Software Testing Notes

Found at: benjaminshaw.uk

1 Terms

Black-Box

Examine the functionality of an application without looking at its inner workings.

White-Box

Testing the internal structures of an application

ITF: Independently Testable Features

Features that should be tested separately.

2 Functional Testing

Specification-based testing, black-box testing

A type of testing that focuses on *what* the system does. Test cases are derived on the specification of the component under scrutiny.

Functional testing does not look at methods of a module, rather it tests a *slice of the whole program's functionality*; verifying the program by checking it against the specification.

Functional testing is a *systematic* style of testing; trying to select inputs that are *valuable*.

Functional testing is *not random* because its not an effective method of finding issues with the software.

2.1 Partition Testing

If we have knowledge that some inputs are more likely to fail than others, we can utilise this to produce more effective testing.

The entirety of possible inputs may be sparse in errors overall, but some regions may be more dense than others.

These partitions are created using the specification. These partitions will be tested, alongside their *boundaries*, which tends to be an area rife with errors.

2.2 Creating Test Cases

1. Break the specification into *Independently Testable Features*
2. Identify *representative* values to be used as input, *e.g. correct values and malformed values*
3. Create *test specifications* that are a combination of these inputs

3 Combinatorial Testing

Where we identify some *attributes* of an environment, and produce *combinations* of these attributes to be tested.

3.1 Category-Partition Testing

Similar to functional testing, makes use of manual identification of values.

Parameters (inputs) and *environmental elements* are defined for each ITF - these values are referred to as *categories*.

We generate a number of variables for each of these categories.

Finally we define constraints to remove combinations of variables that would not make sense (*property constraints*), or a combination that we only need to test once, *i.e. one variable renders the meaning of the other variables useless*, known as *single constraints*.

The constraints allow us to reduce the number of test cases that we have, as an unwieldy test suite does not provide much benefit.

3.2 Pairwise Combinatorial Testing

Category partitioning is quite an exhaustive process, which pairwise combinations try to avoid.

Most bugs in software are a result of an erroneous single input. It is far less common to find a bug that involves an interaction between three or more parameters.

Pairwise covers all combinations of size 2. This produces an inherently smaller test suite than taking the entire Cartesian product.

3.3 Catalog Based Testing

This method of testing is based on experience made by an organisation over time in deriving valuable test suites.

4 Structural Testing

Specification-based testing, glass-box testing

Judging test suite thoroughness based on the structure of the program itself. Structural testing is still testing product functionality against its specification.

Specification-based testing complements and is often done after functional testing. It finds faults in parts of programs that are not executed by the functional-based testing suite.

Structural coverage increases confidence in thoroughness of testing but *it gives no guarantees* of finding all faults, since the state might not be corrupted when statement is executed with some values.

4.1 Statement Testing

Statement coverage: $\frac{\text{\# executed statements}}{\text{\# all statements}}$

Rationale: a fault in a statement can only be revealed by executing the faulty statement.

4.2 Condition Testing

Branch coverage exposes faults in how a computation is intuitively attractive but it also groups cases with the same outcome.

Condition coverage considers *individual conditions* in a compound boolean expression.

4.2.1 Basic condition testing

Each basic condition must be executed at least once.

Coverage: $\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$

Basic condition adequacy criterion can be satisfied without satisfying branch coverage.

Branch and basic condition testing *are not comparable*.

4.2.2 Compound condition testing

Exponential number of test cases

Cover all possible evaluations of compound conditions and all branches of a decision tree.

4.2.3 Modified condition/decision (MC/DC)

Linear number of test cases. N+1 test cases for N basic conditions.

Good balance of thoroughness and test size and therefore widely used.

Effectively test *important* combinations of conditions, where *important* means: each basic condition shown to independently affect the outcome of each decision. MC/DC is basic condition coverage, branch coverage with the addition of the

Tests required:

- For each basic condition C, two test cases
- Tests where values of all evaluated conditions except C are the same
- Compound condition of tests described above should evaluates to true for one value of C and false for the other

4.3 Path testing

Path testing focuses on combination of decisions and conditions along a path.

Path coverage: $\frac{\text{\# executed paths}}{\text{\# all paths}}$

Theoretically, the strongest coverage metric but generally impossible to achieve because of infeasible/infinite number of tests.

