

ECSE 429 Study guide

Francis Piché

October 7, 2019

Contents

1	License Information	3
I	Introduction	3
2	Idealized Tests	3
3	What Does Testing Do?	3
4	Software Development Processes	4
4.1	Specification Phase	4
4.2	Design Phase	5
4.3	Generic Software Process Models	5
5	Quality	7
5.1	Quality Assurance and Control	7
II	Testing and Verification Terminology	8
6	Validation vs Verification vs Testing	8
7	Testing vs Debugging	8
8	Cost of Testing	9
9	Faults, Failures and Errors	9
9.1	Errors	9
9.2	Faults	9
9.2.1	Fault Causes	9
9.3	Failures	10
9.4	Defects	10
10	The 7 Testing Principles	11
10.1	Bug Presence over Absence	11
10.2	Exhaustive Testing is Impossible	11
10.3	Defect cost is exponential	11
10.4	Defects Cluster Together	11
10.5	Software Becomes Resistant to Testing	12
10.6	Testing is Content Dependent	12
10.7	Absence of errors is a Fallacy	12
11	Four Types of Testing Activities	12
12	IQ, QQ, PQ	12

13 V Model	12
14 Levels of Testing	13
14.1 Unit Testing	13
14.2 Integration Testing	13
14.3 System Testing	14
14.4 Acceptance Testing	14
15 Software Aging	14
16 Hardware Faults and Fault Modeling	14
16.1 Fault Models	15
16.2 Stuck-Open and Short-Faults	15
16.3 Notation and Terminology	16
16.4 Test Tables and the XOR Method	16
16.5 Path-Sensitizing Method	17
16.6 Controllability and Observability	17
16.7 Fault Equivalence	18
16.7.1 Gate equivalence	18
16.7.2 Fault Collapsing	18
16.8 Classes of Stuck-At-Faults	19
16.9 Redundant and Untestable Faults	19
16.10 Fault Dominance	20
17 Fault Simulation and Detection	20
17.1 Serial and Parallel Fault Simulation	20
17.2 Multiple s-a-v Faults and Delay Test	21
17.3 Test Coverage	21
18 Software Testing Terminology Revisited	22
19 Mutation Testing	22
19.1 Equivalence Partitioning	23
19.2 Summary	24

1 License Information

These notes are curated from Professor Kataryzna Radecka lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.5 Canada License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/ca/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Part I

Introduction

2 Idealized Tests

There is no single testing method to cover all situations. This would be an ideal test, that is, one that:

- Detects any and all defects
- Pass all good devices

The challenges when trying to design idealized tests are that there is a very large number of defects to be tested, of different types, and not all defects are testable (due to the way the system interfaces are designed, dependence on external factors etc).

There are external dependencies to almost any program, whether that be the hardware, the OS the program is running in, applications yours depends on, interacts with, or libraries your program utilizes. An ideal test would have to cover cases of failure in ALL of these dependencies, which with modern systems is infeasible.

3 What Does Testing Do?

With increased reliance on software systems, we must be able to have confidence in mission-critical pieces of software that govern our lives. Improper or lack of testing can lead to massive losses.

Traditionally and in hardware, testing has the following roles:

- Determine if the device has faults
- Diagnose which faults occurred
- Determine and correct errors in design or testing

- Failure mode analysis: What could have caused the defects in the devices

More specifically for software, testing has 3 responsibilities:

1. Does the software satisfy the requirements
2. Does the software perform adequately
3. Does the software have defects?

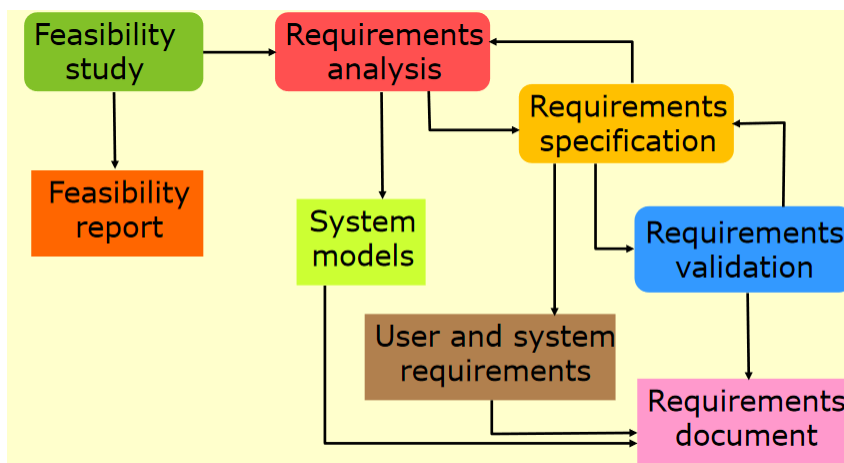
4 Software Development Processes

A software development process is a structured plan or set of activities to develop a software system. It has the task of specifying requirements, designing, validating and evolving the system further.

