SWE30009 – Software Testing and Reliability

Project Report

Student name: Vi Luan Dang

Student ID: 103802759

Task 1: Random Testing

Subtask 1.1: Understanding Random Testing Methodology

Intuition of random testing

Random testing, also known as monkey testing, is a software testing technique where the system is tested by generating random and independent inputs and test cases. It is a black box testing technique, meaning the tester does not need to know the internal workings of the system. The goal is to find unexpected behaviors or bugs by providing random inputs and observing the outputs.

Distribution profiles for random testing

To understand this concept, let's take a look at an array of integers with a size of 7, the value of integer can be negative or positive. If that is the case, we will have 2 approaches of distribution.

1. Uniform Distributions

In uniform distributions, each input has an equal probability of being selected. This means that every possible input is equally likely to be chosen during the testing process. In this approach, every possible integer within a specified range has an equal chance of being selected

2. Operational Distributions (Profiles)

In operational distributions, the selection probability follows the probability of being used in real-world operations. Inputs are chosen based on their likelihood of occurrence in actual usage scenarios. For example, if we know that positive numbers are more common in real usage, we can adjust the probabilities accordingly.

Process of random testing

1. **Identify Input Domain**: Define the range and types of inputs that the system can accept.
2. **Select Test Inputs Independently/Randomly from the Input Domain**: Use a random generator to produce inputs based on the defined input space and distribution profile.
3. **Test the System on These Inputs and Form a Random Test Set**: Feed the random inputs into the system and observe the behavior.
4. **Compare the Results with System Specification**: Check whether the outputs match the expected results.
5. **Take Necessary Action if the Report Fails**: Investigate and fix any issues if the outputs do not match the expected results.

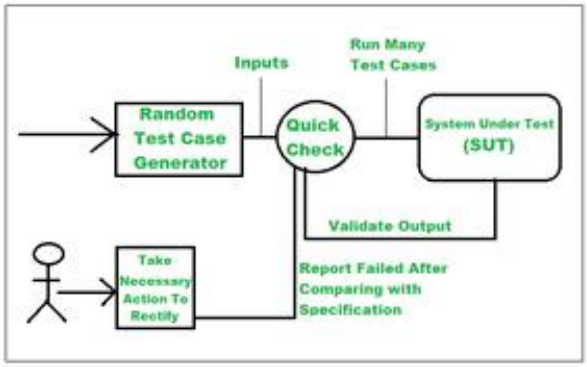

Figure 1: Random Testing process

Application of random testing

1. Uniform Distribution

In a uniform distribution approach, every possible input within a specified range has an equal probability of being selected. This approach is useful in scenarios where every input is equally likely and there is no prior knowledge about the input distribution. For example:

- **Stress Testing of Web Services**: Ensures all request types and parameters are tested equally.
- **Randomized User Interface Testing**: Generates random user actions to test the robustness of the interface.

2. Operational Distribution (Profiles)

In an operational distribution approach, inputs are generated based on their likelihood of occurrence in real-world scenarios. This approach is useful when certain inputs are more common or likely than others, and testing should focus more on these common cases. For example**:**

- **Testing E-commerce Websites**: Focuses on common user actions and popular products to simulate real-world usage.
- **Mobile Application Testing**: Simulates frequent user actions and common navigation paths based on real-world usage data.

Subtask 1.2: Generating Concrete Test Cases for the provided program

To test a program designed to sort a non-empty list of integer numbers, including duplicated numbers, we can generate test cases based on different partitions of inputs using random testing. Below is a table of possible test cases.

