Dependable Computing
A Multilevel Approach

Behrooz Parhami
University of California, Santa Barbara

Oct. 2019

Part III — Faults: Logical Deviations

STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)
Goals
Models

Part II — Defects: Physical Imperfections (The Device-Level View)
Methods
Examples

Part III — Faults: Logical Deviations (The Circuit-Level View)
Methods
Examples

Part IV — Errors: Informational Distortions (The State-Level View)
Methods
Examples

Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)
Methods
Examples

Part VI — Degradations: Behavioral Lapses (The Service-Level View)
Methods
Examples

Part VII — Failures: Computational Breaches (The Result-Level View)
Methods
Examples

Appendix: Past, Present, and Future

1. Background and Motivation
2. Dependability Attributes
3. Combinational Modeling
4. State-Space Modeling
5. Defect Avoidance
6. Defect Circumvention
7. Shielding and Hardening
8. Yield Enhancement
9. Fault Testing
10. Fault Masking
11. Design for Testability
12. Replication and Voting
13. Error Detection
14. Error Correction
15. Self-Checking Modules
16. Redundant Disk Arrays
17. Malfunction Diagnosis
18. Malfunction Tolerance
19. Standby Redundancy
20. Resilient Algorithms
21. Degradation Allowance
22. Degradation Management
23. Robust Task Scheduling
24. Software Redundancy
25. Failure Confinement
26. Failure Recovery
27. Agreement and Adjudication
28. Fail-Safe System Design
About This Presentation

This presentation is intended to support the use of the textbook *Dependable Computing: A Multilevel Approach* (traditional print or on-line open publication, TBD). It is updated regularly by the author as part of his teaching of the graduate course ECE 257A, Fault-Tolerant Computing, at Univ. of California, Santa Barbara. Instructors can use these slides freely in classroom teaching or for other educational purposes. Unauthorized uses, including distribution for profit, are strictly prohibited. © Behrooz Parhami

<table>
<thead>
<tr>
<th>Edition</th>
<th>Released</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
</tr>
</thead>
</table>
9 Fault Testing
The good news is that the tests don’t show any other problems.
## STRUCTURE AT A GLANCE

| Part I — Introduction: Dependable Systems (The Ideal-System View) | Goals | 1. Background and Motivation  
2. Dependability Attributes  
3. Combinational Modeling  
4. State-Space Modeling |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Models</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Part II — Defects: Physical Imperfections (The Device-Level View) | Methods | 5. Defect Avoidance  
6. Defect Circumvention  
7. Shielding and Hardening  
8. Yield Enhancement |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10. Fault Masking  
11. Design for Testability  
12. Replication and Voting |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14. Error Correction  
15. Self-Checking Modules  
16. Redundant Disk Arrays |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

18. Malfunction Tolerance  
19. Standby Redundancy  
20. Resilient Algorithms |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

22. Degradation Management  
23. Robust Task Scheduling  
24. Software Redundancy |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Part VII — Failures: Computational Breaches (The Result-Level View) | Methods | 25. Failure Confinement  
26. Failure Recovery  
27. Agreement and Adjudication  
28. Fail-Safe System Design |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Appendix: Past, Present, and Future
9.1 Overview and Fault Models

The faulty state and transitions into and out of it
A Taxonomy of Fault Testing

TEST GENERATION
(Preset/Adaptive)

- FUNCTIONAL
  (Exhaustive/Heuristic)
- STRUCTURAL
  (Analytic/Heuristic)

FAULT MODEL
switch-or gate-level
(single/multiple stuck-at, bridging, etc.)

FAULT COVERAGE
DIAGNOSIS EXTENT
none (check-out, go/no-go) to full resolution

ALGORITHM
D-algorithm, boolean difference, etc.

FAULT TESTING
ENGINEERING
Manufacturing
Maintenance
Correct design? Correct implementation? Correct operation?

