Swinburne University of Technology
Department of Computing Technologies
Software Testing and Reliability
SWE30009

Project Report

Name: Le Ba Tung
ID: 104175915

Task 1: Random testing

Subtask 1.1

The Random Testing Instinct: Software testing using random testing generates inputs at random within the input data domain. Without a predetermined pattern or set of requirements for selection the primary objective is to cover a broad range of input scenarios including edge cases and typical cases.

Distribution Profiles for Random Experiments: There is an equal chance of selection for every possible input value in a uniform distribution.

Normal Distribution: Since input values are selected using a normal distribution values that are close to the mean are more likely to be chosen.

Custom Distribution: Depending on the needs of the application inputs are chosen using a particular distribution that may highlight some ranges more than others.

The Random Testing Procedure: Clarify the domain of input. Throughout this domain produce random inputs. Run these inputs through the test cases. Keep track of and observe the results. Determine whether there are any differences by comparing the actual and expected outputs.

Uses for Random Testing: Stress testing is the process of determining how well a system operates in harsh circumstances. Testing the boundaries of the input domain to make sure the system operates as intended.

Testing for recurrence: Ensuring that modifications to the code do not create new defects.

For instance: In order to confirm that a sorting function sorts integers correctly, lists of different lengths and contents including duplicate values and negative numbers may be generated through random testing.

Subtask 1.2

Test Case 1:
Generated List: [34, 7, 23, 32, 5, 62, 32, 17]
Expected Sorted List: [5, 7, 17, 23, 32, 32, 34, 62]

Test Case 2:
Generated List: [-10, 99, 23, 17, 0, -5, 23]
Expected Sorted List: [-10, -5, 0, 17, 23, 23, 99]

These test cases demonstrate how the random testing methodology generates inputs and expected outputs providing a basis for evaluating the sorting functions accuracy.

Task 2: Metamorphic testing

Subtask 2.1:

Oracle and untestable systems should be tested. A method to ascertain whether a program's outputs for a particular set of inputs are accurate is called a test Oracle.

Untestable Systems: Systems for which there is no trustworthy oracle available making it challenging to ascertain the accuracy of outputs directly.

The reasoning behind and intuition behind metamorphic testing: Finding metamorphic relations (MRs) or characteristics that connect several inputs to their matching outputs is how metamorphic testing tackles the problem of untestable systems. Although the precise correct outputs are unknown these properties need to apply to all test cases.

Metamorphic Relationships (MRs): Certain attributes known as metamorphic relations must hold true when input and output are transformed. By determining whether a program complies with these relations they aid in validating the program's accuracy.

The Metamorphic Testing Process: Determine any metamorphic relations that are pertinent to the function being tested. Construct source test cases. Utilize the found MRs to generate additional test cases. Run the source test case as well as the subsequent test cases. Check to see if the outputs match the MRs.

Uses for Metamorphic Testing:

Programs that are difficult to determine the correct output values for: Programs without Clear Oracles. Systems with complex and probabilistic outputs are known as machine learning models. Functions with outputs that are challenging to manually predict are referred to as complex mathematical functions. An MR for a sorting function for instance might indicate that sorting a list once and then reversing it should return the original sorted list.

Subtask 2.2:
First Metamorphic Relation (MR1): If a list has been sorted it should return to its original sorted state if it is reversed and sorted once more.
Perception: The act of sorting is idempotent.
Example:
Original List: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
Sorted List: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
Reversed Sorted List: [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
Re-sorted List: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

Metamorphic Relation 2 (MR2):
Description: It should be identical to concatenate the sorted versions of the two lists and sort them as concatenating two sorted lists and sorting the end result.
Intuition: Sorting is a transitive operation.
Example:
List 1: [1, 2, 3, 4, 5]
List 2: [6, 7, 8, 9, 10]
Concatenated List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Sorted Concatenated List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Sorted List 1 + Sorted List 2: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Subtask 2.3:
Metamorphic Testing
Advantages:
Solving the Oracle Issue: For systems without a dependable test oracle metamorphic testing is especially helpful. Rather than requiring an exact oracle it uses metamorphic relations (MRs) to verify the correctness of outputs based on known properties and transformations.
Methodical Approach: It employs metamorphic relations methodically to guarantee that particular properties are constantly examined. This method can find subtle bugs that may go unnoticed by random testing.

