Carlton Brady
Dan Cortes
CISC615
Professor Clause
Due: 4/8/2020

Input Space Partitioning

Repository: https://github.com/CarltonTB/Input_Space_Partitioning

Interface-based approach:

Function: toRoman(int n)
Characteristic: relationship with zero
Partitioning Criteria and Blocks (block # in parenthesis):

- n less than zero
  - True (1)
  - False (2)
- n equal to zero
  - True (3)
  - False (4)
- n greater than zero
  - True (5)
  - False (6)

Input space verification: Our partitions for the interface-based approach for the function toRoman(int n) are disjoint and complete. Since our input is an integer the way we partitioned based on this syntax covers all possible values an integer can be. The disjoint property is displayed in the wording of our partitions. The values used can only be 0, >0, or <0, never a combination of the three. This proves that our answer is disjoint.

Tests:
Base choice n = 1
Tests to cover base choice criterion and all blocks:

- n = 1 covers blocks (2, 4, 5)
- n = 0 covers blocks (2, 3, 6)
- n = -1 covers blocks (1, 4, 6)

Function: fromRoman(String s)
Characteristic: length of String
Partitioning Criteria and Blocks:

- s is empty
  - True (1)

    - False (2)
- s is of length 1
    - True (3)
    - False (4)
- s is of length greater than 1
    - True (5)
    - False (6)
- s is null
    - True (7)
    - False (8)

Input space verification: Our partitions for the interface-based approach for the function fromRoman(String s) are disjoint and complete. From the syntax, we saw that the input to the function is a string. When deciding our partitions we kept in mind what properties make up a string. By translating all the properties that make up a string into partition we showed completeness. Our answer is also considered disjoint because our partitions are worded so that strings that we used can never be more than one of the partitions.

Tests:
Base Choice: s = I
Tests to cover base choice criterion and all blocks:
- S = “”: covers blocks (1, 4, 6, 8)
- S = “I”: covers blocks(2, 3, 6, 8)
- S= “IX”: covers blocks (2, 4, 5, 8)
- S = “NULL”: covers blocks (2, 4, 6, 7)

Interface-based Test Coverage:
Our tests achieved 97% (35/36) statement coverage and 71% (5/7) edge coverage. The tests did not find any errors.

Functionality-Based Approach

Function: toRoman(int n)
Characteristic: Values of Roman Numerals
Partitioning Criteria and Blocks (block # in parenthesis):

Characteristic	**b1**	**b2**
Q1 = “I”	**true(1)**	**false(2)**
Q2 = “V”	**true(3)**	**false(4)**
Q3 = “X”	**true(5)**	**false(6)**

Q4 = “L”	**true(7)**	**false(8)**
Q5 = “C”	**true(9)**	**false(10)**
Q6 = “D”	**true(11)**	**false(12)**
Q7 = “M”	**true(13)**	**false(14)**
Q8 = invalid	**true(15)**	**false(16)**
Q9 = combination of two or more roman numerals that are not invalid	**true(17)**	**false(18)**

Input space verification: Our partitions for the functionality-based approach for the function toRoman(int n) are disjoint and complete. Thinking about what the function produces we know that it displays a roman numeral. Roman numerals have 6 base values used to derive any integer given. Our partitions first each to see if those bases worked correctly. The base values are only single characters, other numbers produced use a combination of the 6 base roman numerals. Using this knowledge we made another partition that checks combinations of more than 2 roman numerals that are valid since our function has constraints. Those constraints are then checked with the last partition which is invalid. Combining all our partitions the set is considered complete because it covers every possible value a roman numeral can be. Our partitions are disjoint because all partitions are separate and one value can never be two or more partitions. The 6 base roman characters are all different from each other so they are disjoint. The single roman characters are not combinations of two or more roman numerals so our answer is still disjoint. Finally invalid syntax will never be one of the 6 base roman characters or a combination of valid roman characters . In the end our answer still displays the disjoint property even with all the different partitions.