4.3.1 Boundary interior path adequacy

Partition the infinite set of paths into a finite number of classes. Group together paths that differ only in the subpath they follow when repeating the body of a loop. Follow each path in the control flow graph up to the first repeated node. The set of paths from the root of the tree to each leaf is the required set of subpaths for boundary/interior coverage.

Limitations:

- The number of paths can still grow exponentially. N non-loop branches results in 2N paths.
- Choosing input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent

4.3.2 Loop boundary adequacy

Variant of the boundary/interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths.

Criterion: A test suite satisfies the loop boundary adequacy criterion iff for every loop, in at least one test case, the loop body is iterated:

- zero times
- once
- more than once

4.3.3 Linear Code Sequence And Jumps(LCSAJ) dequacy

Often, we want to reason about the subpaths that execution can take. A subpath from one branch of control to another is called a LCSAJ.

We can require coverage of all sequences of LCSAJs of length N. The most basic metric is the proportion of statements executed, Test Effectiveness Ratio 1 (TER1).

TER_1 = Statement coverage

TER_2 = branch coverage

$TER_N = \frac{\text{\# LCSAJ of length N executed by test data}}{\text{\# total LCSAJs of length N}}$

4.3.4 Cyclomatic adequacy

Cyclomatic coverage counts the number of independent paths that have been exercised, relative to the cyclomatic complexity.

Cyclomatic number - number of independent paths in the CFG. A path is representable as a bit vector, where each component of the vector represents an edge.

For e = #edges, n = #nodes, c = #connected components of a graph the cyclomatic number is:

- e - n + c for an arbitrary graph
- e - n + 2 for a CFG

4.4 Procedure call testing

4.4.1 Entry and Exit Testing

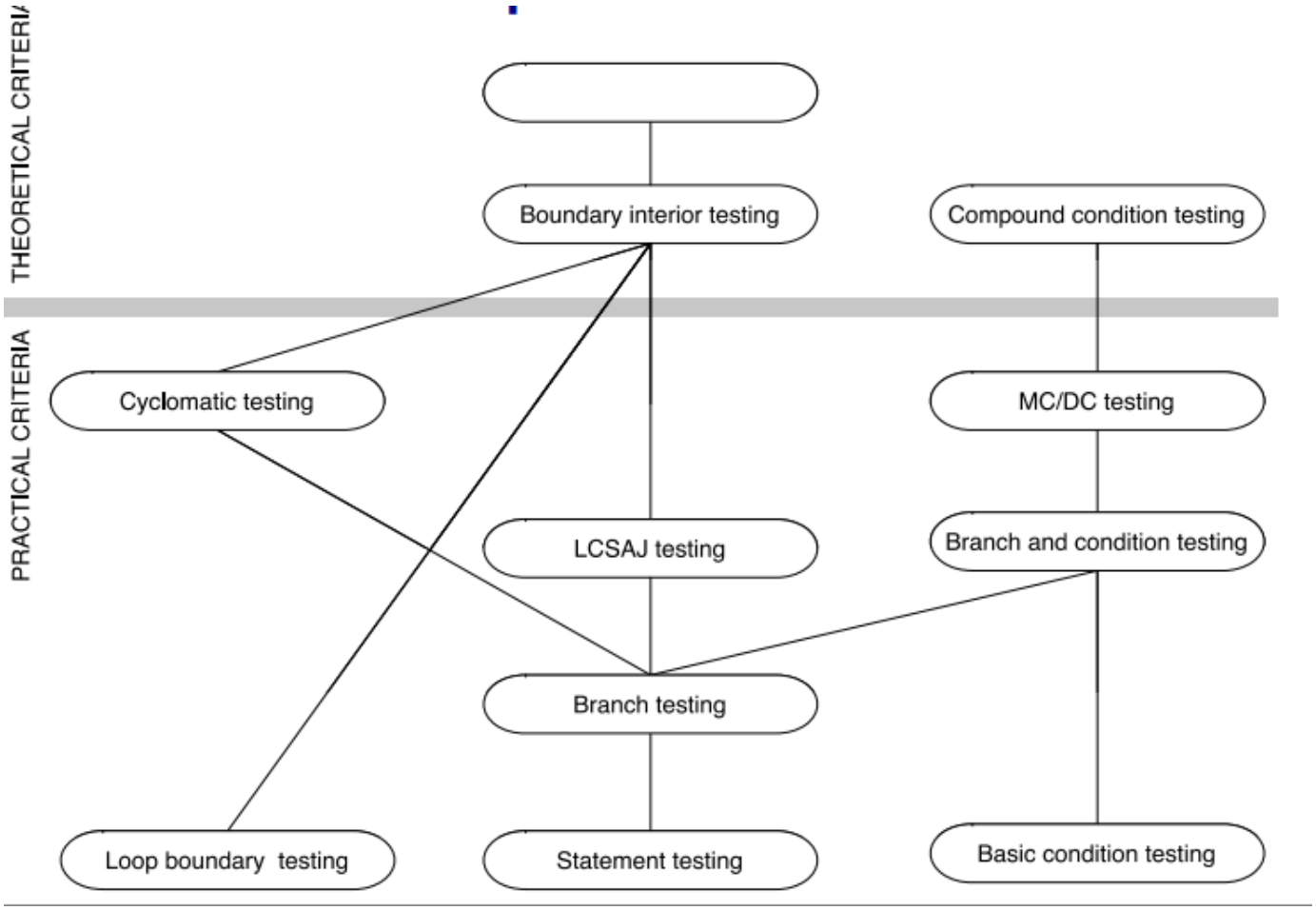
Write tests to ensure the procedure entry/exit points are entered and exited in the context they are intended to be used. Finds interface errors that statement coverage would not find.

