Lakshman Madhav Kollipara
ONID- kollipal
CS562_Applied Software Engineering

Differential Testing Using Harness Maker

As per paper published by Dr. Alex Groce and Dr.Jervis Pinto
"A Little Language for Testing"

Using the harness maker tool mentioned in the paper, we have done differential random testing on my dominion and other dominion.

Steps:
1. Write an ACT file: test_dom.act
2. Compile the ACT File using harness-maker: python2.7 harnessmaker.py -a ACT file -t TARGET File -c CLASS name --OPTION
3. Get sut.py as output: sut.py
4. Run random test on sut.py: randomtest.py

Working with ACT file:

TSTL:
TSTL variable: Any valid python identifier with a % before and after it.
Tags: @, pool:, source:, reference:, compare:
Action line: Any line that doesn't start with a tag.

Pool variable: Any variable that is declared using the pool tag. Ex.: `pool: %N% 2`. This means there are 2 %N%'s p_N[0] and p_N[1]

%N% := %[1..10]%
This says the pool variable %N% takes values from the pool on the RHS.

Action: Whenever TSTL sees an action line and the action line uses a pool variable. Then an action is created for each variable in the pool. Whenever an action uses a pool value. That action is applied to all values in the pool.

Reference: pool: %N% 2 REF
Means there are 2 values for %N%. REF tells that this is a reference variable. Which means whenever there is an action using %N% there should be two things that should hold:
When you use REF, reference tag should exist, which is applied to both the value and the reference value. The function and reference function both are applied on each value of N in the pool.
Reference: function1 ==> function2, Acts like REGEX replace, Replaces LHS of Arrow with RHS of Arrow

Compare: Used to compare Reference variables.

Property: Tells the restrictions imposed by the system. If the restrictions are violated during the course of actions then there would be an assertion error.

check(): Checks if all properties are correct. So the assertion check does not happen magically. In the code where we test our SUT we have to call t.check().

log: adds functionality

:= Operator: Does not let you assign new values again unless you have used the variable assigned. Basically the assignment action becomes disabled after it occurs until the variable is used.

Tilde(~): TSTL won't discard that the value of that variable after taking action, (~%GAME%) means don't throw away the %GAME% variable, as we will reuse it for other calls.

Compiling ACT File:

I encountered mainly three errors: IndexError, AttributeError, AssertionError.
Most failed tests were similar and due to something wrong in function code and random tests on act file generated a huge log of similar errors which were redundant. Random testing on SUT easily creates the bug, but sometimes the sequence of steps performed to get the bug is large. so it is hard to know which part of dominion code causes the bug exactly. I.e. I found solving the Fault localization problem hard.
In playCard there were not many AssertionError type bugs probably because most of them were IndexError, or of AttributeError types.
I got an AssertionError in scoreFor, which helped me confirm that the score calculation in my dominion and other's dominion is not matching.

I have come to the conclusion that using completely random testing with this system is not particularly useful, especially with the generalized rules that I used. To further drive this point home, after reporting the coverage over the files in a testing run, one file only had a coverage of about 15%. This is extremely low and therefore unhelpful for testing the vast majority of the program's functionality.

In order to compare the results of two functions, which have the exactly same number of parameters, but have different types, it is necessary to either make some modifications on the original code, or write an additional adapter to guarantee that they have the same parameter types. According to this experience, one of the preconditions of put the current version of TSTL into use of the differential testing is to guarantee that SUTs are implemented based on the same set of APIs.

The paper, A Little Language for Testing, provides some knowledge and useful sample codes about how to use TSTL. If without the limitation of space, I think there should be more information to introduce the detailed workflow behind the scene.

Lakshman Madhav Kollipara
ONID- kollipal
CS562_Applied Software Engineering

Errors

Another problem with this random differential testing is that both code bases are not fully bug free. This creates problem like deciding on which version is the bug in. But the bigger problem is when both versions are wrong in the same way. There is no way to catch that using differential testing.

In drawCard deckCount is not defined, which should be g.deckCount.

In the other code, endTurn did not include drawing cards for the player whose turn just ended. So,after one round of each player's turn, the game cannot run again since all the player's hand is empty.It does not update Coins for the next player too. These gave assertion errors while trying to buy a card after ending turn.

After playCard, we need to updateCoins so that we can use the most recent value for buying a card. I had not included that in my playCard function, which led to assertion error while buying a card in the next step.

Error in Return Value in scoreFor:

Wrong score calucaltion

```
File "randomtest.py", line 63, in <module>
    action()
  File
"/home/hduser/lakshman/cs562w15/kollipal/dominion_Assignment2/harness/
sut.py", line 748, in act54
    assert __result == __result_REF, " (%s) == (%s) " % (__result,
__result_REF)
AssertionError:  (0) == (1)
```

Error in Return value fr getWinners:

Because of wrong score calculation

```
File "randomtest.py", line 63, in <module>
    action()
  File
"/home/hduser/lakshman/cs562w15/kollipal/dominion_Assignment2/harness/
sut.py", line 779, in act56
    assert __result == __result_REF, " (%s) == (%s) " % (__result,
__result_REF)
AssertionError:  ([1, 1, 0, 0]) == ([1, 0, 0, 0])
```