TEST VALIDATION

- THEORETICAL
  (Exhaustive/Heuristic)
- EXPERIMENTAL
  (Analytic/Heuristic)

SIMULATION
software (parallel, deductive, concurrent) or hardware (simulation engine)

FAULT INJECTION
MANUAL
AUTOMATIC (ATE)
TEST MODE (BIST)

CONCURRENT on-line testing (self-checked design)

TEST APPLICATION
EXTERNALLY CONTROLLED
INTERNALLY CONTROLLED

- EXTERNALLY CONTROLLED
  Off-line testing

- INTERNALLY CONTROLLED

FAULT TESTING
(Engineering, Manufacturing, Maintenance)
Off-line testing

Correct design?
Correct implementation?
Correct operation?
Requirements and Setup for Testing

Test pattern source \rightarrow \text{Circuit under test (CUT)} \rightarrow \text{Comparator} \rightarrow \text{Reference value} \rightarrow \text{Pass/Fail}

Easier to test if direct access to some inner points is possible

Testability requires **controllability** and **observability** (redundancy may reduce testability if we are not careful; e.g., TMR)

Reference value can come from a “gold” version or from a table

Test patterns may be randomly generated, come from a preset list, or be selected according to previous test outcomes

Test results may be compressed into a “signature” before comparing

Test application may be off-line or on-line (concurrent)
Importance and Limitations of Testing

Important to detect faults as early as possible

Approximate cost of catching a fault at various levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>$1</td>
</tr>
<tr>
<td>Board</td>
<td>$10</td>
</tr>
<tr>
<td>System</td>
<td>$100</td>
</tr>
<tr>
<td>Field</td>
<td>$1000</td>
</tr>
</tbody>
</table>

Test coverage may be well below 100% (model inaccuracies and impossibility of dealing with all combinations of the modeled faults)

“Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often.” Steve C. McConnell

“Program testing can be used to show the presence of bugs, but never to show their absence!” Edsger W. Dijkstra
Fault Models at Different Abstraction Levels

Fault model is an abstract specification of the types of deviations in logic values that one expects in the circuit under test.

Can be specified at various levels: transistor, gate, function, system.

Transistor-level faults
- Caused by defects, shorts/opens, electromigration, transients, . . .
- May lead to high current, incorrect output, intermediate voltage, . . .
- Modeled as stuck-on/off, bridging, delay, coupling, crosstalk faults
- Quickly become intractable because of the large model space

Function-level faults
- Selected in an ad hoc manner based on the function of a block (decoder, ALU, memory)

System-level faults (malfunctions, in our terminology)
- Will discuss later in Part V
Gate- or Logic-Level Fault Models

Most popular models (due to their accuracy and relative tractability)

Line stuck faults
- Stuck-at-0 (s-a-0)
- Stuck-at-1 (s-a-1)

Line bridging faults
- Unintended connection (wired OR/AND)

Line open faults
- Often can be modeled as s-a-0 or s-a-1

Delay faults (less tractable than the previous fault types)
- Signals experience unusual delays

Other faults
- Coupling, crosstalk
9.2 Path Sensitization and D-Algorithm

The main idea behind test design: control the faulty point from inputs and propagate its behavior to some output

Example: s-a-0 fault
Test must force the line to 1

Two possible tests
(A, B, C) = (0 1 1) or (1 0 1)

This method is formalized in the D-algorithm

**D-calculus**
1/0 on the diagram above is represented as $D$
0/1 is represented as $\overline{D}$
Encounters difficulties with XOR gates (PODEM algorithm fixes this)
Selection of a Minimal Test Set

Each input pattern detects a subset of all possible faults of interest (according to our fault model)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>s-a-0</td>
<td>s-a-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>-</td>
</tr>
</tbody>
</table>

Choosing a minimal test set is a covering problem

Equivalent faults: e.g., $P\ \text{a-s-0} \equiv L\ \text{a-s-0} \equiv Q\ \text{a-s-0}$
$Q\ \text{a-s-1} \equiv R\ \text{a-s-1} \equiv K\ \text{a-s-1}$
Capabilities and Complexity of D-Algorithm

Reconvergent fan-out
  Consider the s input s-a-0

Simple path sensitization does not allow us to propagate the fault to the primary output z

Worst-case complexity of D-algorithm is exponential in circuit size
  Must consider all path combinations
  XOR gates cause the behavior to approach the worst case
  Average case is much better; quadratic

PODEM: Path-oriented decision making
  Developed by Goel in 1981
  Also exponential, but in the number of circuit inputs, not its size

PODEM solves the problem by setting y to 0
9.3 Boolean Difference Methods

\[ K = f(A, B, C) = AB \lor BC \lor CA \]

\[ dK/dB = f(A, 0, C) \oplus f(A, 1, C) \]
\[ = CA \oplus (A \lor C) \]
\[ = A \oplus C \]

\[ K = PC \lor AB \]

\[ dK/dP = AB \oplus (C \lor AB) = C(\overline{AB}) \]