4.4.2 Call Coverage

Call coverage requires that a test suite executes all possible method calls.

Challenging for OO systems, where a method call might be bound to different objects at runtime.

4.5 Subsumption relation



5 Dependence and Data Flow Models

5.1 Def-Use Pairs

A def-use pair associates a point in a program where a value is produced with a point where it is used.

Definition: where a variable gets a value

- Variable declaration
- Variable initialization
- Assignment
- Values received by a parameter

Use: extraction of a value from a variable

- Expressions
- Conditional statements
- Parameter passing
- Returns

A *definition-clear path* is a path along the CFG from a definition to a use of the same variable without another definition of the variable between.

A def-use pair is formed if and only if there is a definition-clear path between the definition and the use.

If, instead, another definition is present on the path, then the latter definition *kills* the former.

5.2 Data Dependence Graph

Nodes - same as in CFG

Edges - def-use pairs, labeled with the variable name

5.3 Data Flow Models

Data flow models detect patterns on CFGs:

- Nodes initiating the pattern
- Nodes terminating it
- Nodes that may interrupt it

Pros:

- Can be implemented by efficient iterative algorithms
- Widely applicable (not just for classic “data flow” properties)

Limitations:

- Unable to distinguish feasible from infeasible paths
- Analyses spanning whole programs (e.g., alias analysis) must trade off precision against computational cost

6 Data Flow Testing

Middle ground in structural testing, since node and edge coverage doesn't test interactions and path-based criteria requires impractical number of test cases.