4.1 Specification Phase

The specification phase of software development is where we must establish what the system must be able to do, the constraints under which the system is to be built and operated. To determine this, there are several phases involved:

- Determine if the requirements are feasible
- Analyze requirements
- Formally specify requirements
- Validate requirement specification matches true requirements



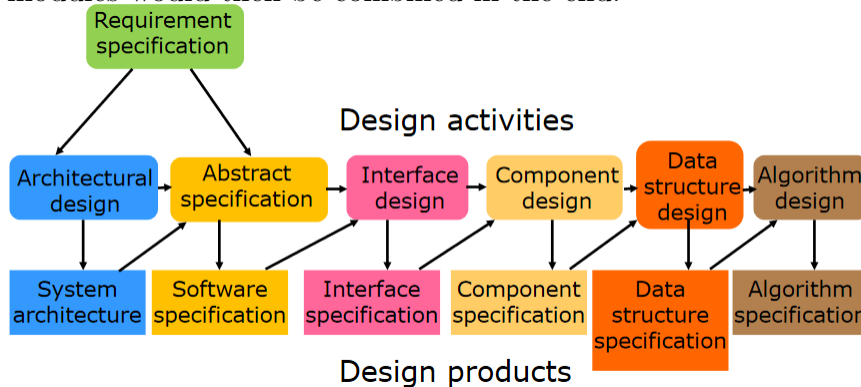
4.2 Design Phase

The design phase is where the system is built "in-theory" to a level of detail which proves it can match the requirements determined in the specification phase.

Phases:

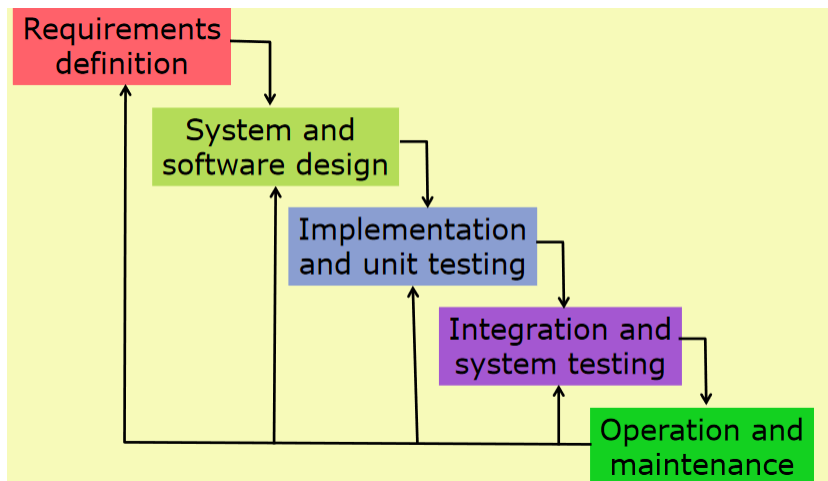
- Data design : How the data will be structured
- Architectural Design : How the structure of the application will be laid out
- Interface Design : How the components will communicate
- Procedural Design : Specify how the execution flow will work
- Algorithm Design : Low-level design of individual algorithms used

To accomplish this there are two common approaches: *refinement* in which the entire system is described in increasing levels of detail, and *Modularity* where the software is divided into conceptual (and functional) modules which can be developed independently. These modules would then be combined in the end.



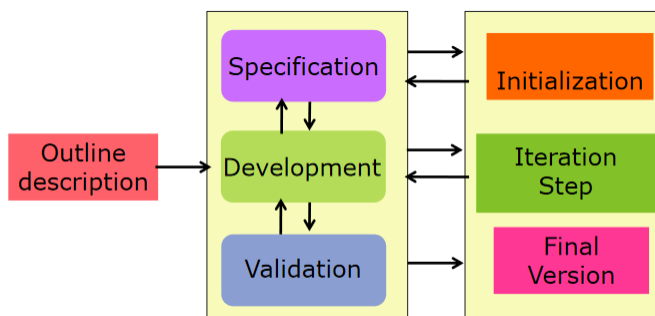
4.3 Generic Software Process Models

There are several generic models for designing software. The classic is the *Waterfall Model*. In this method, the phases are distinct and disjoint. First specify requirements, then implement, test etc. There is never any "backtracking" to previous phases.



