William Leslie

CS 362

Dominion Final Test Report

For all the tests I ran for this report, I used random testing based off of Prof. Groce’s testDrawCard program. testDrawCard is largely unchanged from the original implementation, so it’s construction will not be spoken of in great detail. The buyCard test is very similar to the drawCard test with a few minor changes. The main function sets random numbers for the player whose turn it is, the amount in coins that player has, the number of buys allowed for that player, the number of cards in the discard pile, and the number of cards available of that type. The tester is then called with the player number, the card in question, and the game state along with a message telling what the program will test for. The test itself makes a copy of the current game state and will compare it to the one after buyCard is run. To compare this, there is a field of ifs and asserts as follows:

if(r != 0) printf("did not return 0\n");

assert (r == 0);

if(post->coins > pre.coins) printf("coins are greater than when started\n");

assert(post->coins <= pre.coins);

if(post->numBuys != pre.numBuys-1) printf("numBuys did not decrement\n");

assert(post->numBuys == pre.numBuys-1);

if(post->supplyCount[card] != pre.supplyCount[card]-1) printf("supplyCount did not decrement\n");

assert(post->supplyCount[card] == pre.supplyCount[card]-1);

if(post->discardCount[p] != pre.discardCount[p]+1) printf("discard did not increment\n");

assert(post->discardCount[p] == pre.discardCount[p]+1);

This insures that the 2 game states are the same, but will print out any inconsistencies to the output file. The testAdventurer program is set up similarly to what I have just described above with the differences that it will fill up the deck with random cards to test the drawing feature of the adventurer card. It has the following checks in place:

if(r != 0) printf("did not return 0\n");

assert (r == 0);

if(post->deckCount[p] < 0) printf("deckCount is negative\n");

assert (post->deckCount[p] >= 0);

if(post->discardCount[p] < 0) printf("discard is negative\n");

assert (post->discardCount[p] >= 0);

if((pre.handCount[p] + 2) != (post->handCount[p])) printf("handcount is not incremented\n");

assert ((pre.handCount[p] + 2) == (post->handCount[p]));

if((pre.numActions - 1) != (post->numActions)) printf("actions did not decrement\n");

assert ((pre.numActions - 1) == (post->numActions));

To view the output statistics I created a testdom script similar to the one Prof. Groce created in class. This script outputs the date and time and all the outputs of the tests to test.out. At the end of the script I have gcov run on dominion.c to tell how much code coverage I got from running these tests.

I started these tests on my own implementation of dominion.c. The test started with the drawCard test. There were no segmentation faults detected in this portion of the code, but some strange outputs did show up. Most of the “current hand count” outputs were in the hundreds, which is an unreasonable number for a normal game of dominion. This would indicate that the number of draws does not increment properly, either more cards than expected were drawn or, more likely, some bogus values are being fed in somehow.

Then the script moved on to the buyCard test. The test attempted to buy card number 10 for 2 coins and succeeded at running the program, but failed the check that the number of buys after the program ran would be decremented by 1. The test soon after aborted and I have yet to figure out the cause of this problem. After this I decided to view the code by hand to see if there were any obvious problems. I noticed that updateCoins is used twice in the program where it probably is not necessary. I also noticed that I had written an “if” without a logical else. At this point there is a conditional that says the cost of the card must be less than the amount in coins the player has. This may be the cause of many of the issues in the test; the function is not programmed to fail if it is impossible.

Next I moved on to the adventurer test. The output was similar to the output of the buyCard test, where the tester just stops after it outputs the fact that the number of cards in the player’s hand did not increment by 1. I proceeded to also check this one by hand, but could not find any glaring flaws. So I checked the coverage status with gcov. Gcov indicated that nearly 8% of the 553 lines of code in dominion were covered.

I then moved on to my fellow group members’ code. First I tested ellingsn’s code. This code had almost identical results to when I tested my own code. The gcov results state that over 8% of the 520 lines of code were covered in the tests. I then looked at the code only to find that the code did not change from the last time I looked at it and, in fact was nowhere close to being finished.

Then I tested omaraa’s code. The code worked similarly in drawCard, but was different in everything else. The tester dies after the program finds that the test case does not have enough money to buy the card in question and stats that the assert for r returning 0 failed. Adventurer segfaults under my test. Code inspection of buyCard and adventurer look ok on this implementation. Gcov indicates that over 8% of 537 lines were covered.

Then I tested taylodav’s code. The test also died in this one due to the fact that num buys did not decrement. Unfortunately, I realized this was because of the fact that the random test was always spitting out an invalid combination where the card could not be bought. I could not figure out how to fix these problems though. During code inspection it looked fine, though. The test took too long on the adventurer implementation which may indicate an infinite loop, so I terminated the program. Not statistics showed up on the output file.

At this point I realized that my testers were fundamentally flawed, while they could indicate whether an operation failed, it could not show much since it could not make it passed an initial pass of the code.

If I had more time to work on this project, I would completely re-write the code. Instead of using random testing, I would just test a few canned cases that the game might run into, that way I would have a better idea of what was going on, seeing that the test would not just die. I would have also used CBMC but did not have enough time to play with it. CBMC would have been good for proving the correctness of the code in question mathematically. In this case hand inspection was invaluable not only in finding problems with the code, but also finding bugs in the tester. The few tools I did use were also invaluable: gcov was consistent and the C compiler was able to tell me were segfaults and other errors occurred.