Tests that detect \( P \) s-a-0 are solutions to the equation \( P \frac{dK}{dP} = 1 \)
\[(A \oplus B) C(\overline{AB}) = 1 \quad \Rightarrow \quad C = 1, \ A \neq B \]

Tests that detect \( P \) s-a-1 are solutions to the equation \( \overline{P} \frac{dK}{dP} = 1 \)
\[(A \oplus B) C(\overline{AB}) = 1 \quad \Rightarrow \quad C = 1, \ A = B = 0 \]
9.4 The Complexity of Fault Testing

The satisfiability problem (SAT)
Decision problem: Is a Boolean expression satisfiable? (i.e., can we assign values to the variables to make the result 1?)

Theorem (Cook, 1971): SAT is NP-complete
In fact, even restricted versions of SAT remain NP-complete

Theorem (Cook, 1971): 3SAT is NP-complete
In 3SAT, the logic expression is a product of 3-term OR clauses

According to the Boolean difference formulation, fault detection can be converted to SAT (find the solutions to $P \frac{dK/dP}{dP} = 1$)

To prove the NP-completeness of fault detection, we need to show that SAT (or another NP-complete problem) can be converted to it

Proof of NP-completeness is due to Ibarra and Sahni [Ibar75]
A simple alternate proof by Fujiwara [Fuji82] is in the textbook
Proof that Fault Detection is NP-Complete

Theorem (Cook, 1971): 3SAT is NP-complete

Theorem: Clause-monotone SAT (CM-SAT) is NP-complete

CM-SAT has OR clauses each of which consists entirely of complemented or uncomplemented variables, but not both

3SAT can be converted to CM-SAT by replacing each mixed OR clause with the product of two clauses involving a new variable

Example: \((x_i \lor x_j \lor x'_k)\) is replaced by \((x_i \lor x_j \lor v_k)(v'_k \lor x'_k)\)

Clause-monotone SAT can be converted to fault detection in a circuit

First level has ANDs for all clauses with complemented variables

Second level has ORs for all clauses with uncomplemented variables, plus an OR gate with level-1 outputs as its inputs (one input to this gate is \(y\))

Third level has one AND gate that receives all level-2 outputs as its inputs

A test for \(y\) s-a-1 satisfies the original clause-monotone expression
9.5 Testing of Units with Memory

The presence of memory expands the number of required test cases.

To test a sequential machine, we may need to apply different input sequences for each possible initial state.

- Exponentially many possible input sequences
- Exponentially many possible machine states
Testing of Memory

Simple-minded approach: Write 000 ... 00 and 111 ... 11 into every memory word and read out to verify proper storage and retrieval

Problems with the simple-minded approach:

- Does not test access/decoding mechanism – How do you know the intended word was written into and read from?
- Many memory faults are pattern-sensitive, where cell operation is affected by the values stored in nearby cells
- Modern high-density memories experience dynamic faults that are exposed only for specific access sequences

Memory testing continues to be an active research area

Built-in self test is the only viable approach in the long term

Challenge: Any run time testing consumes some memory bandwidth
9.6 Off-Line vs. Concurrent Testing

