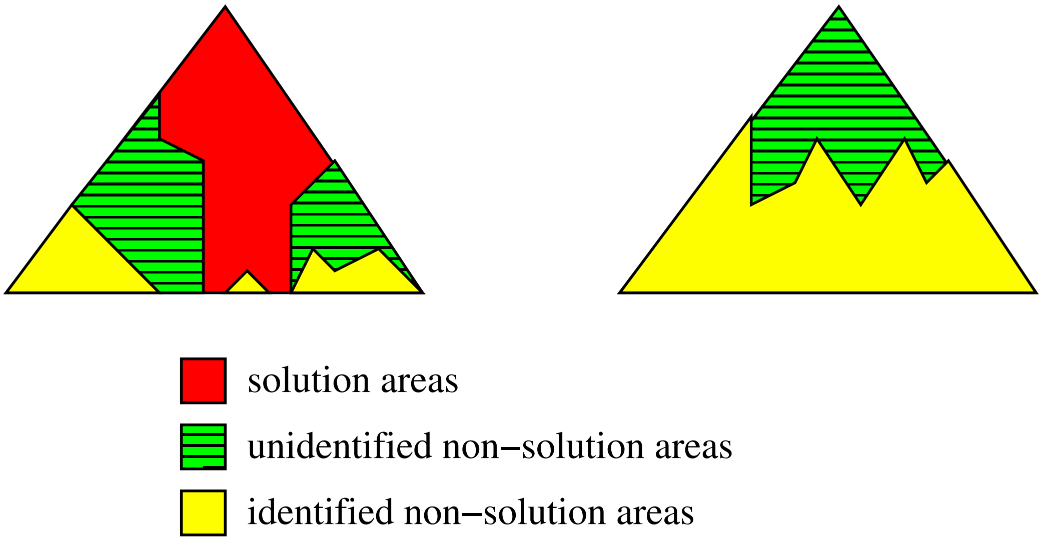
- Active Clauses (29 literals):
  $(\overline{x}_a + x + x_f)(\overline{x}_a + \overline{x} + \overline{x}_f)(\overline{z}_a + z + z_f)(\overline{z}_a + \overline{z} + \overline{z}_f)$
  $(\overline{g}_a + g + g_f)(\overline{g}_a + \overline{g} + \overline{g}_f)(\overline{h}_a + h + h_f)(\overline{h}_a + \overline{h} + \overline{h}_f)$
  $(\overline{x}_a + z_a + g_a)(\overline{z}_a + h_a)$

- Fault Site (3 literals) Goal (2 literals):
  $(x)(\overline{x}_f)(x_a)$ $(g_a + h_a)$

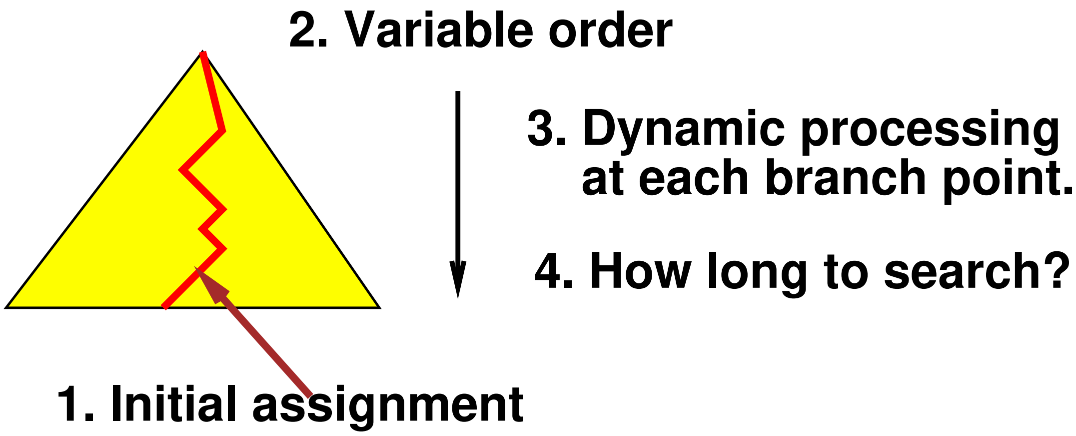
Search Strategies

- Use a basic branch-and-bound approach.

Has a solution: testable fault	Has no solutions: redundant fault



- Four basic parameters



Orthogonal Strategies

- In *theory*, any complete algorithm will find a test for every fault if one exsits.
- In *practice*, we cannot afford to wait for the worst case.
- Compromise: try several strategies in succession for a short time (backtrack limit).
  - Improves average performance.
  - Increases robustness.
  - Has difficulty with “hard” redundancies.
- Two strategies which complement each other are called *orthogonal*.

Static Variable Ordering

- Larrabee's heuristics (LSAT)
  - Add clauses for structural heuristics (dominators, critical paths, etc.)
  - Static variable ordering.
  - Three search strategies.
  - Static nonlocal implications (after all faults have been tried without them).
- No results reported without random patterns.
- Implemented as `atpg` command in `misII`.