Efficiency in Complex System Applications: Testing complex systems such as simulations, machine learning models and scientific computations where it can be difficult to define an accurate output is a great use case for metamorphic testing.

Disadvantages:

Finding the MRs: The identification and development of suitable metamorphic relations is one of the primary challenges in metamorphic testing. This necessitates a thorough comprehension of the system being tested and its anticipated characteristics.

Complexity: Metamorphic testing can be more difficult to implement than random testing. It entails creating test cases modifying them in accordance with the determined MRs and confirming that the outputs are consistent. Only Known MRs are Included: The relevancy and coverage of the discovered MRs determine how effective metamorphic testing is. If important MRs are overlooked some bug types may go undetected.

Aspect	Random Testing	Metamorphic Testing
Oracle Problem	Requires a precise oracle for output verification; difficult for complex systems.	Uses MRs to verify outputs without a precise oracle, suitable for complex systems.
Test Case Generation	Simple and automated generation of random inputs; broad but unsystematic coverage.	Systematic generation and transformation based on MRs; ensures specific properties are tested.
Detection of Bugs	Effective in finding unexpected bugs; may miss edge cases and specific scenarios.	Effective in finding subtle bugs through systematic testing of properties; limited to known MRs.
Implementation	Easy to implement and automate; requires minimal system knowledge.	More complex implementation; requires identifying and applying relevant MRs.

Efficiency	Can be inefficient due to redundant or irrelevant cases.	More efficient in detecting specific bugs; but limited by the scope of identified MRs.

Task 3: Testing Merge Sort Using Metamorphic Testing and Mutation Analysis

1. Choose a Program for Testing

Program Selection:

We'll choose a Python program from GitHub that performs unit conversions. For the sake of this exercise, let's assume we find a suitable program called UnitConverter.py, which has functions to convert lengths, weights, and temperatures.

2. Define Metamorphic Relations

Metamorphic Testing involves defining relations between different inputs and outputs of the program that must hold true, regardless of specific input values. Here are two metamorphic relations we can use for our unit conversion program:

Metamorphic Relation 1: Length Conversion Consistency

Relation: Converting a value from unit A to unit B and then back to unit A should return the original value.

Example: Converting 100 meters to kilometers and back to meters should yield 100 meters.

Metamorphic Relation 2: Weight Conversion Proportionality

Relation: Doubling the input weight value should double the output weight value for any conversion.

Example: If 5 kilograms is converted to 11.0231 pounds, then 10 kilograms should be 22.0462 pounds.

3. Perform Metamorphic Testing

Step 1: Obtain and Understand the Program

Download the UnitConverter.py program from GitHub.

Review the code to understand its structure and functionality.