This section will be forthcoming.
10  Fault Masking
### STRUCTURE AT A GLANCE

| Part I — Introduction: Dependable Systems (The Ideal-System View) | Goals | 1. Background and Motivation  
2. Dependability Attributes  
3. Combinational Modeling  
4. State-Space Modeling |
|---|---|---|
| Models | 5. Defect Avoidance  
6. Defect Circumvention  
7. Shielding and Hardening  
8. Yield Enhancement |
10. Fault Masking  
11. Design for Testability  
12. Replication and Voting |
| Examples | 13. Error Detection  
14. Error Correction  
15. Self-Checking Modules  
16. Redundant Disk Arrays |
| Part III — Faults: Logical Deviations (The Circuit-Level View) | Methods | 17. Malfunction Diagnosis  
18. Malfunction Tolerance  
19. Standby Redundancy  
20. Resilient Algorithms |
| Examples | 21. Degradation Allowance  
22. Degradation Management  
23. Robust Task Scheduling  
24. Software Redundancy |
| Part IV — Errors: Informational Distortions (The State-Level View) | Methods | 25. Failure Confinement  
26. Failure Recovery  
27. Agreement and Adjudication  
28. Fail-Safe System Design |
| Examples | 29. Error Detection  
30. Error Correction  
31. Self-Checking Modules  
32. Redundant Disk Arrays |
| Part V — Malfunctions: Architectural Anomalies (The Structure-Level View) | Methods | 33. Malfunction Diagnosis  
34. Malfunction Tolerance  
35. Standby Redundancy  
36. Resilient Algorithms |
| Examples | 37. Degradation Allowance  
38. Degradation Management  
39. Robust Task Scheduling  
40. Software Redundancy |
| Part VI — Degradations: Behavioral Lapses (The Service-Level View) | Methods | 41. Failure Confinement  
42. Failure Recovery  
43. Agreement and Adjudication  
44. Fail-Safe System Design |
| Examples | 45. Error Detection  
46. Error Correction  
47. Self-Checking Modules  
48. Redundant Disk Arrays |
| Part VII — Failures: Computational Breaches (The Result-Level View) | Methods | 49. Malfunction Diagnosis  
50. Malfunction Tolerance  
51. Standby Redundancy  
52. Resilient Algorithms |
| Examples | 53. Degradation Allowance  
54. Degradation Management  
55. Robust Task Scheduling  
56. Software Redundancy |

Appendix: Past, Present, and Future
10.1 Fault Avoidance vs. Masking

Fault

Avoid
- Prevent
  - Quality Assurance
  - Testing
  - Prevent
    - Detect
      - Test
        - Repair
          - Yes
            - Full?
              - Yes: Perfect
              - No: Fixed
          - No: Injured
        - Screened
          - Faulty
          - Faulty-safe
          - Degraded
          - Restored
          - Unaffected
  - Discard
    - Yes: Injured
    - No: Screened

Mask
- Remove
  - Dynamic Redundancy
  - Static Redundancy
  - Miss
    - Monitor
      - Detect
        - Reconfigure
          - Yes
            - Full?
              - Yes: Restored
              - No: Unaffected
10.2 Interwoven Redundant Logic

Even nonredundant circuits have some masking capability

Is there a way to exploit the inherent masking capabilities of logic gates to achieve general fault masking?
How Interwoven Logic Works

Let $x_1$, $x_2$, $x_3$, and $x_4$ be 4 copies of the signal $x$

1 $\rightarrow$ 0 change is critical for AND, subcritical for OR

0 $\rightarrow$ 1 change is critical for OR, subcritical for AND

Alternating layers of ANDs and ORs can mask each other’s critical faults

To mask $h$ critical faults:
Number of gates multiplied by $(h + 1)^2$
Gate inputs multiplied by $h + 1$

For $h = 1$, the scheme is known as Quadded logic
Interwoven Logic for Nanoelectronics

Half-adder implemented in quadded logic


IEEE D&T
July-Aug. 2005
pp. 328-339
Highly Reliable Logic with “Crummy” Relays

Moore & Shannon, 1956

\(a\): prob [contact made | energized]
1 – \(a\): prob [contact open | energized]
\(c\): prob [contact made | not energized]
1 – \(c\): prob [contact open | not energized]

No matter how crummy the relays (i.e., how close the values of \(a\) and \(c\)), one can interconnect many of them in a redundant series-parallel structure to achieve arbitrarily high reliability

\[
\text{prob [connection made | energized]} = 2a^2 - a^4 \quad (> a \text{ if } a > 0.62)
\]

\[
\text{prob [connection made | not energized]} = 2c^2 - c^4 \quad (\text{always } < c)
\]
10.3 Static Redundancy with Replication

TMR: \[ R = 3R_m^2 - 2R_m^3 \geq R_m \]

Condition on the module reliability:
\[ R = R_m [1 + (1 - R_m)(2R_m - 1)] \]
\[ (1 - R_m)(2R_m - 1) > 0 \implies R_m > 1/2 \]

\[ \text{RIF}_{TMR/\text{Simplex}} = \frac{(1 - R_m)/(1 - R)}{1 - R_m(2R_m - 1)} \]

\[ \text{MTTF: TMR} \frac{5}{6\lambda} \]
\[ \text{Simplex} \frac{1}{\lambda} \]
A TMR Application and Its Bit-Voting Unit

Single-event upset (SEU) = Soft error
Change of state caused by a high-energy particle strike

SEU effect on DRAMs
(from SANYO website)