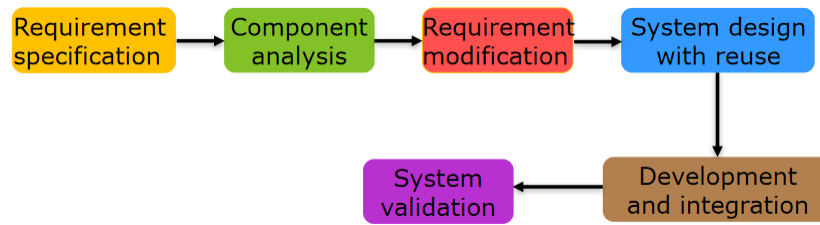
The advantage to this model is that if a design cannot be implemented for any reason, this will be detected and corrected far before time and money has gone into the implementation. This was especially useful for hardware, where manufacturing is also involved. The disadvantage is that it's very difficult to say a phase is "completed" and never return, as this requires full-knowledge of the requirements, limitations and changes to the requirements are limited or non-existent.

In the *evolutionary* model interleaves the phases and iteratively improves the software by adding more and more requirements.



The advantage to this is that problems can be addressed early, requirements are more flexible, and feedback from the stakeholders can be leveraged to make course-corrections. However, the disadvantage is that it can involve a lot of re-writing of "poorly-designed" components, and often it is difficult to make the early stages "generic" enough to be flexible for later changes. This strategy is best used for small to medium sized interactive systems, or for parts of a larger system.

In *component* based model the software is built by plugging several existing components together.



. Here the advantage is that often it can be assumed that the components built can be trusted, and so their inner-workings do not need to be tested. Instead, only interactions between components is to be tested. Using "off-the-shelf" solutions also saves development time, and (depending on the cost of the module) costs.

5 Quality

Software quality is the degree to which a system meets specific requirements, customer needs and expectations.

The environment in which the software is developed brings many challenges for quality assurance. The features requested, budgeting, timeline, customer relation, team dynamics, etc.

Software Quality Factors:

- Usability: How is the user experience?
- Maintainability: Find and fix bugs
- Flexibility: ability to add new features
- Testability
- Portability: adaptation to execution environments
- Reusability of components
- Interoperability with other components/systems
- Robustness, performance, understandability etc.

5.1 Quality Assurance and Control

Quality Assurance aims to plan and minimize cost of guaranteeing quality, while *quality control* is a set of activities to implement quality assurance principles.

The cost of independent contractors for QA is often high. However, forming QA teams within a company is often done to minimize cost and maximize effectiveness, since the team will form a deeper knowledge of your software. These teams aim to know what the software is doing, what it should be doing, and how to measure the difference. This is accomplished

through being involved with development teams, business teams, and using methods to measure software performance:

- Formal methods: mathematically verification
- Testing: explicit input/output to the software and check results
- Inspections: Human interaction to check requirements, design, code etc
- Metrics: data generated about the program to analyze quality

Part II

Testing and Verification Terminology

6 Validation vs Verification vs Testing

Validation is the process of evaluating software to ensure compliance with intended usage. "Did we build the *right* thing?"

Verification is the process of determining if products of a given phase of the software development cycle fulfills the requirements of the previous phase. "Did we build the thing *correctly*?"

Validation is more than just testing. It must demonstrate reliability, ensure compliance, generate knowledge about the application, and establish future requirements.

There are several techniques of validation:

- Formal Methods: Math based, often not possible
- Fault Injection: Intentional, methodical injection of faults to see how the system react and verify it matches the expected behavior.
- Dependency Analysis: Identification of hazards and proposing methods of how to get around them.

7 Testing vs Debugging

Testing is finding the existence of bugs, failures and defects. This job is handled by a QA team.

Debugging is finding the cause, analyzing and correcting the bugs. Done mostly by development team.

8 Cost of Testing

At least half of development budget is spent on testing. In reality, testing is a post-design activity, which increases cost since errors are not caught early during the design phase.

Not 2

9 Faults, Failures and Errors

Often it is not possible to establish that software is correct under all conditions due to application complexity. Therefore we must rely on showing that software fails under more specific condition. The problem then lies in which conditions we simulate with the system to target the majority of all possible cases.

9.1 Errors

An **error** is defined as an incorrect internal state that is the manifestation of some fault.

This fault must change the operation to be performed by a statement without changing the execution flow, OR, one that changed the execution flow.

However the software may still execute correctly even if it is Erroneous. (Redundant code, dead code etc.)

9.2 Faults

Faults are only design faults when talking about software. Whether it was mistyped or misunderstood, it is a fault. The same is not true for hardware where faults can be caused by outside influences.

A fault is *active* if it causes an error, otherwise it is *dormant*. *Fault activation* is when dormant fault becomes active through some change to the execution inputs or context.

9.2.1 Fault Causes

Faults can often be traced to a single root cause, whether that be software or hardware. This cause is not always logical errors in code or defects in hardware. The causes can come from:

- Failures in Communication
- Improper Resource management
- Inadequate Verification

- Bad Risk management
- Faulty Assumptions

9.3 Failures

An error which causes incorrect behavior with respect to the requirements specification. Software fails due to two things:

- Differ from specification
- Specification does not describe intended behavior.

Failure of some software component causes a **permanent fault** in software that uses that component. Failure of the whole software leads to a **permanent external fault**.