Test:
Base Choice: n = 1
Tests to cover base choice criterion and all blocks:

- N = 1 covers blocks (1, 4, 6, 8, 10, 12, 14, 16, 18)
- N = 5 covers blocks (2, 3, 6, 8, 10, 12, 14, 16, 18)
- N = 10 covers blocks (2, 4, 5, 8, 10, 12, 14, 16, 18)
- N = 50 covers blocks (2, 4, 6, 7, 10, 12, 14, 16, 18)
- N = 100 covers blocks (2, 4, 6, 8, 9, 12, 14, 16, 18)
- N = 500 covers blocks (2, 4, 6, 8, 10, 11, 14, 16, 18)
- N = 1000 covers blocks (2, 4, 6, 8, 10, 12, 13, 16, 18)
- N = 4000 covers blocks (2, 4, 6, 8, 10, 12, 14, 15, 18)
- N = 165 covers blocks (2, 4, 6, 8, 10, 12, 14, 16, 17)

Function: fromRoman(String s)

Characteristic: Number of digits in the returned integer representation
Partitioning Criteria and Blocks:

Characteristic	**b1**	**b2**
Q1 = “1-9”	**True(1)**	**false(2)**
Q2 = “10-99”	**true(3)**	**false(4)**
Q3 = “100-999”	**true(5)**	**false(6)**
Q4 = “1000-3999”	**true(7)**	**false(8)**
Q5 = invalid	**true(9)**	**false(10)**

Input space verification: Our partitions for the functionality-based approach for the function fromRoman(String s) are disjoint and complete. First to look to see what the function will return. We know that it returns the number representation of the roman numeral given to it. Since it’s supposed to display a number, we modeled our partitions to cover the range of numbers 1-3999 that is within our constraints. Using this method helps us ensure that our partitions show completeness. Keeping our partitions disjoint we separated them into ranges to prevent partitions from overlapping. Both these properties prove our answer is complete and disjoint.

Test:
Base Choice: S = I
Tests to cover base choice criterion and all blocks:

- S = V (5) covers blocks (1, 4, 6, 8, 10)
- S = LXXVIII (78) covers blocks (2, 3, 6, 8, 10)
- S = CCCLXIV (364) covers blocks (2, 4, 5, 8, 10)
- S = MCMLXXXIV (1984) covers blocks (2, 4, 6, 7, 10)
- S = POP covers blocks (2, 4, 6, 8, 9)
- S = asd covers blocks (2, 4, 6, 8, 9)

Functionality-based Test Coverage:
Our tests achieved 100% (36/36) statement coverage and 85% (6/7) edge coverage. The tests did not find any errors.

Conclusions

Comparison to other RomanConverted Test Suite:

- **Designing tests:** Designing tests for the input space partitioning took a lot more effort compared to the first assignment with RomanConverter. For the input space, we needed to consider a lot of properties and constraints in order to come up with partitions for inputs. With two ways of performing input space partitioning a lot of tests/partitions were used to implement this method. The first assignment generated tests by looking at the statements and writing tests to get as much code coverage as possible. You could have fewer tests as long as the test itself covered more statements. The time it took to write and perform the tests only depended on how fast you could understand the function you are testing. In conclusion, input space partitioning requires a lot more knowledge about syntax and functionality than code coverage.

- **Test coverage and quality:** The test suite we created for RomanConverter as part of the first assignment achieved higher statement coverage and edge coverage than the test suite we created for this assignment. However, we were specifically designing tests in the first assignment to meet node and edge coverage criteria, so it's unsurprising that the first test suite was better along those metrics. The test suite we created during this assignment gives us more confidence in the code though. The functionality-based approach caused us to come up with test cases that do a better job of covering the full variety of possible inputs, and having the requirements of the program well specified in this assignment allowed us to design tests that give us 100% confidence that the program meets those specifications. Overall, we believe this test suite is better than the suite we created for the first assignment.