TMR flip-flop for SEU tolerance
Example: SEU Hardened Flip-Flop

For list of flip-flop hardening methods and their comparison, see:  
http://klabs.org/richcontent/fpga_content/pages/notes/seu_hardening.htm
**N-Modular Redundancy (NMR)**

Triple-modular redundancy (TMR) can be generalized to \( N \) units

\( N \)-modular redundancy (NMR) uses \( N \) modules along with a voter, with \( N \) usually being odd

Example: 5MR
Operates correctly as long as
3 of the 5 modules are healthy

Voter complexity rises rapidly
with increasing \( N \)

Even values of \( N \) are also feasible

Example: 4MR, with 3-out-of-4 voting
Voter masks single faults; can be designed to detect double faults
10.4 Dynamic and Hybrid Redundancy

1. **Detect and replace**
   Dynamic redundancy (cold/hot standby)
   Detection via
   -- coding, watchdog timer, self-checking
   -- duplication (pair-and-spare)

2. **Mask in place**
   Static redundancy
   May revert to simplex instead of duplex
   Design challenges include
   -- synchronization for voting
   -- voting on imprecise results

3. **Mask, diagnose, and reconfigure**
   Hybrid redundancy
   Fault masked at output, but diagnosed
   -- e.g., via comparison with voter output
   Faulty circuit is replaced by spare
   Becomes static upon spare exhaustion
Comparing Replication Schemes

**Advantages**
- Less power (cold standby)
- Long life (just add spares)
- Immediate masking
- High safety
- Immediate masking
- Long life and high safety

**Drawbacks**
- Coverage factor
- Tolerance latency
- Power/area penalty
- Voting critical
- Power/area penalty
- Switch-voting critical

![Diagram of replication schemes](attachment:image.png)
Switch for Standby Redundancy

Standby redundancy requires an \( n \)-to-1 switch to select the output of the currently active module

The detectors use various info to deduce fault conditions
-- Error coding
-- Reasonableness checks
-- Watchdog timer

Once a fault has been detected, the switch reconfigures the system by flagging the faulty unit and activating next spare in sequence

If we use an \( n \)-to-2 switch and compare the two selected outputs, the configuration is known as “pair-and-spare.”
Fault Detection in Standby Redundancy

Activity monitoring

Duplication and comparison

Self-checking design
Preview of Self-Checking Design

Covered in Chapter 15

Function unit designed such that internal faults manifest themselves as invalid outputs.

Can remove this checker if we do not expect both units to fail and Function unit 2 translates any noncodeword input into noncode output.

Output of multiple checkers may be combined in self-checking manner.
Switch for Hybrid Redundancy

Hybrid redundancy with $n$ active and $s$ spare modules requires an $(n + s)$-to-$n$ switch to select the outputs of the active modules.

Self-purging redundancy is a variant of hybrid redundancy in which all modules are active at the outset, but they are purged as they disagree with the majority output.

Voting unit in self-purging redundancy is a threshold voter that considers the inputs with weights of 1 (active) or 0 (purged).
10.5 Time Redundancy

Retry upon a detected fault: particularly useful for transient faults

Recomputation not useful with permanent faults

Can make recomputation work by slightly changing the operands, but this is not always applicable

\[
\text{Compute } a \times (2b) \text{ instead of } (2a) \times b
\]

Compute \( b + a \) or \(-(-a - b)\) instead of \( a + b \)
10.6 Variations and Complications

Static redundancy makes fault testing more challenging

For static redundancy to be effective, we must ensure that initially all redundant components are fault-free.
Applications of NMR and Hybrid Redundancy

NASA’s Space Shuttle (retired in 2012):
Used 5-way redundancy in hardware
  Originally, 3 operational units + 2 spares
    (one warm, one cold)
  More recently, 4 operational + 1 spare
Also, uses 2 independently developed software systems (Design diversity)

Japanese Shinkansen “Bullet” Train
Triple-duplex system (6-fold redundancy)
11 Design for Testability
"Someone in this house flunked his earth science test because someone else in this house told him that love makes the world go around!"

"Algebra class will be important to you later in life because there's going to be a test six weeks from now."

Savage Chickens

by Doug Savage

ARE MULTIPLE CHOICE EXAMS AN ACCURATE MEASURE OF ONE'S KNOWLEDGE?

A. YES
B. A AND C
C. A AND B
D. ALL OF THE ABOVE

\[ 4x \div y^2 \\
xyz = 4^3 \\
x = ? \]
11.1 The Importance of Testability

A small circuit with a limited number of inputs and outputs can be tested with a reasonable amount of effort and time.