9.4 Defects

A defect is anything that produces wrong results causing failure. A single defect can cause a wide range of failures, but not all defects result in failure. Kind of like a fault.

So all in all, faults activate errors which propagate to failures. Failures are caused ultimately by faults.

There are 9 major causes of defects:

1. Faulty requirements definition
2. Client-developer communication failure
3. Deliberate deviation (in development or usage of system)
4. Design logic errors
5. Coding errors
6. Non-compliance with documentation of code
7. Improper testing
8. Documentation errors
9. Procedure errors (misusing a UI etc)

10 The 7 Testing Principles

10.1 Bug Presence over Absence

The purpose of testing is not to show the absence of bugs, but rather to show the presence of bugs.

If no bugs are found this does not mean that the software is correct.

Of course this doesn't mean the software is bad no matter what, its just that we did not stretch the software to its limit during testing.

10.2 Exhaustive Testing is Impossible

There is often very large number of possible inputs, outputs, states, execution paths for a piece of software, which makes it practically impossible to exhaustively test. Further, specifications may be ambiguous or subjective, leading to multiple "correct" implementations.

Clearly, if the function execution time is long, then it is infeasible to do many inputs.

Example:

```
for (int i=0; i < n; i++){  
    if (some cond){  
        do one thing  
    }else{  
        do another thing  
    }  
}
```

For each loop, there is 2 possible paths. So for $n=1$ there are 3 paths, $n=2$ 5 paths, and so on. So there are $2^n + 1$ possible paths which quickly grows impossible to exhaustively test.

10.3 Defect cost is exponential

The later defects are caught and fixed, the more they cost. It turns out this is generally exponential.

10.4 Defects Cluster Together

Generally defects are not distributed evenly across the application. This means that the tester can focus on specific key areas.

10.5 Software Becomes Resistant to Testing

After several test iterations the software will be revised to make the tests pass. Thus the tests must always also be revised and improved to try and find more bugs.

10.6 Testing is Content Dependent

Different systems are tested differently. This means that something mission-critical like a piece of military software must be more rigorously tested than something like a cat picture web scraper.

10.7 Absence of errors is a Fallacy

Finding and fixing defects will not make your application "good" if its unusable or doesn't fulfil the users requirements/expectations, it's still bad software.

11 Four Types of Testing Activities

- Test Design: Select appropriate test method based on characteristics of the software
- Test Automation: Tools for making testing feasible and repeatable
- Test Execution: Run the test cases
- Test Evaluation: Check test coverage criteria to estimate the quality of the test and reliability of the results.

12 IQ, QQ, PQ

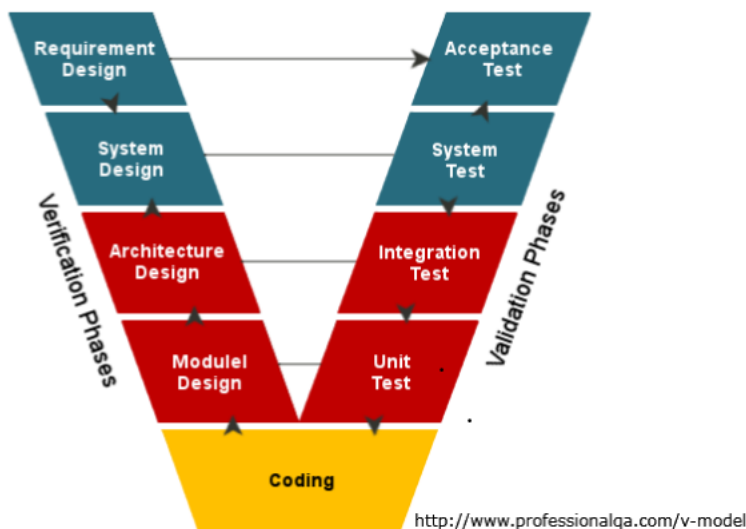
TODO

- IQ: Installation Qualification
- OQ: Operational Qualification
- PQ: Performance Qualification

13 V Model

The V-model is is a verification and validation model that extends to the waterfall design model.

It matches each design phase with a verification/validation counterpart.



V Model is best implemented when there are no ambiguous or undefined requirements, the project is short, and the product definition is static.

The advantages of this model is that testing starts early, testers can be prepared with test cases by the time development is done. Testing is done at each stage so more opportunity to refine the product.

The disadvantages to this model is that it lacks flexibility when it comes to late modifications. There is also no "early prototype" of the product.

14 Levels of Testing

14.1 Unit Testing

Unit testing attempts to test a single component of the system in isolation. This may cover functional and non-functional aspects. Often these are very detailed and low-level. Tests the design, code, data model and component specs.

Often approached with Test Driven Development (TDD) or white-box coding testing scenarios.

14.2 Integration Testing

Component Integration testing tests the interactions and interfaces between integrated components. Typically also automated.

