DIT635 - Assignment 2: Functional Testing, Structural Testing, and Data Flow

Due Date: Sunday, March 1, 23:59 (Via Canvas)

There are four questions, worth a total of 100 points. You may discuss these problems in your teams and turn in a single submission for the team (zipped archive) on Canvas. Answers must be original and not copied from online sources.

Cover Page: On the cover page of your assignment, include the name of the course, the date, your group name, and a list of your group members.

Peer Evaluation: All students must also submit a peer evaluation form. This is a seperate, individual submission on Canvas. Not submitting a peer evaluation will result in a penalty of five points on this assignment.

Problem 1 (24 Points)

For the GNU tail utility (described below):

1. Identify the parameters that can be controlled through testing.
2. Identify testing categories (controllable items) for each parameter.
3. Identify representative input values for each category.
4. Identify semantic constraints (if-constraints, ERROR, SINGLE) suitable for generating a set of test case specifications.

See page 187 in the textbook for an example solution to a similar problem discussed in the book. See Exercise Session 3 for another example of this process

Note – when a flag uses capital letters for input (for example, --pid=PID), that means that the user supplies a value. When lower-case is used (for example, --follow=name), that means the literal word is used (name, in this case).

Tail Documentation

tail - output the last part of files

tail [OPTION]... [FILE]...

Print the last 10 lines of each FILE to standard output. With more than one FILE, precede each with a header giving the file name. With no FILE, or when FILE is -, read standard input.

OPTIONS

- --bytes=K
    - output the last K bytes; or use --bytes +K to output bytes starting with the Kth of each file
- --follow[={name or descriptor}]
    - output appended data as the file grows; an absent option argument means 'descriptor'
- -F
- same as --follow=name --retry
- --lines=K
    - output the last K lines, instead of the last 10; or use --lines +K to output starting with the Kth
- --max-unchanged-stats=N
    - with --follow=name, reopen a FILE which has not changed size after N (default 5) iterations to see if it has been unlinked or renamed (this is the usual case of rotated log files); with inotify, this option is rarely useful
- --pid=PID
    - with --follow, terminate after process ID, PID dies
- --quiet
    - never output headers giving file names
- --retry
    - keep trying to open a file if it is inaccessible
- --sleep-interval=N
    - with --follow, sleep for approximately N seconds (default 1.0) between iterations; with inotify and --pid=P, check process P at least once every N seconds
- --verbose
    - always output headers giving file names
- --help
    - display this description text and exits
- --version
    - output version information and exit

If the first character of K (the number of bytes or lines) is a '+', print beginning with the Kth item from the start of each file, otherwise, print the last K items in the file. K may have a multiplier suffix: b 512, kB 1000, K 1024, MB 1000*1000, M 1024*1024, GB 1000*1000*1000, G 1024*1024*1024, and so on for T, P, E, Z, Y.

With --follow, tail defaults to following the file descriptor, which means that even if a tail'ed file is renamed, tail will continue to track its end. This default behavior is not desirable when you really want to track the actual name of the file, not the file descriptor (e.g., log rotation). Use --follow=name in that case. That causes tail to track the named file in a way that accommodates renaming, removal and creation.

Problem 2 (30 Points)

Consider the C program provided below. It is an implementation of a weighted-round-robin (WRR) resource allocation algorithm. The program takes the number of clients (n) and their individual weights, $\{w_i \mid 1 \leq i \leq n\}$ as input and determines a weighted-round-robin schedule for allocating some shared resource to the clients.

The output is a sequence of (repeated) client numbers (from the range 1….n) to be allocated in order, one unit of the shared resource.

1. Show the control-flow graphs for the three procedures in the program.
2. Provide a test suite that achieves statement coverage, but not branch coverage.
3. Provide a test suite that achieves branch coverage, but not condition coverage.
4. Provide a test suite that achieves condition and branch coverage. What additional test cases, if any, do you need for MC/DC coverage?
5. Do the above test suites (taken together) achieve loop boundary coverage for all the loops in the program? If yes, identify the test cases that achieve this for each of the loops. If not, provide additional test cases to achieve loop boundary coverage (zero, one, and 2+ through the loop).

