Boundary Value Testing

Chapter 2 - Boundary Value Testing: Generalizing Boundary Value Analysis, Limitations of BVA, Robustness Testing, Worst-case Testing, Special Value Testing, Test cases for Triangle Problem, Test cases for the NextDate function, Test cases for the Commission Problem, Random Testing and Guidelines for Boundary Value Testing.

Boundary Value Testing (BVT) is a specification based testing method that involves creating test cases based on the boundary values of input domains. Boundary values are the values at the edges of an input domain, just inside and just outside the boundaries, where the behavior of a system might change. This technique is based on the observation that errors tend to occur at the boundaries of input values rather than in the center.

Key Concepts of Boundary Value Testing:

Boundary Values: These are the values at both ends of input ranges. For example, if an input field accepts values from 1 to 100, the boundary values would be 0, 1, 2, 99, 100, and 101. Test Cases: Boundary Value Testing focuses on creating test cases for these boundary values rather than testing with any value within the range. This approach helps to efficiently detect errors that are related to incorrect handling of data at the edges.

Function Mapping: Just as a mathematical function maps inputs (domain) to outputs (range), a program takes specific inputs and generates outputs based on those inputs. Understanding this functional nature helps in designing effective test cases by considering the inputs and expected outputs.

Cross Products: When a program's inputs or outputs are combinations of different variables, these can be treated as cross products, which are sets formed by combining each possible value of one variable with each possible value of another.

Example

Boundary Value Testing

Suppose a function is designed to accept an integer value from 1 to 100 inclusive. Boundary Value Testing would generate test cases for values at and around the boundaries:

Just below the minimum boundary (e.g., 0)

At the minimum boundary (e.g., 1)

Just above the minimum boundary (e.g., 2)

Just below the maximum boundary (e.g., 99)

At the maximum boundary (e.g., 100)

Just above the maximum boundary (e.g., 101)

What is Input Domain Function?

An input domain function refers to the range of valid input values that can be accepted by a function or program. In the context of software testing, the input domain function is defined by the boundaries within which input variables must fall to ensure the correct functioning of the program.

Example: If we consider a function F that takes two variables x1 and x2, the input variables x1 and x2 are constrained by certain boundaries:

a<= x1<=b

c<= x2<=d

These boundaries [a, b] and [c, d] define the valid ranges for x1 and x2

What is Boundary Value Testing?

Boundary Value Testing (BVT) also known as Input domain testing is a specification-based testing technique that focuses on the inputs a program can accept. This type of testing is based on the idea that errors are most frequent at the edges of an input range, hence testing these boundary values can be more effective in finding bugs.

Importance of Boundary Value Testing?

Boundary value testing is a software testing technique that involves creating test cases based on the boundary values of input domains. This method is particularly useful and frequently employed because it effectively identifies errors that occur at the edges of input ranges, where bugs are most likely to appear,

1. High Error Detection Rate at Boundaries: Many errors in software occur at the boundaries of input ranges due to off-by-one errors and other boundary-related issues. Boundary value testing specifically targets these potentially problematic areas, which increases the likelihood of catching bugs that might not be detected by other testing methods that use values well within the range.
2. Efficiency: Boundary value testing is a cost-effective method in terms of the number of test cases generated versus the potential defects found. By focusing on the edge cases, it reduces the number of test cases needed compared to exhaustive testing, which would require much more time and resources.
3. Common Requirement Specifications: Requirements often define operations or behaviors at the limits of input ranges (e.g., "the age should be between 18 and 60"). Testing these boundary conditions directly checks the system's adherence to its specified requirements.
4. Usability and Reliability: By ensuring that the software behaves correctly at boundary values, developers can improve the usability and reliability of their software. This is because handling boundary conditions gracefully often reflects the software's ability to handle unexpected or extreme inputs, which are critical in real-world operations.
5. Early Defect Identification: Identifying defects at the boundaries early in the testing process can lead to more efficient debugging and resolution, reducing the likelihood of critical issues in later stages of development.

6. Integrates with Other Test Methods: This method can be effectively combined with other testing strategies such as equivalence partitioning (where inputs are divided into logically similar groups), further refining the efficiency and effectiveness of the testing process.

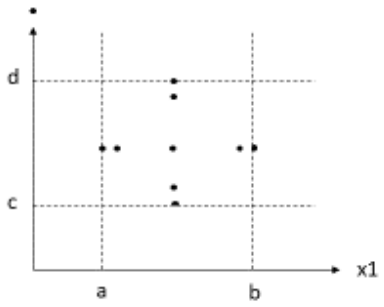
Types of Boundary Value Testing

Boundary value testing is a critical technique in software testing where special focus is placed on values at the edge of input domains. The four types of boundary value testing are:

1. Normal Boundary Value Testing: Normal Boundary Value Testing focuses on testing values at the boundaries within the valid range.
2. Robust Boundary Value Testing: Robust Boundary Value Testing extends Normal Boundary Value Testing by including values just outside the valid range. It tests the system's ability to handle inputs slightly beyond the expected boundaries.
3. Worst-Case Boundary Value Testing: Worst-Case Boundary Value Testing examines the effects of all combinations of boundary values across multiple variables. It explores interaction between variables at their boundary conditions.
4. Robust Worst-Case Boundary Value Testing: Robust Worst-Case Boundary Value Testing combines out-of-range values for multiple variables to stress test the system. It include extreme combinations, even those outside the valid input ranges.

Normal Boundary Value Testing

Normal Boundary Value Testing (NBVT) is a technique that focuses on testing the boundaries the input space to uncover potential errors that often occur near extreme values of input variables. The rationale behind NBVT is to test input values at their minimum, just above the minimum, at nominal value, just below the maximum, and at the maximum value. This approach helps to identify common errors such as off-by-one errors, incorrect conditional checks (using < instead of <=), and misunderstandings about where counting should start (from zero or one). Normal boundary value test cases for two variables are shown in Figure



Methodology of Normal Boundary Value Testing

The testing focuses on the boundary values of input variables. This includes:-

▲ Min: The minimum value the variable can take.