System Integration testing tests interactions between packages, subsystems, microservices and external services. Often this is done after or in parallel with system testing.

This tends to catch incorrect message structure, timing issues, interface mismatches, communication failures, and incorrect assumptions about the boundaries and units of the system.

14.3 System Testing

Typically this is carried out by humans. The goal is to verify the completeness and usability of the system. This is done through end-to-end testing, exploratory testing and behavior-driven-development. It tends to catch incorrect calculation, failures in end-to-end functional tasks, unexpected behaviors, discrepancies with user manuals, failures to operate in production environment etc.

14.4 Acceptance Testing

Acceptance testing determines the applications fitness for use by end users. It is run in a real or simulated environment.

Generally formally documented procedure.

Operational acceptance testing is done by SysAdmins to ensure backup/restore, install / uninstall, disaster recovery, user management , performance and vulnerability checks are all up to standard.

Additionally, tests for contractual and regulatory restrictions may be tested (generally by some sort of audit group).

Alpha and beta testing may also be used to further increase confidence before final release.

15 Software Aging

Often software faults are the source or cause of software aging. Software aging is when a software gets worse and worse over time due to accumulation of bugs.

This may be caused by failure to keep up with changing demands, or can be caused by bad patches to the code introducing new faults.

16 Hardware Faults and Fault Modeling

An **Error** in hardware is improper behavior, ie wrong circuit output.

Defects in hardware is an unintended implementation problem. This includes material inconsistencies, environmental issues, designer mistakes, specification imprecisions etc.

A **fault** is an abstracted representation of defects. For example suppose we have the AND

gate: $c = a \wedge c$, if a is shorted to the ground (defect), then $c = 0, a = 1, b = 1$ (error). The fault is represented as **s-a-0**.

16.1 Fault Models

Fault modeling is the translation of physical defects into a mathematical construct that can be used by software simulation.

This is used to test rather than than real defects since real defects are too numerous and often not analyzable.

A fault model identifies targets for testing and enables analysis.

Logical fault modeling aims to model only faults which affect logical function or time-based performance. This shifts the scope of fault analysis from physical faults to software simulations.

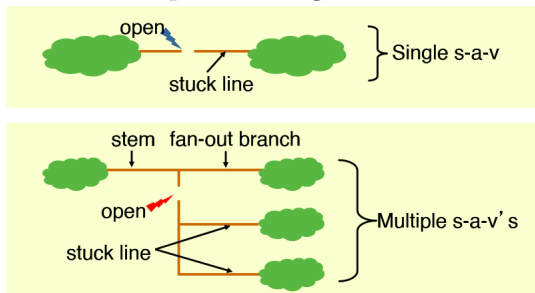
Solid or **permanent** faults do not change over time, while **non-permanent** faults occur and random moments for unspecified periods of time. Detecting non-permanent faults is difficult. They may be **transient** (hard to detect and not-well defined ie: cosmic rays, dust, temperature) or **intermittent** (disappearing / reappearing).

A **non-logical** fault is one that doesn't change the logical function of the unit but affects other circuit parameters such as delays, voltage or current levels.

16.2 Stuck-Open and Short-Faults

Stuck-at faults assume that the components are correct (no-faults) but that the interconnections have faults. **Shorts** are formed by unintended connections, and **opens** are results of broken connections.

These correspond to logical faults known as **stuck-at value** faults (s-a-v) where $v \in \{0, 1\}$.



A single stuck-at-fault (s-a-v) is characterized by :

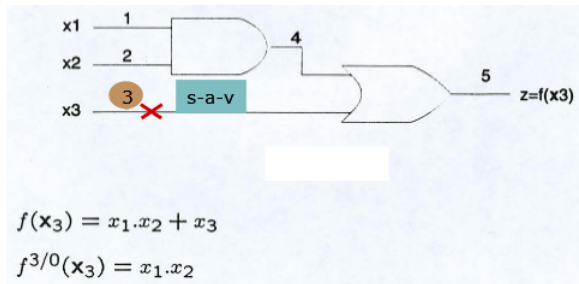
- One faulty line

- Line is set to 0 or 1 (permanently)
- Fault can be input or output of a gate.

Testing for s-a faults is a matter of supplying several input vectors collecting the output responses and comparing to the expected values.

16.3 Notation and Terminology

$f(x_n) = f(x_1, \dots, x_n)$ represents a fault-free circuit, while $f_d^p(x_n)$ is the same circuit with fault p/d where p is a wire, and d is an s-a-v fault (0 for s-a-0 and 1 for s-a-1).



A test, T_i detecting a fault p/d is an input x_n^j to the network producing a different response when the fault is present vs not present. If the response was the same regardless, then x_n^j is not a test.

So we can say that $f(x_n^j) \text{ XOR } f^{p/d}(x_n^j) = 1$ since they are always inverse of each other.