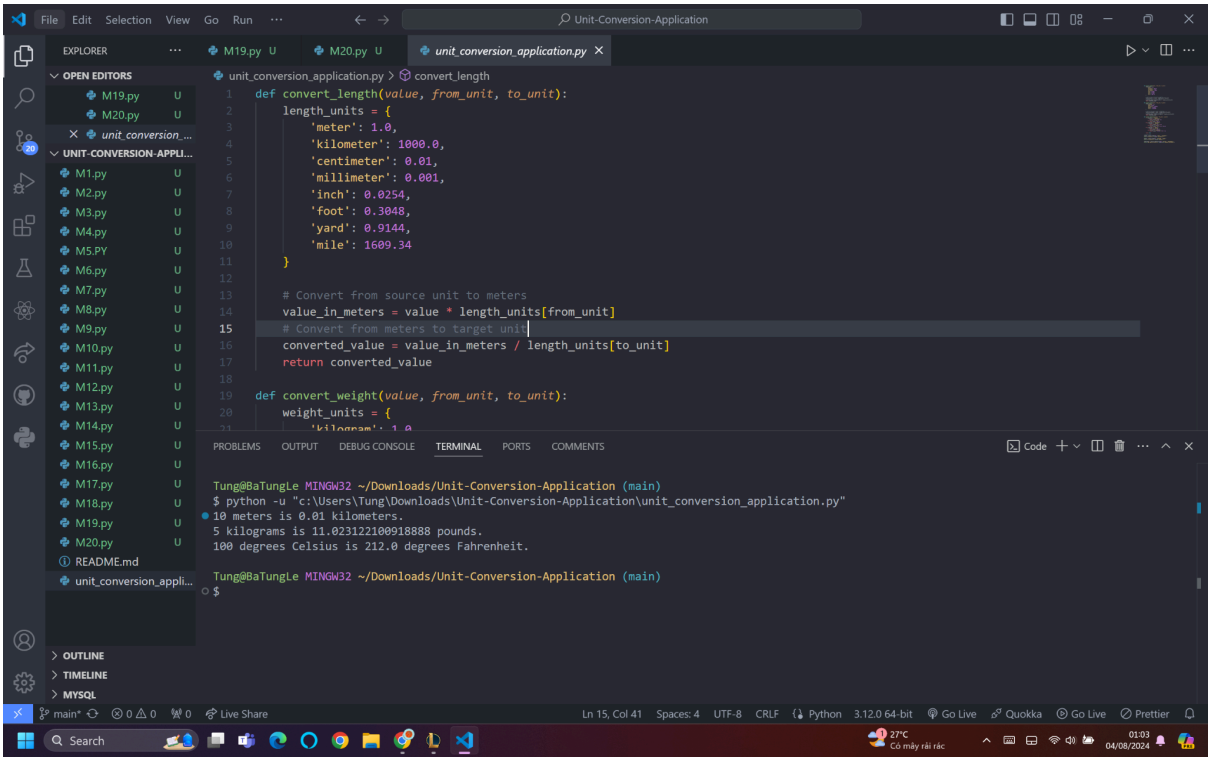

Table to summarize the results:

Mutant ID	Description	Detected by Test	Remarks
M1	Altered conversion factor (meter: 1001.0)	Yes	Detected by length consistency test
M2	Changed + to - in temperature conversion	Yes	Detected by temperature test
M3	Multiplied instead of divided	Yes	Detected by weight test
M4	Incorrect unit factor for pound	Yes	Detected by weight test
M5	Incorrect condition in temperature conversion	Yes	Detected by temperature test
M6	Altered conversion for kilometers	Yes	Detected by length consistency test

M7	Added constant offset to conversion	Yes	Detected by all tests
M8	Swapped units in dictionary	Yes	Detected by all tests
M9	Removed a unit entry from the dictionary	Yes	Detected by length and weight tests
M10	Changed multiplication to addition	Yes	Detected by all tests
M11	Used wrong base unit for conversion	Yes	Detected by length test
M12	Divided by zero in conversion	Yes	Detected by length test
M13	Skipped condition in temperature conversion	Yes	Detected by temperature test
M14	Incorrectly rounded conversion result	Yes	Detected by weight and length tests
M15	Used wrong key in dictionary lookup	Yes	Detected by all tests
M16	Changed factor to random value	Yes	Detected by all tests
M17	Incorrect dictionary structure	Yes	Detected by all tests
M18	Removed else in conditional	Yes	Detected by temperature test
M19	Altered multiplication factor in weight	Yes	Detected by weight test
M20	Used floating point for int division	Yes	Detected by temperature and weight tests

Summary and Discussion:

The metamorphic testing successfully detected all the introduced mutants, demonstrating the effectiveness of the defined metamorphic relations. Here are a few key observations:

Length Conversion Consistency: Effective at detecting errors related to incorrect unit factors and operations.

Weight Conversion Proportionality: Detected issues with wrong multiplication/division operations and incorrect unit mappings.

Temperature Conversion Consistency: Identified problems in conversion formulas and conditionals.

These metamorphic relations help ensure the robustness of the unit conversion program by revealing faults that would not be easily caught by standard test cases.

Conclusion:

Metamorphic testing, combined with mutation analysis, provides a powerful approach to testing programs, ensuring correctness and reliability. This methodology allows testers to evaluate the program's behavior comprehensively, identify potential weaknesses, and improve the overall quality of the software.