Domain Partition	Input list	Expected output
All positive number	[1048576, 98304, 327, 2097152, 65536, 4194304]	[327, 65536, 98304, 1048576, 2097152, 4194304]
All negative number	[-50, -2000, -1500, -750, -2500, -1000]	[-2500, -2000, -1500, -1000, -750, -50]
All possible value	[134, -572, 8, -43, 89, 0, -231]	[-572, -231, -43, 0, 8, 89, 134]
All Possible Values with Duplicates	[45, -12, 7, 45, -3, -12, 7, 89, -99, 45]	[-99, -12, -12, -3, 7, 7, 45, 45, 45, 89]

The above test cases demonstrate how random testing can be used to test a specific program; it follows three main steps:

1. **Identify Input Domain**: Non-empty lists of integers, which may include duplicates and both negative and positive values.
2. **Generate Random Inputs**: Use a random generator to create lists of integers. This could be script written in any languages, below is a Python code to generate random input/

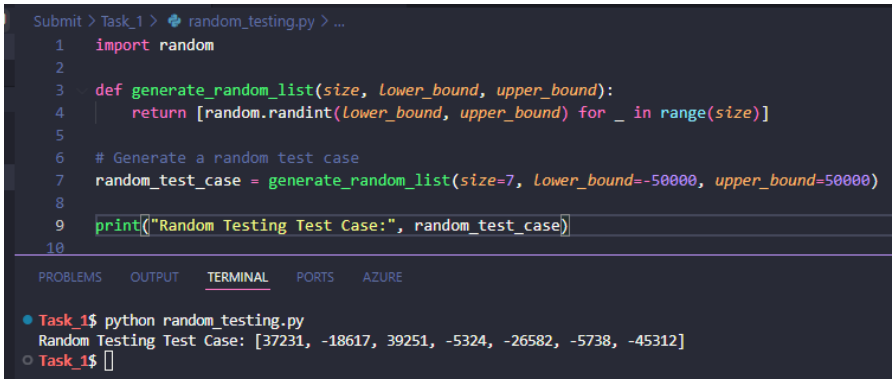

Figure 2: Random testing

3. **Expected Output**: A sorted list of integers in ascending order.

Task 2: Metamorphic Testing

Subtask 2.1: Understanding Metamorphic Testing Methodology

What is Test Oracle

A test oracle is a mechanism or process used to determine whether the outputs of a software program are correct, test oracle of traditional testing and metamorphic testing are different to address evolving issues:

1. **In traditional testing**, a test oracle is a mechanism or source that provides the expected output for a given input. It is used to verify the correctness of the software by comparing the actual output generated by the system under test against the expected output defined by the oracle
2. **In metamorphic testing**, the concept of a test oracle is adapted to address scenarios where traditional oracles are not available. Instead of providing a direct expected result, metamorphic testing relies on metamorphic relations (MRs), which describe how outputs should change in response to specific changes in inputs.

Example of Test Oracle

Traditional Testing:

- Input: A list of integers [4, 2, 7, 1]
- Expected Output (Oracle): [1, 2, 4, 7]
- Test Result: The software's output is compared to the expected sorted list. If the software returns [1, 2, 4, 7], the test passes.

Metamorphic Testing:

- Input: [4, 2, 7, 1] (sorted)
- Metamorphic Relation: Sorting twice should give the same result.
- Follow-up Test Case: [1, 2, 4, 7] (result of sorting [4, 2, 7, 1])
- Check (Oracle): Sorting [1, 2, 4, 7] again should still yield [1, 2, 4, 7].

What are Untestable Systems

Untestable systems are those where determining the correctness of outputs cannot be verified (or cannot be verified in practice:

Reason	Example
No Clear Way to Verify Outputs	Generative art systems create artwork based on algorithmic processes. Since art is subjective, it's challenging to define a "correct" output
Complex or Context-Dependent Conditions	Autonomous vehicles must navigate real-world environments filled with unpredictable variables like pedestrians, other vehicles, and changing weather conditions.
Varying Outputs Under Same Conditions	Procedural content generation systems create game environments or levels dynamically
Hard-to-Establish Relationships	Personalized recommendation systems, like those used by streaming services or online retailers, provide suggestions based on user behavior and preferences. The relationship between the input data (user interactions) and the output (recommendations) is complex and influenced by many factors.

Motivation

Metamorphic testing is motivated by the need to test systems where traditional methods fail due to the absence of precise test oracles. It focuses on the relationships between inputs and outputs, leveraging these to test complex systems more effectively. By using MRs, metamorphic testing ensures that systems behave consistently according to defined input-output relationships, even when exact results are not known.