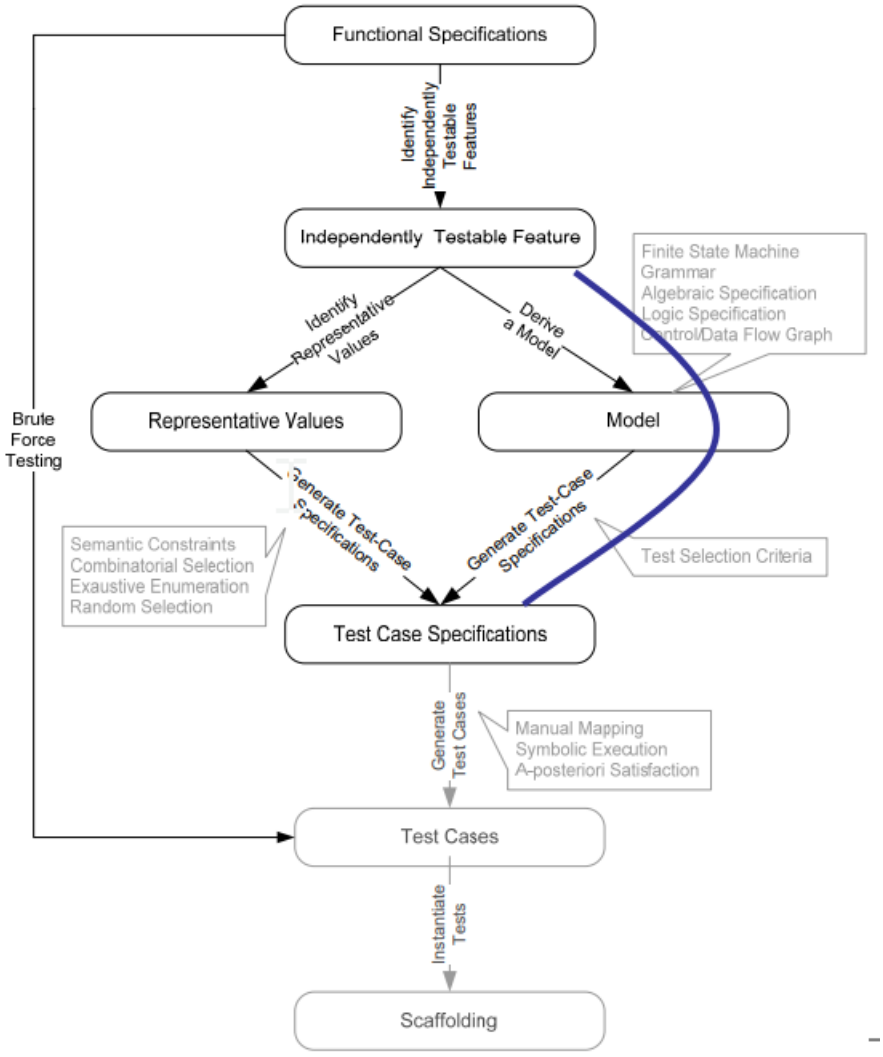
Criteria:

- Each DU pair is exercised by at least one test case
- Each simple(non looping) DU path is exercised by at least one test case
- For each definition, there is at least one test case which exercises a DU pair containing it

Limitations:

- Worst case is bad (undecidable properties, exponential blowup of paths) so pragmatic compromises are required

7 Model Based Testing



Models are useful abstractions. In specification and design, they help us think and communicate about complex artifacts emphasizing key features and suppressing details. They convey structure and help us focus on one thing at a time.

7.1 Deriving test cases from a FSM

A common kind of model for describing behavior that depends on sequences of events or stimuli. E.g. UML state diagrams

State coverage: Every state in the model should be visited by at least one test case

Transition coverage: Every transition between states should be traversed by at least one test case. This is the *most used criterion*

Base assumption: States fully summarize history.

So FSM testing is generally *not path sensitive* - no distinction based on how we reached a state; this should be true of well-designed state machine models. If assumption does not hold we use same criteria as in branch coverage (most commonly boundary-interior criteria).

7.2 Deriving test cases from decision structures

Some specifications are structured as decision tables, decision trees, or flow charts. We can exercise these as if they were program source code.

8 Testing OOP Software

Characteristics of OO Software:

- State dependent behavior
- Encapsulation
- Inheritance
- Polymorphism and dynamic binding
- Abstract and generic classes
- Exception handling

As with testing of other software, start with functional tests and follow with structural testing.

Unit in OO is unit a class or (small) cluster of strongly related classes (e.g. sets of Java classes).

Unit testing = intraclass testing

Integration testing = interclass testing

8.1 Intraclass Testing

8.1.1 State Machine Testing

Object methods are modeled as state transitions in a FSM. Test cases are sequences of method calls.

8.1.2 State Diagram Testing

If state diagram is provided in the specification, use that to produce test cases. Either flatten the diagrams provided into a standard FSM or use the diagram model directly.

8.2 Interclass Testing

First level of integration testing for OO software. Focuses on interactions between classes.

Approach:

- Generate class hierarchy from the class diagram. Sometimes stub generation and breaking of loops is necessary
- Start testing bottom-up
- Consider all combinations of interactions
- Select subset of these interactions(not all because of combinatorial explosion of cases). Could be random selection or significant interaction identified in the specification.