▲ Min+: Just above the minimum value.

▲ Nom (Nominal): A typical or expected value (often the midpoint).

▲ Max-: Just below the maximum value.

▲ Max: The maximum value the variable can take.

1. Single Fault Assumption: NBVT often operates under the "single fault" assumption of reliability theory, which indicates that system failures are usually due to a single fault rather than the interaction of multiple faults. This assumption simplifies the testing process by allowing the focus to be on individual variables one at a time.
2. Test Cases Generation: For a function with two variables, for example, the test cases would keep one variable at its nominal value and vary the other through its boundary values. For example, If we have two variables x1 and x2, then

Variable x1 is held at its nominal value, and x2 is tested at its min, min+, nom, max-, and max.

Similarly, x2 is held at its nominal value, and x1 is tested at its min, min+, nom, max-, and max.

Example 1: Normal Boundary Value Testing - Single Variable

Scenario: Consider a system that grants access based on age, where only individuals aged 18 to 65 are allowed entry.

Boundary Values for Age:

Min (18): Test with age 18 to verify that the system grants access.

Min+ (19): Test with age 19 to ensure access is consistently granted just above the minimum age.

Nom (42): A nominal test with age 42 (midpoint of the range) to check normal operation.

Max-(64): Test with age 64, just below the maximum age limit to ensure access is still granted.

Max (65): Test with age 65 to check that the system still grants access at the upper edge.

Testing at these boundaries helps ensure that the system accurately enforces age restrictions by allowing access to eligible individuals while denying it to those outside the age range (under 18 or over 65).

Example 2: Normal Boundary Value Testing - Two Variables

Scenario: Businesses are required to pay GST monthly by submitting their total sales and selecting the applicable GST rate. The rates are variable depending on the type of goods or services provided, commonly 5%, 12%, 18%, and 28%. The system calculates the tax payable and allows businesses to submit their payments online.

Boundary Values for Variables:

Total Sales (x1): Input is expected to range from 70 (minimum) to 10,00,000 (maximum). Bot representing the sales amount for the month.

GST Rate (x2): Standard GST rates applicable: 5% (min), 12%, 18%, and 28% (max).

Testing Boundary Values for Total Sales:

(0,5%) - Test case with no sales and the lowest GST rate (5%).

(1,5%) - Minimum positive sales amount with the lowest GST rate.

(5,00,000, 18%) - Midpoint of the sales range with a commonly used GST rate (18%).

(9,99,999, 28%) - Just below the maximum sales limit with the highest GST rate (28%).

(10,00,000, 28%) - Maximum sales limit with the highest GST rate.

Testing across different GST rates (using a typical sales amount, e.g., 750,000):

(50,000, 5%) - Testing with a lower GST rate applied to a typical sales figure.

(50,000, 12%)

(50,000, 18%)

(50,000, 28%) - Testing with higher GST rates applied.

By testing the system with these specific boundary values, we can ensure that the GST calculation and payment submission process works correctly under various scenarios, including minimum, maximum, and critic points of the input ranges. This approach helps identify potential issues related to calculations, tax rates, and system behavior at the edges of the expected input values.

Example 3 Normal Boundary Value Testing - Three Variables

Scenario: In the Indian railway ticket booking system, passengers can book tickets for different types of train (such as Express, Superfast, Rajdhani) and different classes (Sleeper, 3AC, 2AC, 1AC). Each train type and class combination has a specific number of tickets available per day.

Boundary Values for Variables:

- Train Type (x1): Express (min), Rajdhani (max)
- Class of Travel (x2): Sleeper (min), 1AC (max)
- Number of Tickets Available (x3): Let's assume a range from 0 to 100. (Max 100 tickets)

Testing Boundary Values for Express Train:

- (Express, Sleeper, 0) - Test scenario with no tickets available.
- (Express, Sleeper, 1) - Testing just above the scenario with no availability.
- (Express, Sleeper, 50) - Nominal availability scenario.
- (Express, Sleeper, 99) - Testing just below full capacity.

- (Express, Sleeper, 100) - Test scenario at full capacity.

Test Cases for Rajdhani Train:

- (Rajdhani, 1AC, 0)- Test scenario with no tickets available.
- (Rajdhani, 1AC, 1) - Minimal available tickets scenario.
- (Rajdhani, 1AC, 50) - Nominal ticket availability scenario.
- (Rajdhani, 1AC, 99) - Testing almost full capacity.
- (Rajdhani, 1AC, 100) - Test scenario at full capacity.

By executing these test cases, we can ensure that the ticket booking system handles different scenarios related to ticket availability for Express and Rajdhani trains across various classes of travel. This approach helps in verifying the system's functionality and its ability to manage ticket availability based on the specified constraints and boundaries.

Generalizing the Boundary Value Analysis

Generalizing Boundary Value Analysis (BVA) in software testing refers to extending the traditional boundary value analysis technique to handle a wider range of scenarios, variables, or types of data. This generalization aims to enhance the applicability and effectiveness of BVA by adapting it to different contexts and testing requirements. The normal boundary value analysis technique can be generalized based on the number of variables and the types of ranges.

1. **Generalizing by Number of Variables:**

- For a function with multiple variables, BVA can be generalized by holding all but one variable at their nominal (typical or expected) values while the remaining variable is tested at its boundary values. This is repeated for each variable independently.
- For example, in a function with three variables, applying boundary value analysis by varying each variable through min, min+, nom, max-, and max values results in 4n + 1 unique test cases.
- This approach ensures comprehensive coverage of different variable combinations and od their boundary conditions

Example: Generalizing the BVA by Number of Variables:

Suppose a function calculates a fee based on three variables: age (10 to 65 years), distance traveled (0 to 100 kilometers), and hours of service (1 to 24 hours). Testing might involve:

- Holding distance and hours at their nominal values (50 km and 12 hours), and varying age through its boundary values (10, 11, 32.5, 64, 65).
- Repeating this process for each variable, leading to a series of tests that comprehensively cover in the boundary conditions for each variable.