A complex unit, such as a microprocessor, cannot be tested solely based on its input/output behavior.

Hence, the need for provisions in the design to facilitate testing.
11.2 Testability Modeling

To allow detection of a fault in point A of a logic circuit, we need to:

- Be able to control that point from the primary inputs
- Be able to observe that point from the primary outputs

Thus, good testability requires good controllability and good observability for every node in the circuit.
Quantifying Controllability

Controllability $C$ of a line has a value between 0 and 1.

Derive $C$ values by proceeding from inputs ($C = 1$) to outputs.

Controllability transfer factor

$$CTF = 1 - \left| \frac{N(0) - N(1)}{N(0) + N(1)} \right|$$

$$C_{output} = (\sum_i C_{input_i} / k) \times CTF$$

$f$-way fan-out

A line with very low controllability is a good test point candidate.

$N(0)$: # input patterns leading to 1 output
$N(1)$: # input patterns leading to 0 output
Quantifying Observability

Observability $O$ of a line has a value between 0 and 1.

Derive $O$ values by proceeding from outputs ($O = 1$) to inputs.

Observability transfer factor

$$ OTF = \frac{N(sp)}{N(sp) + N(ip)} $$

$O_{input} = O_{output} \times OTF$

$f$-way fan-out

$$ 1 - \Pi_j (1 - O_j) $$

A line with very low observability is a good test point candidate.

$N(sp)$: # ways of sensitizing a path to output

$N(ip)$: # ways of inhibiting a path to output
Quantifying Testability

Testability = Controllability × Observability

Controllabilities

| Value |)
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>0.3</td>
</tr>
<tr>
<td>0.5</td>
</tr>
</tbody>
</table>

Observabilities

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.15</td>
</tr>
<tr>
<td>0.15</td>
</tr>
<tr>
<td>0.15</td>
</tr>
<tr>
<td>0.6</td>
</tr>
</tbody>
</table>

Testabilities

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.15</td>
</tr>
<tr>
<td>0.045</td>
</tr>
<tr>
<td>0.075</td>
</tr>
<tr>
<td>0.09</td>
</tr>
</tbody>
</table>

Overall testability of a circuit = Average of line testabilities
11.3 Testpoint Insertion

Increase controllability and observability via the insertion of degating mechanisms and control points.

Design for dual-mode operation
- Normal mode
- Test mode

Partitioned design
- Muxes

A

B

 NORMAL MODE

A

B

TEST MODE FOR A

11.4 Sequential Scan Techniques

Increase controllability and observability via provision of mechanisms to set and observe internal flip-flops

Scan design
- Shift desired states into FF
- Shift out FF states to observe

Partial scan design:
Mitigates the excessive overhead of a full scan design
11.5 Boundary Scan Design

Allows us to apply arbitrary inputs to circuit parts whose inputs would otherwise not be externally accessible.

Boundary scan elements of multiple parts are cascaded together into a scan path.

Basic Boundary Scan Cell

11.6 Built-in Self-Test (BIST)

Test patterns may be generated (pseudo)randomly – e.g., via LFSRs
Decision may be based on compressed test results
12 Replication and Voting
"Let's try voting for the greater of the two evils this time and see what happens."

“Fire. Bad. Those in favour?”
STRUCTURE AT A GLANCE

Part I — Introduction:
Dependable Systems
(The Ideal-System View)
Goals
Models
1. Background and Motivation
2. Dependability Attributes
3. Combinational Modeling
4. State-Space Modeling

Part II — Defects:
Physical Imperfections
(The Device-Level View)
Methods
Examples
5. Defect Avoidance
6. Defect Circumvention
7. Shielding and Hardening
8. Yield Enhancement

Part III — Faults:
Logical Deviations
(The Circuit-Level View)
Methods
Examples
9. Fault Testing
10. Fault Masking
11. Design for Testability
12. Replication and Voting

Part IV — Errors:
Informational Distortions
(The State-Level View)
Methods
Examples
13. Error Detection
14. Error Correction
15. Self-Checking Modules
16. Redundant Disk Arrays

Part V — Malfunctions:
Architectural Anomalies
(The Structure-Level View)
Methods
Examples
17. Malfunction Diagnosis
18. Malfunction Tolerance
19. Standby Redundancy
20. Resilient Algorithms