If any of these coverage metrics have infeasible test obligations, justify why that is the case. Feel free to try executing the code, with any additions that you may need, to check whether your tests achieved the desired level of structural coverage.

```c
#include <stdio.h>
#include <stdlib.h>
#define SWAP(a,b) { int _tmp = a; a = b; b = _tmp; }

int cur_count = 0, alloc_count = 0, total_count = 0, num_elems = 0;
int *counts, *psums, *elems;

void initialize(int n, FILE *fptr) {
    int i = 0;
    num_elems = n;
    counts = (int *) calloc(n, sizeof(int));
    psums = (int *) calloc(n, sizeof(int));
    elems = (int *) calloc(n, sizeof(int));
    for (i = 0; i < n; ++i) {
        elems[i] = i + 1;
        fscanf(fptr, "%d", &counts[i]);
        psums[i] = total_count += counts[i];
    }
}

int next_elem() {
    int top = num_elems - 1;
    int ret = elems[top];
    if (alloc_count >= total_count)
        return 0;
    ++alloc_count;
    ++cur_count;
    --psums[top];
    --counts[top];
    while (top > 0 && counts[top] <= (cur_count * psums[top - 1])) {
        psums[top - 1] += (counts[top] - counts[top - 1]);
        SWAP(counts[top], counts[top - 1]);
        SWAP(elems[top], elems[top - 1]);
        --top;
    }
    if (top != num_elems - 1)
        cur_count = 0;
    return ret;
}

int main() {
    int next;
    printf("number of elements: ");
    scanf("%d", &num_elems);
    printf("\nweights:\n");
    initialize(num_elems, stdin);
    printf("\n");
    while ((next = next_elem()) > 0)
       printf("%d ", next);
    printf("\n");
}
```

Problem 3 (22 Points)

The following code is a Java function that does insertion sort.

```
public static void InsertionSort(int[] num)
{
       int j;   // the number of items sorted so far
       int key; // the item to be inserted
       int i;

       for (j = 1; j < num.length; j++)    // Start with 1 (not 0)
       {
             key = num[j];
```

```java
                // Smaller values are moving up
                for(i = j - 1; (i >= 0) && (num[ i ] < key); i--)
                {
                        num[i+1] = num[i];
                }
                num[i+1] = key;    // Put the key in its proper location
        }
}
```

1. Identify the def-use pairs for all the variables. Explain how you handle the array variable.
Hint: arrays are treated like pointers in Java.
2. Provide a test suite that achieves all def-use pairs coverage. If there are def-use pairs that cannot be covered, explain why.

Problem 4 (24 Points)

In a directed graph with a designated exit node, we say that a node m post-dominates another node n, if m appears on every path from n to the exit node.

Let us write *m pdom n* to mean that m post-dominates n, and *pdom(n)* to mean the set of all post-dominators of n, i.e., *{m | m pdom n}*.

Answer the following, providing justification for each:
1. Does *b pdom b* hold true for all b?
2. Can both *a pdom b* and *b pdom a* hold true for two different nodes a and b?
3. If both *c pdom b* and *b pdom a* hold true, what can you say about the relationship between c and a?
4. If both *c pdom a* and *b pdom a* hold true, what can you say about the relationship between c and b?
5. How would you use the answer to your previous question to characterize the immediate post-dominator of a node other than the exit node?
(Hint: immediate post-dominator is, in some sense, the “nearest post-dominator” of a node)
6. Suppose the set of post-dominators *pdom(n)* is known for every successor n of a node m, can we then derive *pdom(m)*?
7. Cast the computation of *pdom(m)* in terms of flow equations. What are the gen and kill sets for m? How would you classify the flow analysis: forward or backward, any-path or all-path?
8. Provide an iterative work-list algorithm (see Chapter 6 in the textbook and Lecture 9 for examples) for computing the post-dominance relation based on the flow equations.