This method ensures each variable's influence on the outcome is thoroughly examined, with 4n+1 unique test cases generated where n is the number of variables.

2. **Generalizing by Types of Ranges:**

Variables can have different types and ranges. The nature of variables determines the ranges for boundary value analysis.

- Discrete and Bounded Variables: Such as months in a year or days in a month, where boundaries are inherently defined by the domain (e.g., January to December for months).
  Example: A movie ticket booking system allows customers to choose a month for a special monthly screening event. The variables are Months of the year (January to December).Test cases would typically include the first month (January), the last month (December), and a mid-year month like June to cover the boundaries and a nominal value.
- Variables without Explicit Bounds: These require artificial boundaries. For example, ton of me if there is no upper limit specified for a numeric input, the maximum might be set as the largest representable integer.
  Example: In the context of the triangle problem where side lengths are the variables, determining boundary values involves setting the lower bound at 1 (as negative side lengths are invalid) and selecting an upper bound, such as 200 or MAXINT. This generalization ensures that the testing covers a wide range of scenarios, including extreme values and boundary conditions, to validate the behavior of the triangle classification algorithm accurately.
- Boolean and Logical Variables: BVA becomes less useful because they usually have only video two states (True and False). These types of variables are better suited to other testing on techniques like decision table testing.
- Context-Specific Adjustments: For variables like a customer's PIN or transaction type in an ATM system, conventional BVA may not be very insightful or practical because these are typically categorical or have a restricted range of valid inputs.

Limitations of Boundary Value Analysis

1. **Requirement for Ordering Relations:** BVA is most effective when the variables involved have a natural ordering, meaning it is logical to determine that one value is greater than, less than, or equal to another. This is crucial for defining boundary values meaningfully.
   Example: Temperature and pressure have a natural ordering. For instance, 0°C can be logically compared to 100°C (0°C ≤ 100°C). In contrast, sets of colors or names of football teams do not have an intrinsic order. It's not logical to assert that "Red" is less than "Blue" or that "Team A is greater than "Team B".
2. **Independence of Variables:** BVA assumes variables are independent, but this is not always the case. Dependencies between variables can lead to complex interactions that BVA might not adequately test.
   Example: In the context of the NextDate function, the validity of a date depends on the interactions between day, month, and year. February 29 is a valid date but only in a leap year, highlighting a dependency between the day and year that traditional BVA might overlook.

3. **Focus on Physical Quantities:** Boundary value analysis is most suitable for variables representing physical quantities like temperature, pressure, or air speed, where physical boundaries play a crucial role.
   Example: The closure of Bangalore International Airport due to temperatures exceeding the maximum value shows how critical physical boundaries are. Here, BVA could have identified potential issues with instrument settings at extreme temperatures.

4. **Challenges with Logical Variables:** Logical or categorical variables, such as PIN numbers or telephone numbers, do not benefit from BVA due to the lack of physical boundaries or meaningful extremities.
   Example: Consideration of logical variables such as Personal Identification Numbers (PINs) or telephone numbers may not reveal significant faults through boundary value testing. Testing PIN values like 0000, 0001, 5000, 9998, and 9999 may not uncover substantial issues due to the nature of logical, non-physical variables.
5. **Inadequacy in Handling Complex Dependencies:** BVA may fail to account for complex dependencies within the system, which could lead to significant oversights in testing.
   Example: Imagine a digital thermostat that controls both heating and cooling in a smart home system. The thermostat is programmed to switch on heating when the temperature drops to 18°C or lower and activate cooling when the temperature rises to 26°C or higher. In typical BVA, we might test the thermostat's response at 18°C and 26°C separately to ensure triggers the heating and cooling systems correctly. However, suppose the thermostat experiences rapid temperature changes, fluctuating between 17°C and 27°C in a short period due to unusual weather conditions or HVAC issues.
   This scenario could test the thermostat's ability to handle quick switching between heating and cooling, a condition not covered by simple boundary tests for individual temperatures. If there's a delay or failure in switching modes under rapid fluctuation, the system might fail to maintain a stable room temperature, potentially causing discomfort or even damaging the HVAC system due to the rapid cycling of heating and cooling.

Robust Boundary Value Testing

Robust boundary value testing (RBVT) is a simple extension of normal boundary value testing: in addition to the five boundary value analysis values of a variable, we see what happens when the inputs are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min-).

Robust Boundary Value Testing (RBVT) is an extension of Normal Boundary Value Testing (NBVAT) that aims to enhance test coverage by considering values beyond the boundaries. RBVT includes values slightly outside the boundary limits to ensure the software behaves robustly even with inputs that are close to the edges. This approach helps identify potential vulnerabilities and corner cases that may not be captured by Normal Boundary Value Testing.

This approach is designed to assess the robustness of a system by observing how it handles inputs that fall outside the expected input range. It aims to uncover potential failures that could

occur due to inputs that users may not typically provide but could potentially be used either maliciously or by mistake.

Key Components of RBVT

1. Standard Boundary Values:

Min: The smallest value within the acceptable range.

Min+: A value just above the minimum to verify edge cases within the operational range.

Nominal: A typical value expected during regular use.

Max-: A value just below the maximum to test the upper limits of normal operation.

Max: The largest value within the acceptable range.

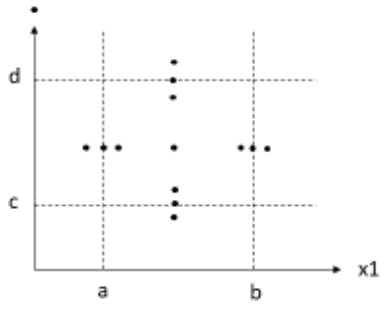
2. Extended Test Values:

Min-: A value slightly less than the minimum accepted input, testing the system's error handling or validation processes.