Part VI — Degradations:
Behavioral Lapses
(The Service-Level View)
Methods
Examples
21. Degradation Allowance
22. Degradation Management
23. Robust Task Scheduling
24. Software Redundancy

Part VII — Failures:
Computational Breaches
(The Result-Level View)
Methods
Examples
25. Failure Confinement
26. Failure Recovery
27. Agreement and Adjudication
28. Fail-Safe System Design

Appendix: Past, Present, and Future

Oct. 2019
Part III – Faults: Logical Deviations
Slide 57
12.1 Hardware Redundancy Overview

**Data path methods:**
- Replication in space (costly)
  - Duplicate and compare
  - Triplicate and vote
  - Pair-and-spare
  - NMR/hybrid
- Replication in time (slow?)
  - Recompute and compare
  - Recompute and vote
  - Alternating logic
  - Recompute after shift
  - Recompute after swap
  - Replicate operand segments
- Mixed space-time replication
- Monitoring (imperfect coverage)
  - Watchdog timer
  - Activity monitor
- Low-redundancy coding
  - Parity prediction
  - Residue checking
  - Self-checking design

**Control unit methods:**
- Coding of control signals
- Control-flow watchdog
- Self-checking design

**Glue logic methods:**
- Self-checking design
12.2 Replication in Space

The following schemes have already been discussed in connection with fault masking:

- **Duplicate and compare**
  - Diagram showing two inputs (1 and 2) and a comparator (C) with an error signal.

- **Triplicate and vote**
  - Diagram showing three inputs (1, 2, and 3) feeding into a voting unit (V).

- **Pair-and-spare**
  - Diagram showing two comparators (1 and 1') with error signals and a switch (S) for selection.

- **NMR/Hybrid**
  - Diagram showing four inputs (1, 2, 3, and 4) with a switch-voting unit (S, V) for selection.
TMR with Imperfect Voting Unit

\[ R = R_v (3R_m^2 - 2R_m^3)^2 > R_m \]

Condition on the voting unit reliability
\[ R_v > 1 / [3R_m - 2R_m^2] \]

\[ dR_v^{\text{min}} / dR_m = (-3 + 4R_m) / (3R_m - 2R_m^2)^2 \]

Condition on the module reliability
\[ \frac{3 - \sqrt{9 - 8/R_v}}{4} < R_m < \frac{3 + \sqrt{9 - 8/R_v}}{4} \]

Example: \( R_v = 0.95 \) requires that \( 0.56 < R_m < 0.94 \)

When \( R_v = 1 - \varepsilon \) is close to 1, we have
\[ 1/R_v \approx 1 + \varepsilon \] and \((1 - 8\varepsilon)^{0.5} \approx 1 - 4\varepsilon\), leading to \( 0.5 + \varepsilon < R_m < 1 - \varepsilon \)
TMR with Compensating Faults

\[ R_m = 1 - p_0 - p_1 \quad (0\text{-} and \text{ 1-fault probabilities}) \]

\[ R = (3R_m^2 - 2R_m^3) + 6p_0p_1R_m \]

Example: \( R_m = 0.998, p_0 = p_1 = 0.001 \)

\[ R = \frac{0.999,984 + 0.000,006}{0.002 / 0.000,016} = 0.999,990 \]

\[ \text{RIF}_{\text{TMR/Simplex}} = \frac{0.002}{0.000,016} = 125 \]

\[ \text{RIF}_{\text{Compen/TMR}} = \frac{0.000,016}{0.000,010} = 1.6 \]
12.3 Replication in Time

Can be slow, but in many control applications, extra time is available

Interleaving of the primary and duplicate computations saves time

Computation flowgraph, and schedule with 2 adders
Recompute and Compare/Vote

Repeat computation and store the results for comparison or voting

Comparison or voting need not be done right away; primary result may be used in further computations, with the result subsequently validated, if appropriate

On a simultaneous multithreading architecture, multiple instruction streams may be interspersed

Some Cray machines take advantage of extensive hardware resources to execute instructions twice
12.4 Mixed Space/Time Replication

Instead of duplicating the computation with no hardware change (slow) or duplicating the entire hardware (costly), we can add some hardware to make the interleaved recomputations more efficient.

Recomputation with same hardware resources \((T = 5, \text{excluding compare time})\)

Consider the effect of including a second adder

Duplicate computation

Recomputation with the inclusion of an extra adder \((T = 3, \text{excluding compare time})\)
12.5 Switching and Voting Units

We begin with some simple voting unit designs:

If in the case of 3-way disagreement any of the inputs can be chosen, then a simple design is possible.