Benchmark Networks

- MCNC Test Generation Benchmarks

Network	PI	PO	Gates	Faults
C432	36	7	159	413
C499	41	32	370	850
C880	60	26	320	703
C1355	41	32	506	1 058
C1908	33	25	441	1 102
C2670	233	140	734	1 678
C3540	50	22	961	2 177
C5315	178	123	1 473	3 798
C6288	32	32	2 384	4 835
C7552	207	108	2 164	4 901
S1494	14	25	577	1 220
S5378	214	213	1 149	2 850
S9234	247	250	2 264	4 675
S13207	700	790	3 287	7 086
S15850	611	684	4 057	8 340
S35932	1 763	2 048	12 236	28 222
S38417	1 664	1 742	10 232	21 113
S38584	1 464	1 730	11 774	26 008
Total				121 029

- Full scan assumed for sequential networks.

Larrabee's Heuristics

- **With** random tests (sec./# ab.).

Ckt.	LSAT		Socrates	
c499	72.	0	4.9	0
c880	49.	0	5.2	0
c1355	99.	0	13.9	0
c1908	184.	0	33.8	0
c2670	479.	0	57.5	0
c3540	607.	0	56.6	0
c5315	464.	0	31.3	0
c6288	227.	0	87.0	0
c7552	2 052.	0	248.0	0

- **Without** random tests (sec./# ab.).

Ckt.	LSAT		Socrates	
c499	5 240.	3	29.	0
c880	1 730.	0	10.	0
c1355	9 540.	1	44.	0
c1908	31 000.	53	81.	0
c2670	8 600.	3	80.	0
c3540	110 000.	64	159.	0
c5315	56 000.	145	102.	0
c6288	980 000.	4 506	139.	0
c7552	270 000.	197	418.	0

Greedy Heuristics

- Heuristics must be evaluated without using random tests.
- Static variable ordering is not effective.
- Dynamic ordering can require too much computation at each branch of the search.
- Solution: greedy, dynamic orderings.
  At each branch point, select:
  — 1^{st} literal in 1^{st} unsatisfied clause.
  — last literal in 1^{st} unsatisfied clause.
  — 1^{st} literal in last unsatisfied clause.
  — last literal in last unsatisfied clause.
- Results: improved performance and robustness.

Algorithm

- For each uncaught fault:

```
extract CNF formula
try 4 greedy strategies.
if all fail then
    find static NLI
    repeat 4 strategies
end if
if satisfied then
    fault simulate test
else
    flag redundant fault
end if
```

- Simple algorithm
  — no testability measures
  — no 5, 9-valued algebras
  — no multiple backtracing

- Standalone program `TEGUS` (also in `sis 1.1` but slower by 1-2 orders of magnitude).

Experiments

- Theoretical worst case performance is same for all complete `atpg` algorithms.

- Heuristics must be evaluated by experiment.

- To compare heuristics:
  - Use the 10 ISCAS '85 benchmark networks and 8 larger ISCAS '89 networks (assuming full scan).
  - Run on same model computer as other reported results, with same options.
  - Compare CPU time (not backtracks).
  - Try with and without random tests.

TEGUS Base Results

- Robust: 0 aborted faults on ISCAS '85/'89 networks *without fault simulation*.

- Efficient: for 18 ISCAS networks

  - no fault sim: 10 min. total CPU*
    75% extract, 25% SAT.

  - with random: 1 min. total CPU*
    55% extract, 20% fault sim,
    15% SAT, 10% I/O.
    10 Mbyte peak memory

- Simple: 3k lines of code
  — 300 to extract CNF formula
  — 800 for SAT package

*(DEC 3000/500)

Robustness

- Available results with **no** fault simulation for ISCAS '85/'89 networks.

Algorithm	No. of Circuits	Aborted Faults	Normalized CPU Time
CHE88	7	798	1.6
CHN89	9	1 208	2.2
JAC89	8	693	6.6
GIR90	10	0	16.0
ABR90	6	408	30.0
RAJ90	10	1 268	24.0
GIR91	10	0	12.0
TEGUS	18	0	1.0

See Tech Report UCB/ERL M92/112, also TCAD Sept '96, Vol 15, No.9, pp. 1167-1175

- Algorithms with backtrack limits are incomplete.

- Heuristics which abort on fewer faults are more robust.

Efficiency

- Available results with random tests for ISCAS '85/'89 networks.

Algorithm	No. of Circuits	Aborted Faults	Normalized CPU Time
SIM89	6	0	30.0
SCH89	10	0	1.7
MIN89	9	16	22.0
LAR89	10	0	31.0
JAC89	10	0	6.8
SCH90	18	0	36.0
WAI90	18	0	1.0
MAH90	15	0	9.4
CHK91	1	0	2 900.0
GIR91	10	0	25.0
MAT92	18	0	0.9
TEGUS	18	0	1.0

See Tech Report UCB/ERL M92/112.

- Random tests mask effectiveness of deterministic algorithm.

SAT Conclusions

- Hybrid test generation algorithm extracts CNF formula for structural fault, and applies generic SAT algorithm to generate a test.

- Greedy, dynamic ordering heuristics are as efficient as best published results for structural algorithms.

- Heuristics must be carefully compared using real implementations, and without random tests.

- Other fault models and test restrictions can be accomodated without modifying the generic SAT solver.