Max+: A value slightly greater than the maximum accepted input, similarly aimed probing the robustness of error handling and input validation.

Robust Boundary Value Testing is crucial for systems where input validation directly impact functionality and security. By including tests for inputs just outside the accepted ranges, RBVT he ensure that the application is secure against unusual or unexpected inputs, enhancing the over resilience and reliability of the system. This testing approach is particularly valuable in protects against errors that could lead to exceptions, system crashes, or security breaches.

Robust boundary value test cases for two variables are shown in Figure



Robust Boundary Value Testing (RBVT) - Single Variable

Scenario: An online application form requires users to enter their age, which should be between 18 and 65 years inclusive.

Define Boundary and Extended Values:

Min (18 years): Check that the form accepts the minimum age.

Max (65 years): Ensure that the form accepts the maximum age.

Min-(17 years): The form should reject this input, ideally with a clear error message.

Max+ (66 years): Similar to Min-, the form should not accept this age and should provide an error message.

Test Cases:

At Minimum (18 years): Verify the form processes this input correctly.

Just Above Minimum (19 years): Confirm the form continues to function correctly slightly above the minimum.

Nominal (42 years): A common age to test the form under typical conditions.

Just Below Maximum (64 years): Test the upper operational limits.

At Maximum (65 years): Ensure the maximum boundary is respected.

Below Minimum (17 years): The system should identify and reject this out-of-bounds input.

Above Maximum (66 years): Similarly, this should be rejected to confirm robust boundary handling.

Worst-Case Boundary Value Testing

Worst-Case Boundary Value Testing (WCBVT) is a testing approach that goes beyond traditional boundary value testing by considering extreme values for multiple variables simultaneously. This method aims to explore scenarios where more than one variable reaches its boundary limits to assess the software's behavior under such conditions. By analyzing worst-case scenarios, testers can uncover potential vulnerabilities that may not be evident with single-variable boundary testing.

Unlike Robust Boundary Value Testing, which tests each variable independently at and just outside its boundaries, WCBVT involves the Cartesian product of the boundary values of all variables. This approach is designed to detect issues that may arise specifically from interactions between variable at their extreme operational limits.

Key Points

1. Rejecting Single-Fault Assumption: Worst-case boundary value testing challenges the single-fault assumption by examining the impact of extreme values on multiple variables. This approach is particularly useful in scenarios where the failure of the software due to extreme conditions can have severe consequences.
2. Generating Test Cases: To conduct worst-case boundary value testing, testers start with a five-element set for each variable, including the minimum, slightly above minimum, nominal

slightly below maximum, and maximum values. By taking the Cartesian product of these sets for multiple variables, a comprehensive set of worst-case test cases is generated.

3. Comparison with Normal Boundary Value Testing: Worst-case boundary value testing is more exhaustive than normal boundary value testing as it considers extreme values for multiple variables simultaneously. The number of test cases generated for worst-case testing is significantly higher (5° for n variables) compared to normal boundary value testing (4n+1 test cases).

Key Components of WCBVT

1. Standard Boundary Values:

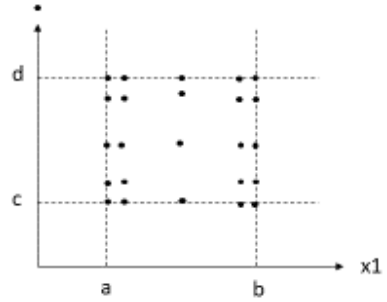
- Min: The smallest value within the acceptable range for each variable.
- Min+: A value just above the minimum, within the operational range.
- Nominal: A typical or average value expected during regular usage.
- Max-: A value just below the maximum, still within the operational limits.
- Max: The largest value within the acceptable range for each variable.

2. Cartesian Product of Boundary Values:

This approach multiplies the boundary scenarios of each variable with every boundary scenario of the other variables, producing a comprehensive set of test cases that explore interactions between variables at their boundary conditions.

Importance of WCBVT

1. WCBVT is important in environments where multiple variables interact in complex ways, potentially impacting the system's behavior under extreme conditions. By systematically testing all combinations of boundary values, WCBVT can uncover issues that might not be visible when variables are tested in isolation or only within their normal operational ranges.
2. This type of testing is particularly useful in critical systems where failure can result in significant consequences, ensuring that the system is robust against a wide range of inputs and conditions. It's essential for ensuring the reliability and stability of systems in real-world scenarios where multiple factors may affect outcomes simultaneously.
   The result of the two-variable version of this is shown in Figure,



Example: Worst Case Boundary Value Testing (WCBVT)

Scenario: A system manages an online promotional campaign where users can enter the number of items they wish to purchase and select a delivery option. The valid range for the number of items is from 1 to 20, and the delivery options are categorized into regular (1) and express (2).

Variables:

Number of Items (x1):

Min: 1 item

Max: 20 items

Delivery Option (x2):

Min: 1 (regular)

Max: 2 (express)

Test Cases: Apply the boundary values for each variable and create combinations using the Cartesian product.

Boundary Values for Number of Items: [1 (Min), 2 (Min+), 10 (Nominal), 19 (Max-), 20 (Max)]

Boundary Values for Delivery Options: [1 (Min, regular), 2 (Max, express)]

Test Case Combinations: Each combination of the above boundary values forms a test case, such as:

(1 item, regular)

(1 item, express)

(20 items, regular)

(20 items, express)

(2 items, regular)

(19 items, express) And so on, through all possible combinations of these boundary values.

Robust Worst-Case Boundary Value Testing (RWCBVT)

RWCBVT takes the principles of both robust boundary value testing and Worst-case Boundary Value Testing to create even more stringent testing environment. This approach focuses on evaluating the software's behavior under extreme conditions while also testing for robustness against unexpected inputs and variations in the boundary values of multiple variables simultaneously.