This design can be readily generalized to a larger number of inputs.

One can perform pseudo voting that yields the median of 3 analog signals (Dennis, N.G., *Microelectronics and Reliability*, Aug. 1974).

Median and mean voting are also possible with digital signals.
Implementing a Bit-Voting Unit

TMR bit-voting: \( y = x_1 x_2 \lor x_2 x_3 \lor x_3 x_1 \)  
(carry output of a single-bit full-adder)

What about 5MR, 7MR?

Gate-level design quickly explodes in size

Other designs are also possible
  Arithmetic: add the bits, compare to threshold
  Mux-based
  Selection-based (majority of bit values is their median)

3-out-of-5 voting unit built of 2-input gates  Two mux-based designs for a 3-out-of-5 bit-voting unit
Complexity of Different Bit-Voting Unit Designs

Cost of majority bit-voting units as a function of the number $n$ of inputs
Majority-Friendly Nanotechnologies

Certain new nanotechnologies offer efficient majority gates
Sometimes, the majority gate is even simpler than an ordinary gate

Can we use majority gates as building-blocks in realizing voters?
Voting at the Word Level

Using bit-by-bit voting may be dangerous

One might think that in this example, any of the module outputs could be correct, so that producing 1 0 at the output isn’t all that wrong

However, with bit-by-bit voting, the output may be different from all inputs

Design of bit- and word-voting networks discussed in:
12.6 Variations and Design Issues

**NMR/simplex**: Voting unit is replaced with a unit that can also detects disagreements.

When a faulty unit is detected, that unit and one other unit are removed from service.

This makes all votes unambiguous and also improves systems lifetime.

**Self-purging redundancy**: Modules purged when they disagree with the output and the threshold of the voting unit is adjusted accordingly (purged modules produce 0 outputs).
Alternating Logic: Basic Ideas

Transmission of data over unreliable wires or buses
Send data; store at receiving end
Send bitwise complement of data
Compare the two versions
Detects wires s-a-0 or s-a-1, as well as many transients

The dual of a Boolean function $f(x_1, x_2, \ldots, x_n)$ is another function $f_d(x_1, x_2, \ldots, x_n)$ such that $f_d(x_1', x_2', \ldots, x_n') = f'(x_1, x_2, \ldots, x_n)$

Fact: Obtain the dual of $f$ by exchanging AND and OR operators in its logical expression. For example, the dual of $f = ab \lor c$ is $f_d = (a \lor b)c$

Advantages of this approach compared with duplication include a smaller probability of common errors
Alternating Logic: Self-Dual Functions

A function $f$ is self-dual if $f(x_1, x_2, \ldots, x_n) = f_d(x_1, x_2, \ldots, x_n)$

For example, both the sum $a \oplus b \oplus c$ and carry $ab \lor bc \lor ca$ outputs of a full-adder are self-dual functions.

With a self-dual function $f$, the functions $f$ and $f_d$ in the diagram above can be computed by using the same circuit twice (time redundancy).

Many functions of practical interest are self-dual.

Examples (proofs left as exercise)
A $k$-bit binary adder, with $2k + 1$ inputs and $k + 1$ outputs, is self-dual.
So are 1’s-complement and 2’s-complement versions of such an adder.
Recomputing with Transformed Operands

Alternating logic is a special case of the following general scheme, with its encoding and decoding functions being bitwise complementation.

Recompute after shift
When \( f \) is binary addition, we can use shifts for encoding and decoding. Shifting causes the adder circuits to be exercised differently each time. Originally proposed for ALUs with bit-slice organization.

Recompute after swap
When \( f \) is binary addition, we can use swaps for encoding and decoding. Swap the two operands; e.g., compute \( b + a \) instead of \( a + b \). Swap upper and lower halves of the two operands (modified adder).
Time-Redundant, Segmented Addition

Instead of using a $k$-bit adder twice for error detection or 3 times for error correction, one can segment the operands into 2 or 3 parts and similarly segment the adder; perform replicated addition on operand segments and use comparison/voting to detect/correct error.

Sum computed in two cycles: The lower half in cycle 1, and the upper half in cycle 2

Various other segmentation schemes have been suggested

**Example:** 16-bit adder with 4-way segmentation and voting

Townsend, Abraham, and Swartzlander, 2003