Intuition

The intuition behind metamorphic testing is that it is often easier to understand how changes in inputs should affect outputs than to define exact expected outputs. By generating follow-up test cases from initial ones and verifying the consistency of outputs based on MRs, this method helps to uncover inconsistencies and bugs in systems where traditional testing is impractical or infeasible. This is clearly demonstrated in the above example of test oracles between traditional testing and metamorphic testing.

Metamorphic Relations

A metamorphic relation (MR) is a rule that describes how the output of a system should change in response to specific modifications in the input. MRs are used to generate follow-up test cases, ensuring that the system behaves consistently across different but related inputs. For example, in a sorting algorithm, an MR might state that sorting a list twice should yield the same result as sorting it once.

Apart from the examples of sorting algorithm above, here are some other examples of metamorphic relations.

1. **Search Engine:**
   - **MR**: Adding irrelevant words to a search query should not change the relevance of the search results for the original query terms.

   Example:
   - Input Query: "best pizza recipes"
   - Follow-up Query: "best pizza recipes xyz123"
   - Check: The search results for "best pizza recipes xyz123" should be the same or very similar to those for "best pizza recipes."

2. **API Testing:**
   - MR: The response to a GET request should remain consistent if the same request is made multiple times.

   Example:
   - Initial Request: GET /api/v1/users/123
   - Follow-up Request: Repeat GET /api/v1/users/123
   - Check: Both requests should return the same user data.

Process of Metamorphic Testing

1. **Determine Relevant MRs**: Identify MRs that are applicable to the program being tested.
2. **Select Source Test Cases**: Choose initial test cases based on the identified MRs.
3. **Generate Follow-up Test Cases**: Create test cases by applying the MRs to the source test cases.
4. **Run the Program**: Execute the program with both source and follow-up test cases.
5. **Compare Outputs**: Check if the outputs of follow-up test cases adhere to the expected relationships defined by the MRs.
6. **Identify Deviations**: Detect any inconsistencies or deviations between expected and actual outputs.
7. **Modify the Program**: Make necessary corrections to the program and re-run tests if needed.

Applications of Metamorphic Testing

Metamorphic testing can be applied across various domains to ensure software correctness and reliability. Here are some additional applications:

1. **Financial Systems:**

MR: Small fluctuations in stock prices should not cause large discrepancies in calculated portfolio values.
Example: Evaluating portfolio values with slightly different stock prices should result in similar portfolio values.

2. **Simulation Software:**

MR: Running a simulation with a fixed random seed should produce the same results.
Example: Running a weather simulation with the same initial conditions and random seed should yield identical results each time.

3. **Data Cleaning and Transformation:**

MR: Removing duplicates from a dataset should not affect aggregated statistics.
Example: Calculating the average age in a dataset before and after removing duplicate entries should yield the same result.

Subtask 2.2: Applying Metamorphic Testing to the provided Sorting Program

Metamorphic Relation 1: Idempotence of Sorting

Description: Sorting a list twice should yield the same result as sorting it once.
Intuition: If a sorting algorithm works correctly, applying it to an already sorted list should not change the order of the elements.
Concrete Metamorphic Group:

- Source test case input: [7, 3, 5, 1, 8]
- Source test case output: [1, 3, 5, 7, 8]
- Follow-up test case: [1, 3, 5, 7, 8] (result of sorting [7, 3, 5, 1, 8])
- Check that sorting [1, 3, 5, 7, 8] again results in [1, 3, 5, 7, 8]

Metamorphic Relation 2: Symmetry of Sorting

Description: Sorting a list should produce the same result as sorting its reverse.
Intuition: The order of elements in a list, whether ascending or descending, should not affect the final sorted result. Reversing the list and then sorting it should give the same result as sorting it directly.
Concrete Metamorphic Group:

- Source test case input: [9, 2, 4, 6]
- Source test case output: [2, 4, 6, 9]
- Follow-up test case: [6, 4, 2, 9] (result of reversing [9, 2, 4, 6])
- Check that sorting [9, 2, 4, 6] results in sorting [6, 4, 2, 9]

Metamorphic Relation 3: Invariance to Adding Identical Elements

Description: Adding identical elements to a list should not change the relative order of the existing elements when sorted.
Intuition: When identical elements are added to a list, their addition should not affect the relative ordering of the other elements in the list after sorting. This MR tests the stability of the sorting algorithm.