Key Points

1. Incorporation of Non-valid Inputs: RWCBVT includes values beyond the normal operating range for each variable, testing how the system handles inputs that are typically considered invalid.
2. Cartesian Product Including Out-of-Range Values: The approach involves taking the Cartesian product of extended boundary values (which include values beyond the typical range) for multiple variables. This extensive combination aims to simulate potential extreme scenarios that could arise in actual operations. This involves the Cartesian product of the seven-element sets we used in robustness testing resulting in 7n test cases.
3. Comprehensive Testing Scope: This method is the most exhaustive form of boundary value testing, examining not only the interactions between variables at their defined limits but also how these variables behave when pushed beyond these points. The number of test cases for RWCBVT can be significantly higher, especially when extended boundary conditions are considered.

Key Components of RWCBVT:

1. Standard Boundary Values:

Min: The smallest value within the acceptable range for each variable.

Min+: A value just above the minimum, to verify the system's behavior just within the operational range.

Nominal: A typical or average value expected during regular usage.

Max-: A value just below the maximum, testing the near-upper limit operations.

Max: The largest value considered normal for each variable.

2. Extended Boundary Values:

Min-: A value slightly less than the minimum, testing how the system handles inputs below the acceptable range.

Max+: A value slightly above the maximum, probing the robustness of the system's upper

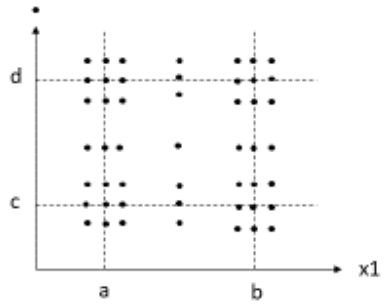
3. Cartesian Product of All Boundary Values: This methodology multiplies every scenario, including out-of-range scenarios, across all variables, creating a comprehensive set of test cases to explore how variable interactions might affect the system under extreme and unexpected conditions.

Importance of RWCBVT:

1. Detecting Hidden Vulnerabilities: By pushing the system beyond its intended operational limits, RWCBVT can uncover hidden issues that might not be evident during standard testing. This is critical for systems where safety and security are paramount.

2. Ensuring System Resilience: This testing is crucial for ensuring that the system can handle erroneous inputs without crashing or behaving unpredictably, which is especially important in high-stakes environments like medical, aerospace, and financial systems.

The below Figure shows the robust worst-case test cases for our two-variable function.



Example Robust Worst Case Boundary Value Testing (RWCBVT)
Scenario: A flight booking system that allows users to select the number of travelers and class of service. The system typically handles up to 5 travelers and offers three classes of service (Economy, Business, First Class).
Variables:

- Number of Travelers (x1):
  Min: 1 traveler
  Max: 5 travelers

- Class of Service (x2):
  Min: 1 (Economy)
  Max: 3 (First Class)

Test Cases: Apply both the typical and extended boundary values for each variable and create combination using the Cartesian product:

- Boundary Values for Number of Travelers: [0 (Min-), 1 (Min), 3 (Nominal), 5 (Max), 6 (Max+)]
- Boundary Values for Class of Service: [0 (Min-), 1 (Economy), 2 (Business), 3 (First Class), 4 (Max+)]

Test Case Combinations: Each combination of the above boundary values forms a test case, such as:

- (0 travelers, Economy) - Testing below minimum travelers with the lowest class.
- (1 traveler, 4 classes) - Minimum travelers with beyond maximum class.
- (6 travelers, 0 class) - Above maximum travelers with non-valid class.
- (5 travelers, First Class) - Maximum valid travelers and maximum valid class.

And so on, through all possible combinations of these boundary values. These combinations ensure that every possible extreme and out-of-bound scenario is tested, providing insights into how well the system ca maintain functionality and reliability under adverse conditions.

Special Value Testing

Special Value Testing also known as ad hoc testing is a form of functional testing that relies on the tester's domain knowledge, experience with similar programs, and understanding of potential weak points in the software to design test cases. This approach is highly dependent on the tester's judgment and expertise as no specific guidelines are followed other than best engineering practices. Special Value Testing can be valuable in uncovering faults that may not be easily detected by other testing methods. Testers using this approach often consider unique or critical scenarios that may not be covered by traditional boundary value testing.

Characteristics of Special Value Testing:

1. Tester-Driven: Special Value Testing heavily relies on the tester's judgment, experience, and expertise. The effectiveness of this testing approach is highly dependent on the individual tester's capabilities and insights in identifying critical test scenarios.
2. Domain Knowledge: Testers leverage their in-depth understanding of the application's pinpoint specific functionalities or modules where defects are more likely to occur and focus domain to anticipate potential error-prone areas. By applying domain knowledge, testers can their testing efforts accordingly.
3. Ad Hoc Approach: Special Value Testing follows an ad hoc approach, lacking standardized procedures or guidelines. Testers have the flexibility to design test cases based on their intuition, experience, and knowledge without strict adherence to predefined testing methodologies. This creative freedom allows testers to explore unique scenarios that may not be covered by traditional testing techniques.
4. Targeting "Soft Spots": Special Value Testing aims to target "soft spots" within the software, which are areas known to be prone to errors or vulnerabilities. These soft spots may include blow complex calculations, unusual input types, historically problematic modules, or functionalities with a higher likelihood of defects. By focusing on these critical areas, testers can uncover hidden issues that might not be revealed through standard testing methods.