A **test set** is the set of all tests applied to a circuit during testing. The test set is **exhaustive** if it guarantee detection of all possible faults in the circuit. Note that exhaustive testing is nearly impossible: example: 32-bit adder would have 2^{32} different inputs possible to test. For this reason generally randomly generated test sets are used. Otherwise, **deterministic test patterns** can be generated by analyzing the circuit, and carefully crafting a test set to cover all possible cases. However even this does not guarantee detection of all possible faults.

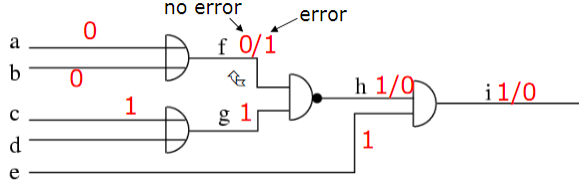
16.4 Test Tables and the XOR Method

A **fault table** displays the outputs of networks subject to test vectors. These are created by first generating the truth table for f and $f^{p/d}$ for each p, d . Next, to generate test cases for $f^{p/d}$, take input cases for which the f column and $f^{p/d}$ values are different.

An alternative method to generate test cases is the boolean difference, in which we express f and $f^{p/d}$ algebraically, and apply rules of Boolean algebra to find a test pattern.

16.5 Path-Sensitizing Method

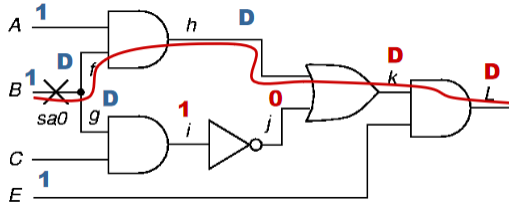
Another way of determining test vectors for a fault is the path-sensitizing method, in which the error is forward propagated to the output to determine the effects of the fault, and then backward propagation is used to determine the test vectors.



Here, to detect a s-a-1 fault at f , we first want to propagate the error to the output. Since this is an s-a-1 fault, if we set $a = b = 0$, then if $f = 1$ we will know for sure there is a fault. In order to see this, we must propagate to the output. g must be 1 so that h can be 1/0 depending on f , and we set e to 1 so that i will show whether the fault exists or not. To determine the test vector, we must back-propagate to determine which initial values lead to the output. We already know $a = b = 0$ and $e = 1$, and since $g = 1$ we can have $c = d = 1$.

Note that this is not unique. We could have set $c = 1, d = 0$ or visa versa.

There are cases when this method doesn't work. For example:



If we chose the sensitizing path forward with f, h, k, L , we cannot determine if the fault exists. Suppose it does not exist, then $B = C = E = 1$ and the output $L = 1$. But if the fault does exist then $h = 0, j = 1$ and the output is still 1. We say this is blocked at j since backpropagation cannot find a value for i for which the outcome will be different depending on the fault.

16.6 Controllability and Observability

Controllability is the ability to activate certain parts of a circuit ie: how easy is it to establish a specific signal at each internal node by setting inputs, and **observability** is to be able to see the outputs to determine correctness, ie: how easily can we determine internal values by controlling inputs and observing outputs.

Inputs closer to the beginning of a circuit are easier to control than those which are outputs of other circuit elements.

5- Valued logic :

1. $0 \Rightarrow 0$ in both true and faulty circuits

2. $1 \Rightarrow 1$ in both true and faulty circuits
3. $D \Rightarrow 1$ in a true circuit, 0 in faulty
4. $\bar{D} \Rightarrow 0$ in a true circuit, 1 in a faulty
5. $X \Rightarrow$ unknown for either true or faulty circuit

The goal is to find some inputs that cause D or \bar{D} at some output.

16.7 Fault Equivalence

Not all faults need to be tested if they are equivalent. Two faults are considered equivalent if all tests that detect one also detects the other.

Formal definition:

$$f_1(V) \equiv f_2(V) \Leftrightarrow f_1(V) XOR f_2(V) = 0$$

for all 2^n vectors V .

Equivalent faults also have identical test sets.

16.7.1 Gate equivalence

For a k input AND gate, there are k possible s-a-0 faults for the inputs and one for the output so $k + 1$ s-a-0 faults overall. Each s-a-0 changes the output to a constant 0 and so are all equivalent. Therefore there is really only 1 unique s-a-0 fault for the AND gate.

Similarly for an OR gate, s-a-1 faults are all equivalent.

In NOT gate, s-a-0 input is equivalent to s-a-1 output, and s-a-1 input is equivalent to s-a-0 output.

Similar logic applies for NAND and NOR gates.

16.7.2 Fault Collapsing

Fault collapsing is a method of determining all unique faults (non-equivalent) in a circuit.