Concrete Metamorphic Group:

- Source test case input: [4, 1, 7, 2]
- Source test case output: [1, 2, 4, 7] (result of sorting [4, 1, 7, 2])
- Follow-up test case: Add identical elements [4, 1, 7, 2, 2, 2] (adding three 2s to the original list)
- Check that sorting [4, 1, 7, 2, 2, 2] results in [1, 2, 2, 2, 4, 7]

Metamorphic Relation 4: Permutation Invariance

Description: The output of the sorting algorithm should be the same regardless of the order of the input list.
Intuition: A correct sorting algorithm should only consider the values in the list and not their initial positions. Hence, permuting the input list and then sorting should yield the same result as sorting the original list.
Concrete Metamorphic Group:

- Source test case input: [3, 5, 2, 4]
- Source test case output: [2, 3, 4, 5]
- Follow-up test case: [5, 2, 4, 3] (permutation of [3, 5, 2, 4])
- Check that sorting [5, 2, 4, 3] results in [2, 3, 4, 5]

Subtask 2.3: Comparison of Random Testing and Metamorphic Testing

Aspect	Random Testing	Metamorphic Testing
Oracle Handling	**Oracle Problem**: Difficult to determine correctness without a known expected result or ground truth.	**Handles Oracle Problem:** Uses metamorphic relations to predict expected outputs, reducing reliance on ground truth.
Test Case Generation	**Non-Targeted**: Random test cases may not cover all important aspects or edge cases of the system.	**Systematic Generation**: Uses defined metamorphic relations to generate meaningful test cases.
Exploration	**Wide Input Coverage**: Can explore a broad range of inputs, potentially uncovering unexpected bugs.	**Focused Testing**: Ensures that changes to inputs are related to specific properties of the system.
Efficiency	**Efficient for Simple Cases**: Can be effective if the system is well-understood and the number of cases is manageable.	**Efficient for Complex Cases**: Reduces the need for exhaustive testing by focusing on input-output relationships.
Flexibility	**High**: Applicable to various types of systems without needing detailed knowledge of their internal workings.	**Context-Specific**: Best used where metamorphic relations can be clearly defined and applied.

Task 3: Test a program of your choice.

Program Selection:

Our selected program is a program that calculates the area of a rectangle given its length and width. This type of program allows us to define clear metamorphic relations. The constraints are that width and length of the rectangle must be greater than 1.

```
1  def rectangle_area_original(length, width):
2      return length * width
```

Figure 3: Chosen program.

Practical Application of the selected program

The rectangle_area program, designed to compute the area of a rectangle given its length and width, serves as a fundamental component in various practical applications. In everyday scenarios, such a program is invaluable for tasks ranging from interior design to land management. For example, in construction and home renovation, calculating the area of rooms, floors, and walls helps in estimating material requirements, such as paint, flooring, and tiles. In agriculture, determining the area of plots for planting crops or allocating space for livestock is crucial for effective land use.