Importance of Special Value Testing:

This method is particularly valuable for:

1. Uncovering Rare Issues: By focusing on specific, often rare conditions that are not typically covered by other testing methods
2. Highly Contextual Applications: Effective in complex systems where the tester's deep understanding of the application can guide the testing process.
3. Areas Prone to Errors: Particularly useful in areas known for their susceptibility to bugs, where testers can apply their insights to explore specific scenarios thought to be risky. Example

Example: Special Value Testing

Scenario: The "NextDate" function calculates the next day's date given a specific day, month, and year. This function can be particularly tricky around the boundaries of months, changes in year, and especially during leap years.

In the context of the NextDate function, Special Value Testing may involve creating test cases related to specific dates such as February 28, February 29 in leap years, and other scenarios that are not covered by standard boundary value testing. While this method may lack the systematic approach of boundary value testing, it can be effective in revealing faults and vulnerabilities in the software.

Special Value Test Cases:

February 28 in a Non-Leap Year: Test what happens on the day after February 28. The expected result should be March 1 of the same year.

February 28 in a Leap Year: Here, the next day should be February 29.

February 29 in a Leap Year: Testing the transition from February 29 to March 1 in a leap year is crucial since February 29 does not exist in non-leap years.

December 31: Testing the transition from December 31 to January 1 of the following year to verify that the function handles year changes correctly.

Triangle Problem

Normal Boundary Value Test Cases/ Boundary Value Test Cases for x, y, z :

min value = 1

close to min = 2

nominal = 200

close to max = 199

max = 200

Test cases are,

4*3 + 1 = 13

C1. $1 \leq a \leq 200$.

C2. $1 \leq b \leq 200$.

C3. $1 \leq c \leq 200$.

C4. $a < b + c$.

C5. $b < a + c$.

C6. $c < a + b$.

Case	X	Y	Z	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	199	100	Isosceles
9	100	200	100	Not a triangle
10	1	100	100	Isosceles
11	2	100	100	Isosceles
12	199	100	100	Isosceles
13	200	100	100	Not a triangle

2. Robust Test Cases: Range (100 to 500) – for x, y, z :

min value : 100

close to min : 101

nominal : 300

close to max : 499

max : 500

lesser than min value : 99

larger than max value : 501

Total test cases,

= 6*n+1 = 6*3+1 = 19. So there will be extra 6 cases apart from the above 13 cases –

X	Y	Z
99	300	300
501	300	300
300	99	300
300	501	300
300	300	99
300	300	501

3. Worst Test Cases: Range (1 – 200)

If we reject “single” fault assumption theory of reliability, and consider cases where more than 1 variable has extreme values, then it is known as worst case analysis.

Total no. of test cases,

5^n = 5^3 = 125 cases

Mathematically, the test cases will be a cross product of 3 sets –

{1, 2, 100, 199, 200}

x {1, 2, 100, 199, 200}

x {1, 2, 100, 199, 200}

Let set A,

= {1, 2, 100, 199, 200}

So, the set of worst cases will be represented by,

= A x A x A

Case	a	b	c	Expected Output
1	1	1	1	Equilateral
2	1	1	2	Not a Triangle
3	1	1	100	Not a Triangle
4	1	1	199	Not a Triangle
5	1	1	200	Not a Triangle
6	1	2	1	Not a Triangle
7	1	2	2	Isosceles
8	1	2	100	Not a Triangle
9	1	2	199	Not a Triangle
10	1	2	200	Not a Triangle
11	1	100	1	Not a Triangle
12	1	100	2	Not a Triangle
13	1	100	100	Isosceles
14	1	100	199	Not a Triangle
15	1	100	200	Not a Triangle
16	1	199	1	Not a Triangle
17	1	199	2	Not a Triangle
18	1	199	100	Not a Triangle
19	1	199	199	Isosceles
20	1	199	200	Not a Triangle
21	1	200	1	Not a Triangle
22	1	200	2	Not a Triangle
23	1	200	100	Not a Triangle
24	1	200	199	Not a Triangle
25	1	200	200	Isosceles
…	…	…	…	…

NextDate Function

Since BVA yields (4n + 1) test cases according to single fault assumption theory, hence we can say that the total number of test cases will be (4*3+1)=12+1=13.

C1: $1 \leq \text{month} \leq 12$

C2: $1 \leq \text{day} \leq 31$

C3: $1900 \leq \text{year} \leq 2025$.

Normal BVA for nextDate function

Test Case ID	Month (mm)	Day (dd)	Year (yyyy)	Expected Output
1	6	15	1900	16 June, 1900
2	6	15	1901	16 June, 1901
3	6	15	1962	16 June, 1962
4	6	15	2024	16 June, 2024
5	6	15	2025	16 June, 2025
6	6	1	1962	2 June, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	1-Jul-62
9	6	31	1962	Invalid Date as June has 30 Days
10	1	15	1962	16 January, 1962
11	2	15	1962	16 February, 1962
12	11	15	1962	16 November, 1962
13	12	15	1962	16 December, 1962

This is how we can apply BVA technique to create test cases for our Next Date Problem.

Worst-case Test cases based on 5^n – 5x5x5 = 125 test cases

Case	Month	Day	Year	Expected Output
1	1	1	1812	1/2/1812
2	1	1	1813	1/2/1813
3	1	1	1912	1/2/1912
4	1	1	2011	1/2/2011
5	1	1	2012	1/2/2012
6	1	2	1812	1/3/1812
7	1	2	1813	1/3/1813
8	1	2	1912	1/3/1912
9	1	2	2011	1/3/2011
10	1	2	2012	1/3/2012
11	1	15	1812	1/16/1812
12	1	15	1813	1/16/1813
13	1	15	1912	1/16/1912
14	1	15	2011	1/16/2011
15	1	15	2012	1/16/2012
16	1	30	1812	1/31/1812
17	1	30	1813	1/31/1813
18	1	30	1912	1/31/1912