First, all faults are partitioned in to equivalence classes, and a single fault from each is kept in a fault list.

$$\text{CollapseRatio} = |\text{collapsed faults}| / |\text{all faults}|$$

16.8 Classes of Stuck-At-Faults

- Potentially detectable faults: The test produces an unknown state, and whether or not the fault can be detected from this state is probabilistic. For example, a flip-flop that cannot be initialized due to a s-a-0 fault. These are really just an artifact from test simulation, not something that can occur in reality.
- Initialization Fault: The fault prevents initialization of the faulty circuit, and can be treated as a potentially detectable fault.
- Redundant Fault: No test exists for the fault because its effects are not visible.
- Untestable Fault: Test generator cannot find a test for it (test vector does not exist).

16.9 Redundant and Untestable Faults

Sometimes, circuits can have redundancies not detected by tests. for example, 3 NOT gates in a line with 2 s-a-v faults between the gates. This is testable, and will always produce distinct outputs for different inputs, but there is clearly 2 redundant NOT gates. This is an example of an *irredundant* circuit since all s-a-v faults can be removed by simplification.

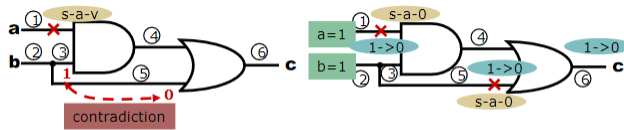
If the circuit contains an undetectable s-a-v fault it is redundant since it can always be simplified by removing a gate or gate input.

For example, if the input to an OR gate is s-a-0, then this is the same as just removing the gate and connecting the other wire directly to the next section in the circuit. There are actually several easy simplification rules:

Gate/Fault	Simplification rule
AND(NAND) input s-a-1	Remove input
AND(NAND) input s-a-0	Remove gate, replace by 0(1)
OR(NOR) input s-a-0	Remove Input
OR(NOR) input s-a-1	Remove gate, replace by 1(0)

Note that if a line i can be replaced with a constant $v = \{0, 1\}$ then fault i/v is untestable. (and redundant).

It is common that an untestable fault become testable (activated) with the presence of some other testable s-a-v fault. However if an untestable fault is activated, this may invalidate other tests!

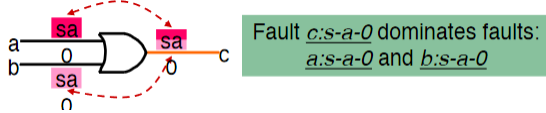


Notice here that the fault on line 1 is untestable, but if there is a s-a-0 on line 5, then the s-a-0 on line 1 becomes testable!

16.10 Fault Dominance

Let T_g be the set of all test vectors detecting a fault g . A fault f dominates $g \Leftrightarrow$ any test t that detects g also detects f . That is, $T_g \subseteq T_f$. Also note that if f_1 dominates f_2 and f_2 dominates f_1 then they are equivalent.

If f_2 dominates f_1 then f_2 can be removed from the fault list.



17 Fault Simulation and Detection

Fault simulations start with fault-free ones to establish correct behavior. Next, there are 4 key steps involved:

1. Fault Specification: The definition of the collapsed fault set
2. Fault Insertion: Select faults to be simulated in parallel, and their recognition by the simulation
3. Fault Effect Generation/Propagation: Create the input vector which will excite the fault.
4. Fault Detection and Discarding: remove the fault from the fault list once observation is complete

As for detecting faults, (Fault Detection), there are 3 major parts:

1. Fault List Generation: First we must find the collapsed list of faults (according to equivalence and dominance rules)
2. Test Patter Generation: Find the vectors detecting the faults
3. Fault simulation, run the fault simulator over the test vectors.

$$\text{Testability} = \frac{\text{detectable}}{\text{total}} \text{ faults}$$

17.1 Serial and Parallel Fault Simulation

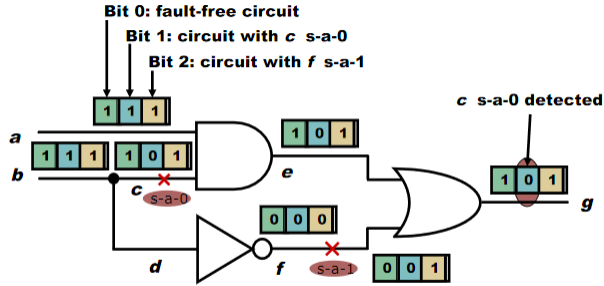
A simple algorithm for serially simulating faults is the following:

First simulate the fault free circuit to determine correct outputs. Next, for each fault in

the fault list, inject one fault and simulate the circuit. If the response differs from the expected, report the fault and suspend further testing.

However for most circuits, this will take much too long. The solution is to simulate multiple faults together at once.

In parallel fault simulation, we do the same thing but for $w - 1$ vectors at a time, where w is the word length. This is possible since Boolean operators are bit-wise independent. Meaning we can compute $10110001 \text{ AND } 10110001 = 11111111$ in one operation.