8.3 Polymorphism and dynamic binding

Combinatorial approach - test combination of subclasses. Same as with combinatorial testing, pairwise combinations can be used to reduce tests cases.

8.4 Inheritance

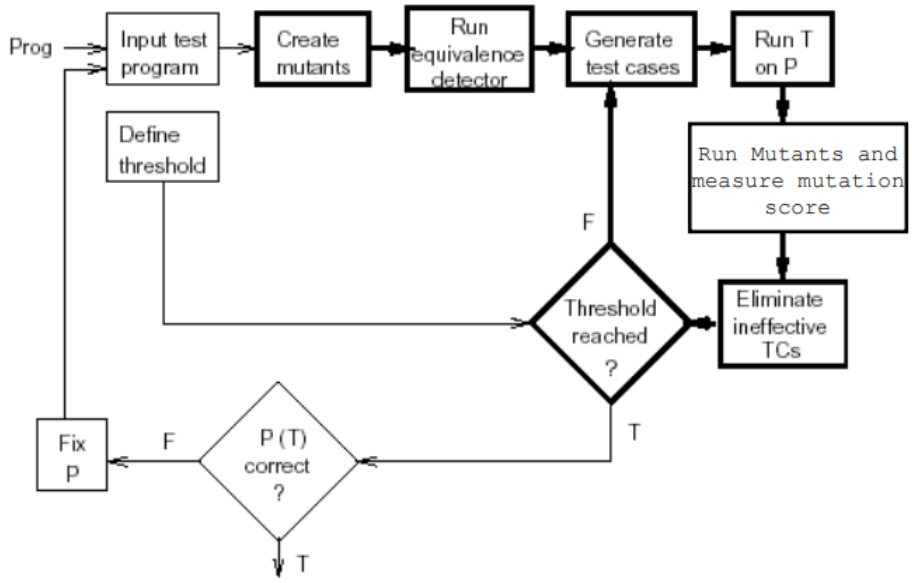
When testing a subclass track the overridden and newly introduced methods and only tests those.

8.5 Exception handling

Impractical to treat exceptions like normal flow so we separately test exceptions. We test each explicit throw or re-throw of an exception.

9 Mutation Testing

Fault-based Testing: directed towards "typical" faults that could occur in a program.



Basic idea:

- Generate class hierarchy from the class diagram. Sometimes stub generation and breaking of loops is necessary
- Generate test suite for a program.
- Create a number of similar programs by modifying the original program with small faults(mutants). This is usually done through an automated generation with specified modification rules.
- Run the original test suite and ensure all faults are being caught(mutants are killed) by your tests.
- If any of a mutant remains live and is not one that results in equivalent program to the original, test suite should be augmented.

9.1 Types of mutants

Stillborn mutants: Syntactically incorrect, killed by compiler, e.g., x = a ++ b

Trivial mutants: Killed by almost any test case

Equivalent mutants: Always acts in the same behavior as the original program, e.g., x = a + b and x = a – (-b)

None of these are useful for testing since we want mutants that change the behavior of the program.

10 Integration and Component-Based Testing

Integration testing focuses on the interface specification details. It tests the interaction and compatibility between object.

Integration testing may serve as a process check for unit testing:

- If module faults are revealed in integration testing, they signal inadequate unit testing .
- If integration faults occur in interfaces between correctly implemented modules the errors can be correctly implemented modules, the errors can be traced to module breakdown and interface specifications.

10.1 Integration faults

- Inconsistent interpretation of parameters or values. Example: Mixed units (meters/yards) in Martian Lander
- Violations of value domains, capacity, or size limits. Example: Buffer overflow
- Side effects on parameters or resources.Example: Conflict on (unspecified) temporary file
- Omitted or misunderstood functionality. Example: Inconsistent interpretation of web hits
- Nonfunctional properties. Example: Unanticipated performance issues.
- Dynamic mismatches. Example: Incompatible polymorphic method calls

10.2 Big Bang Integration Testing

Test only after integrating all modules. Does not require scaffolding but has a lot of drawbacks: high cost of repair, minimum observability, diagnosability, efficacy and feedback.

10.3 Structural orientation

Modules constructed, integrated and tested based on a hierarchical project structure.