19	1	30	2011	1/31/2011
20	1	30	2012	1/31/2012
21	1	31	1812	2/1/1812
22	1	31	1813	2/1/1813
23	1	31	1912	2/1/1912
24	1	31	2011	2/1/2011
25	1	31	2012	2/1/2012
26	2	1	1812	2/2/1812
27	2	1	1813	2/2/1813
28	2	1	1912	2/2/1912
29	2	1	2011	2/2/2011
30	2	1	2012	2/2/2012
31	2	2	1812	2/3/1812
32	2	2	1813	2/3/1813
33	2	2	1912	2/3/1912
34	2	2	2011	2/3/2011
35	2	2	2012	2/3/2012
36	2	15	1812	2/16/1812
37	2	15	1813	2/16/1813
38	2	15	1912	2/16/1912
39	2	15	2011	2/16/2011
40	2	15	2012	2/16/2012
41	2	30	1812	Invalid Date
42	2	30	1813	Invalid Date
43	2	30	1912	Invalid Date
44	2	30	2011	Invalid Date
45	2	30	2012	Invalid Date
46	2	31	1812	Invalid Date
47	2	31	1813	Invalid Date
48	2	31	1912	Invalid Date
49	2	31	2011	Invalid Date
50	2	31	2012	Invalid Date
51	6	1	1812	6/2/1812
52	6	1	1813	6/2/1813
53	6	1	1912	6/2/1912
54	6	1	2011	6/2/2011
55	6	1	2012	6/2/2012
56	6	2	1812	6/3/1812
57	6	2	1813	6/3/1813

58	6	2	1912	6/3/1912
59	6	2	2011	6/3/2011
60	6	2	2012	6/3/2012
61	6	15	1812	6/16/1812
62	6	15	1813	6/16/1813
63	6	15	1912	6/16/1912
64	6	15	2011	6/16/2011
65	6	15	2012	6/16/2012
66	6	30	1812	7/1/1812
67	6	30	1813	7/1/1813
68	6	30	1912	7/1/1912
69	6	30	2011	7/1/2011
70	6	30	2012	7/1/2012
71	6	31	1812	Invalid Date
72	6	31	1813	Invalid Date
73	6	31	1912	Invalid Date
74	6	31	2011	Invalid Date
75	6	31	2012	Invalid Date
76	11	1	1812	11/2/1812
77	11	1	1813	11/2/1813
78	11	1	1912	11/2/1912
79	11	1	2011	11/2/2011
80	11	1	2012	11/2/2012
81	11	2	1812	11/3/1812
82	11	2	1813	11/3/1813
83	11	2	1912	11/3/1912
84	11	2	2011	11/3/2011
85	11	2	2012	11/3/2012
86	11	15	1812	11/16/1812
87	11	15	1813	11/16/1813
88	11	15	1912	11/16/1912
89	11	15	2011	11/16/2011
90	11	15	2012	11/16/2012
91	11	30	1812	Invalid Date
92	11	30	1813	Invalid Date
93	11	30	1912	Invalid Date
94	11	30	2011	Invalid Date
95	11	30	2012	Invalid Date
96	11	31	1812	Invalid Date

97	11	31	1813	Invalid Date
98	11	31	1912	Invalid Date
99	11	31	2011	Invalid Date
100	11	31	2012	Invalid Date
101	12	1	1812	12/2/1812
102	12	1	1813	12/2/1813
103	12	1	1912	12/2/1912
104	12	1	2011	12/2/2011
105	12	1	2012	12/2/2012
106	12	2	1812	12/3/1812
107	12	2	1813	12/3/1813
108	12	2	1912	12/3/1912
109	12	2	2011	12/3/2011
110	12	2	2012	12/3/2012
111	12	15	1812	12/16/1812
112	12	15	1813	12/16/1813
113	12	15	1912	12/16/1912
114	12	15	2011	12/16/2011
115	12	15	2012	12/16/2012
116	12	30	1812	12/31/1812
117	12	30	1813	12/31/1813
118	12	30	1912	12/31/1912
119	12	30	2011	12/31/2011
120	12	30	2012	12/31/2012
121	12	31	1812	1/1/1813
122	12	31	1813	1/1/1814
123	12	31	1912	1/1/1913
124	12	31	2011	1/1/2012
125	12	31	2012	1/1/2013

Commission Problem

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri.

- Locks cost $45,
- Stocks cost $30,
- Barrels cost $25.

The salesperson had to sell at least one lock, one stock, and one barrel (at least one complete rifle) per month

Production limits:

- Maximum of 70 locks ; 1<=locks<=70
- Maximum of 80 stocks ; 1<=stocks<=80
- Maximum of 90 barrels ; 1<=barrels<=90

After each town visit, the salesperson sent a telegram to the Missouri gunsmith with the number of locks, stocks, and barrels sold in that town.

At the end of a month, the salesperson sends a very short telegram showing

–1 lock sold (to indicate the end of inputs for a particular sales person)

The gunsmith then knew the sales for the month were complete and computed the salesperson's commission as follows:

10% on sales up to (and including) $1000

15% on the next $800

20% on any sales in excess of $1800.

The commission program produced a monthly sales report that gave the total number of locks, stocks and barrels sold, the salesperson's total dollar sales and finally the commission.

Sales = lock_price * total_locks + stock_price * total_stocks + barrel_price * total_barrels

If sales >1800

Commission =0.10 * 1000.0

Commission =commission + 0.15*800.0

Commission =commission+0.20* (sales-1800.0)

Else If sales >1000

Commission =0.10 * 1000.0

Commission=commission + 0.15*(sales-1000.0)

Else

Commission=0.10*sales

For commission problem, we look into boundary values for the output range especially near the threshold range 1000$ and 1800$.

[Note: This is Generalized table to produce test cases, to be

Sales value range from 100 (min) to 7800 (max).