17.2 Multiple s-a-v Faults and Delay Test

Multiple s-a-v faults affect more than one circuit line with some combination of values.

The total number of single and multiple stuck-at-faults in a circuit with k single-fault sites is $3^k - 1$.

Tests for single s-a-v are not sufficient for detecting multiple s-a-v faults. However statistically, single s-a-v tests cover most multiple s-a-v faults.

Delay Testing aims to answer if a device works when running at speed. If a circuit passes a delay test then it produces correct outputs when supplied inputs and a specified timing. This is needed because there can be delay in an individual gate, or for time taken for a signal to traverse wires between gates.

17.3 Test Coverage

- Statement coverage: the percent of statements run
- Path coverage: the number of paths run
- Expression coverage: number of expressions within statements run

However 100% coverage does not mean the design is correct!

18 Software Testing Terminology Revisited

- Testing: finding inputs that cause software to fail
- Debugging: finding the faults that caused a given failure
- Software fault-tolerance: Ability to detect and recover from a fault. Often redundancy is used to guarantee that required services are provided in spite of faults.
- Test Case Values: the values that satisfy a test requirement
- Expected Results: the result that should be produced when executing the test
- Observability: Same as hardware, but note that software that interacts with hardware, databases and remote files tend to have low observability. Programs giving no outputs are hard to test for this reason, they are not observable.
- Controllability: Easy to control software with inputs, data abstraction tends to hinder controllability and observability

19 Mutation Testing

The idea behind mutation testing is to take a program and test data generated for it, and to create similar (but different) programs known as mutants which each differ from the original in a small way, each with a fault. The original test data is passed through each of the mutants. If the test data detects all differences, then the mutants are said to be dead, and the test set is adequate.

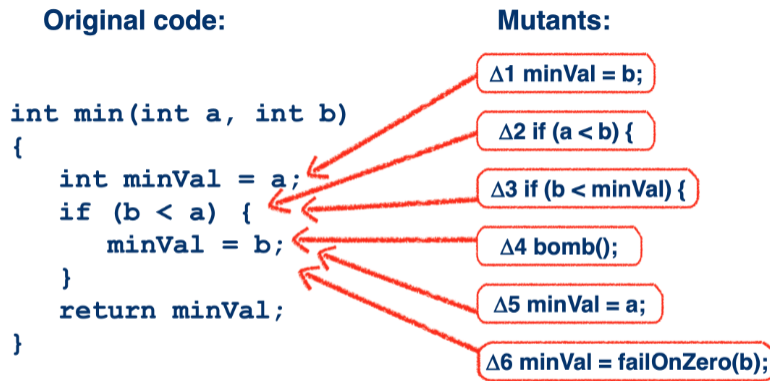
Failure to kill the mutant means that either it is equivalent to the original program or that the test set is inadequate.

There are a few types of mutants:

- Stillborn mutant: Syntactically incorrect, and so killed by the compiler
- Trivial mutant: Killed by most test cases
- Equivalent mutant: Always produces same output as original.

Ideally, we want to select mutation operators that are representative of all types of faults that might occur in practice. However there is still a very large number of them.

Mutation Coverage is the ratio of dead-mutants to all non-equivalent mutants. Each mutant can be seen as a test requirement, and the number of mutants depends on our definition of possible mutant operators for the syntax/structure of the software.



one mutant = one syntactic change introduced at a time

In general, errors are not that simple. We tend to assume that programmers are competent individuals and that errors are the result of a combination of simple errors. We also assume that by testing for simple errors, the tests are sensitive enough that we would also detect more complex errors.

19.1 Equivalence Partitioning

This is a form of black box testing that divides the inputs into classes of data from which tests can be derived. For example, if a program is computing $\sqrt{(X-1) * (X-2)}$ we might divide into $X < -2$, $-2 < X < 1$ and $x \geq 1$. And test cases would be chosen from each equivalence class.

Generally we assume that behavior under these classes is similar, and that programming errors tend to occur near extreme values (boundaries between classes). Thus we generally test values at their minimum, just above minimum, at a normal (nominal) value, just below the max and at the max. The same applies for output values, try to choose inputs that will generate values at the bounds of possible outputs.

Using this method, a function with n variables will need at least $4n + 1$ test cases. However this only works for variables that represent bounded physical quantities, as the actual meaning of the variables is not taken into account. We can also extend this to test for robustness by testing invalid domain values as well (total $6n + 1$ now).

We can also extend this to scenarios where more than one variable has an extreme value by taking the cartesian product of the bounds. Here there would be 5^n cases for n variables.

19.2 Summary

Mutation testing is most useful for white-box unit testing, and is also useful for assessing different test strategies. The mutation score can be used to predict fault detection effectiveness. However, this is very expensive to compute, and high coverage is not a guarantee of a fault-free software.