10.3.1 Top down

Working from the top level (in terms of “use” or “include” relation) toward the bottom. No drivers required if program tested from top-level interface (e.g. GUI, CLI, web app, etc.). Write stubs of called or used modules at each step in construction. As modules replace stubs, more functionality is testable.

10.3.2 Bottom up

Starting at the leaves of the hierarchy, we never need stubs but we must construct drivers for each module. When an intermediate module replaces a driver, it needs its own driver. Several working subsystems, consisting of method and drivers, eventually get integrated into a single system.

10.3.3 Sandwich

Start testing from both ends. Both stubs and drivers needed. Sandwich integration is flexible and adaptable but complex to plan.

10.4 Functional orientation

Modules integrated according to application characteristics or features. Functional strategies require more planning and are more complex than structural strategies but provide better process visibility, especially in complex systems.

10.4.1 Threads

A “thread” is a portion of several modules that together provide a user-visible program feature. Integration is done one thread at a time. Minimizes stubs and drivers but is also complex to plan.

10.4.2 Critical Modules

Start integration form riskiest modules. Risk assessment is necessary first step. Designed to deliver any bad news as early as possible.

10.5 Component Testing

Components are reusable units of deployment and composition. They are often deployed and integrated multiple times and are characterized by an interface or contract. Usually integrated and tested by different teams.

10.5.1 Producer View Testing

Includes thorough functional testing based on application program interface (API). Includes thorough acceptance testing based on scenarios of expected use. Includes stress and capacity testing.

10.5.2 User View Testing

Not primarily to find faults in the component. More concerned with the question of if the component is suitable for this application.

11 Test Driven Development

Only relevant to black-box unit testing.

11.1 TDD cycle

- **Write Test Code:** Guarantees that every functional code is testable. Provides a specification for the functional code.Helps to think about design.Ensure the functional code is tangible
- **Write Functional Code:** Fulfill the requirement (test code). Write the simplest solution that works. Leave Improvements for a later step. The code written is only designed to pass the test
- **Refactor:** Clean-up the code (test and functional). Make sure the code expresses intent. Remove code smells. Re-think and design. Delete unnecessary code.

11.2 Advantages

- Shortens the programming feedback loop.
- Promotes the development of high-quality code
- User requirements more easily understood.
- Reduced interface misunderstandings
- Provides concrete evidence that your software works
- Reduced software defect rates, better code and less debug time

11.3 Disadvantages

- Programmers like to code, not to test
- Test writing is time consuming
- Test completeness is difficult to judge

12 Regression Testing

Changes in software that introduce software bugs and make features stop functioning as intended are called *regressions*. To avoid regressions old test should be run in addition to tests added for new additions in the software.

12.1 Re-test All

The test-all approach is good when you want to be certain that the new version works on all tests developed for the previous version. Becomes increasingly expensive as software grows. Bad if you have limited resources.

12.2 Re-test Test Suite Subset

12.2.1 Code-based selection

Only execute test cases that execute changed or new code.

12.2.2 Control-flow and data-flow selection

Same idea as code-based selection but it analyses changes in the control-flow and data-flow graphs. Can be automated.

12.2.3 Specification-based selection

Pick test cases that test new and changed functionality.

12.3 Program Dependence Graph

Program Dependence Graph (PDG) that captures control and data dependencies between nodes in CFG.

Automated analysis can be done against the PDG to prioritize tests to be run.

13 System & Acceptance Testing

13.1 System Testing

Tests for correctness and completion.

One strategy for maximizing independence: System (and acceptance) test performed by a different organization than the one doing the implementation.

Major focus of system testing is the opportunity to verify global properties against actual system specification. Global properties - performance, latency, reliability. Those properties can vary depending on the testing environment.

13.1.1 Stress testing

13.1.2 Capacity testing

13.1.3 Security testing

13.1.4 Performance testing

13.1.5 Compliance testing

13.1.6 Documentation testing

13.2 Acceptance Testing

Used to measure quality quantitatively.

13.2.1 System Reliability

13.2.2 Component Reliability Model

13.2.3 MTTF: Mean Time To Failure

13.2.4 Serial System Reliability

13.2.5 Parallel System Reliability