LOCKS	STOCKS	BARRELS	SALES	COMMENT	
1	1	1	100	MIN	min+
5	5	5	500	MID	
10	10	10	1000	BORDER	border- border+
14	14	14	1400	MID	
18	18	18	1800	BORDER	border-
48	48	48	4800	MID	
70	80	90	7800	MAX	

CHECKING BOUNDARY VALUE FOR LOCKS, STOCKS AND BARRELS AND COMMISSION

Commission Problem Output Boundary Value Analysis Cases

Case Id	Description	Input data			Expected output		Actual output	
		Locks	Stocks	Barrels	Sales	Commission	Sales	Commission
1	Enter the min value for locks, stocks and barrels	1	1	1	100	10		
2	Enter the min value for 2 items and min +1 for any one item	1	1	2	125	12.5		
3		1	2	1	130	13		
4		2	1	1	145	14.5		
5	Enter the value sales approximately mid value between 100 to 1000	5	5	5	500	50		
6	Enter the values to calculate the commission for sales nearly less than 1000	10	10	9	975	97.5		
7		10	9	10	970	97		
8		9	10	10	955	95.5		

9	Enter the value sales exactly equal to 1000	10	10	10	1000	100		
10	Enter the values to calculate the commission for sales nearly greater than 1000	10	10	11	1025	103.75		
11		10	11	10	1030	104.5		
12		11	10	10	1045	106.75		
13	Enter the value sales approximately mid value between 1000 to 1800	14	14	14	1400	160		
14	Enter the values to calculate the commission for sales nearly less than 1800	18	18	17	1775	216.25		
15		18	17	18	1770	215.5		
16		17	18	18	1755	213.25		
17	Enter the value sales exactly equal to 1800	18	18	18	1800	220		
18	Enter the values to calculate the commission for sales nearly greater than 1800	18	18	19	1825	225		
19		18	19	18	1830	226		
20		19	18	18	1845	229		
21	Enter the normal value for Locks, Stocks and Barrels	48	48	48	4800	820		
22	Enter the max value for 2 items and max - 1 for any one item	70	80	89	7775	1415		
23		70	79	90	7770	1414		
24		69	80	90	7755	1411		
25	Enter the max value for Locks, Stocks and Barrels	70	80	90	7800	1420		

Random Testing

Random Testing is a methodology within the field of software testing where inputs are generated randomly to test the system rather than selecting them based on any predetermined criteria such as boundary conditions or typical values. This approach is unique in its reliance on randomness to uncover errors. It a potentially powerful tool for identifying hidden issues in the software. Random testing is a valuable approach to software testing that involves using a random number generator to select test case values. This method helps in avoiding bias in testing and can uncover unexpected issues in the software.

1. **Statistical Basis:** Random testing is often discussed within academic due to its statistical nature. By using random inputs, the testing process attempts to simulate a broad spectrum of user interactions, potentially uncovering less obvious faults.
2. **Use of Random Number Generators:** Inputs for testing are selected using random number generators to ensure that the values are spread across the entire input domain of the variable being tested. This method helps in mitigating tester biases that might influence the choice of test data.
3. **Challenge of Determining Test Adequacy:** One of the critical challenges with random testing is deciding how many test cases are sufficient to confidently assert the software's reliability. This decision can be somewhat subjective and often requires statistical or risk-based approaches to resolve.
4. **Considerations for Random Testing:** Random testing can be particularly useful for exploring a wide range of inputs and scenarios that may not be covered by traditional test cases. It is important to ensure that the random number generator used is truly random and provides a uniform distribution of values. Random testing can be combined with other testing techniques to achieve comprehensive test coverage. The effectiveness of random testing can be influenced by the quality of the random number generator and the size of the input domain.
5. **Determining the Number of Random Test Cases:** One common question in random testing is how many random test cases are sufficient to provide adequate test coverage. The number of random test cases needed can vary depending on factors such as the complexity of the software, the size of the input domain, and the quality of the random number generator. In practice, running a large number of random test cases can help increase confidence in the robustness of the software.

Advantages of Random Testing:

1. Comprehensive Coverage: Random testing can potentially cover a wide range of input scenarios than manual selection methods.
2. Unbiased Testing: It reduces the likelihood of unconscious bias in choosing test cases, which might overlook certain types of errors.

Limitations:

1. Less Efficiency: It may require a large number of tests to achieve sufficient coverage, especially for variables with a wide range of possible values.

2. Difficulty in Reproducing Errors: Randomly generated test cases can make it hard to reproduce failures unless the specific inputs causing the failure are recorded.

Guidelines for Boundary Value Testing

Boundary value testing is a crucial aspect of software testing that focuses on testing the boundaries of input and output ranges of a program. This technique can be especially useful when testing for edge cases, error message handling, and robustness of internal variables like loop controls and pointers. Key Guidelines for Boundary Value Testing:

1. **Understand Variable Relationships:** Make sure that variables in the software are independent of each other. If they are not independent, consider how they affect each other to avoid unrealistic test scenarios. For example, ensure that dates like June 31 are not accepted.
2. **Apply Testing Widely:** Extend boundary value testing to not only input ranges but also output values and internal variables like indices. Test error messages and output values to ensure they are within expected limits. Use robust testing for internal variables to catch common errors.
3. **Use Semantic Understanding:** Understand the real-world purpose of the software function being tested. This helps in creating test cases that are more relevant and avoid impossible scenarios. For instance, knowing that a function calculates mileage per liter petrol can help in avoiding negative values or division by zero.
4. **Create Diverse Test Cases:** Develop test cases that cover a range of values, including minimum, maximum, values just inside these boundaries, typical values, and values slightly beyond the boundaries. This comprehensive approach ensures thorough testing.
5. **Avoid Strict Technical Focus:** Move away from solely technical testing methods and consider the practical application of the software. By incorporating semantic information, test scenarios become more meaningful. For example, when testing a banking application, ensuring that a user cannot transfer a negative amount of money or transfer funds without having sufficient balance adds practical value to the testing process.

By following these guidelines, testers can conduct effective boundary value testing that covers a wide range of scenarios, considers real-world applications, and ensures the software functions correctly in various situations.