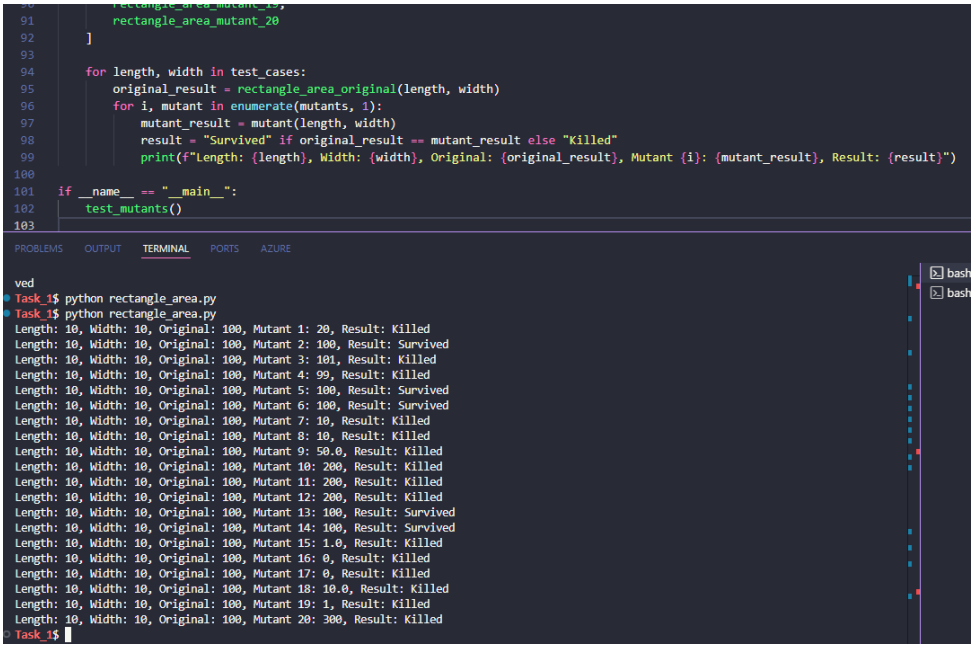

Figure 4: Original program running with its mutants

Mutants' discussion:

Mutant ID	Original Code	Changed Code	Description
1	length * width	length + width	Change multiple to addition
2	length * width	width * length * length	Multiple length twice
3	length * width	length * width + 1	Add 1 after multiplying
4	length * width	length * width - 1	Minus 1 after multiplying
5	length * width	length * length	Multiply length with length
6	length * width	width * width	Multiply width with width
7	length * width	max(length, width)	Return the greater of width and length
8	length * width	min(length, width)	Return the lesser of width and length
9	length * width	(length * width) / 2	Divide by 2 after multiplying
10	length * width	length * width * 2	Multiply by 2 after multiplying
11	length * width	(length + width) * width	Add square of width
12	length * width	length * (width + length)	Add square of length
13	length * width	length * 10	Multiply length by 10
14	length * width	width * 10	Multiply width by 10
15	length * width	length / width	Divide length with width
16	length * width	length - width	Minus length with width
17	length * width	width - length	Minus width with length
18	length * width	(length + width) / 2	Divide sum of length and width
19	length * width	length // width	Integer division of length by width
20	length * width	length * width * 3	Multiply product of length and width by 3

These are the 20 mutants deprived from the original function of calculating the area of the rectangle; by listing out all the mutants, we can have a better understanding of our test cases. Following this part, we will create some test cases to test against all 20 of these mutants.

Metamorphic relation selection

Metamorphic Relation 1: Addition Relation

Description: Add a constant to both length and width and check if the results are consistent with the addition of that constant in the original results.
Intuition: This helps detect if mutants produce consistent results under additions. For instance, if adding 1 to both dimensions changes the output in a consistent manner, it should be reflected in the results.

Metamorphic Relation 2: Swapping Relation

Description: Swap length and width and check if the mutants produce the same results as the original function.
Intuition: If swapping dimensions should not affect the area result (for the original function), any mutant that does not preserve this property can be identified.

Metamorphic Relation 3: One-dimensional increment relation

Description: Add a constant to only length or width and check if the results are consistent with the addition of that constant in the original results.
Intuition: This helps detect if mutants produce consistent results under one-dimensional increment. For instance, if adding 1 to one dimension changes the output in a consistent manner, it should be reflected in the results.

The following table describes the effectiveness of each Metamorphic Relation.

Mutant ID	MR1 (C = 3)	MR2	MR3 (C = 1)
	(W + C) * (L + C)	W * L	L * (W + C)
1	Killed	Killed	Killed
2	Killed	Killed	Killed
3	Killed	Killed	Killed
4	Killed	Killed	Killed
5	-	-	Killed
6	-	-	Killed
7	Killed	Killed	Killed
8	Killed	Killed	Killed
9	Killed	Killed	Killed
10	Killed	Killed	Killed
11	Killed	Killed	Killed
12	Killed	Killed	Killed
13	Killed	-	Killed
14	Killed	-	-
15	Killed	Killed	Killed
16	Killed	Killed	Killed
17	Killed	Killed	Killed
18	Killed	Killed	Killed
19	Killed	Killed	Killed
20	Killed	Killed	Killed
Score	**90%**	**80%**	**95%**

Test cases design:

We will have a large square as our selected input:

ID	Width	Length	Description
2	10	10	Large Square

MR1: Addition Relation

- Source test case input: Width = 10, Length = 10
- Source test case output: Area = 100
- Follow-up test case: Width = 13, Length = 13 (Addition to both dimensions)
- Check that the area of new square is different from area of all mutants

MR2: Swapping Relation

- Source test case input: Width = 10, Length = 10
- Source test case output: Area = 100
- Follow-up test case: Width = 10, Length = 10 (Swapping both dimensions)
- Check that the area of new square is different from area of all mutants

MR3: One-dimensional increment relation

- Source test case input: Width = 10, Length = 10
- Source test case output: Area = 100
- Follow-up test case: Width = 10, Length = 11 (Addition to one dimension)
- Check that the area of new square is different from area of all mutants

Test cases execution using script:

```
                    f"Width Incremented: {incremented_width_mutant_result}, Result: Survived")

# Run the tests
if __name__ == "__main__":
    print("Running Addition Relation Tests:")
    test_addition_relation()
    print("\nRunning Swapping Relation Tests:")
    test_swapping_relation()
    print("\nRunning One-Dimensional Increment Relation Tests:")
    test_one_dimensional_increment_relation()

PROBLEMS  OUTPUT  TERMINAL  PORTS  AZURE

Task_1$ python rectangle.py
Running Addition Relation Tests:
Addition Relation Test case: 1, Length: 10, Width: 10, Follow-up value: 169, Mutant 5: 169, Result: Survived
Addition Relation Test case: 1, Length: 10, Width: 10, Follow-up value: 169, Mutant 6: 169, Result: Survived

Running Swapping Relation Tests:
Swapping Relation Test case: 1, Length: 10, Width: 10, Follow-up value: 100, Mutant 5: 100, Result: Survived
Swapping Relation Test case: 1, Length: 10, Width: 10, Follow-up value: 100, Mutant 6: 100, Result: Survived
Swapping Relation Test case: 1, Length: 10, Width: 10, Follow-up value: 100, Mutant 13: 100, Result: Survived
Swapping Relation Test case: 1, Length: 10, Width: 10, Follow-up value: 100, Mutant 14: 100, Result: Survived

Running One-Dimensional Increment Relation Tests:
One-Dimensional Increment Relation Test case: 1, Length: 10, Width: 10, Follow-up value: 110, Mutant 14: Width Incremented: 110, Result: Survived
Task_1$
```

Figure 5: Test cases execution using script.

Below is the information on the testing in table.

MR ID	Follow-up area	M5 output	M6 output	M13 output	M14 output
MR1	169	169	169	130	130
MR2	100	100	100	100	100
MR3	110	100	121	100	110

Conclusion

Metamorphic testing has proven highly effective in evaluating the rectangle_area function's mutants. The Addition Relation (MR1) achieved a 90% mutation score, showing strong effectiveness in detecting inconsistencies when adding a constant to both dimensions. The Swapping Relation (MR2) obtained an 80% mutation score, indicating a good but slightly less effective ability to identify mutants that fail to handle dimension swaps correctly. The One-Dimensional Increment Relation (MR3) excelled with a 95% mutation score, highlighting its exceptional capability in revealing mutants that do not consistently handle one-dimensional increments. With an average mutation score of 89%, these results underscore the robustness of metamorphic testing in ensuring function correctness and detecting mutant inconsistencies.

Resources:

Source code and mutation testing script can be found via this link.