CS569
Instructor: Professor Alex Groce
Student: Kazuki Kaneoka
ID: 932-277-488
Email: kaneokak@oregonstate.edu

Part 4
Final Doc
Feedback-directed Random Test Generation* in *TSTL

Introduction:
In this document, I would like to explain what *Feedback-directed Random Test Generation* [1] is, how I implemented it in *TSTL*, and how the evaluation it was.

***Feedback-directed Random Test Generation*:**
Feedback-directed Random Test Generation (*FRT*) is a type of Random Test Generations for automated software testing. It contains two unique concepts using runtime information: creating sequence incrementally and checking contracts and filters.

- Creating sequence incrementally:
  First of all, sequence in *FRT* means a sequence of calling methods with input arguments. Therefore, creating sequence incrementally means:
  1. Randomly pick up one or multiple methods with corresponding input arguments to generate sequence.
     NOTE: input arguments are stored in component. Initially, component has very few primitive types. While running the algorithm, component is continuously updated by adding the result data (runtime information) from executing method.
  2. Append this sequence to the previous sequence to create a new sequence.
     NOTE: Assume we store many previous sequences. The previous sequence is also selected randomly from them to create a new sequence.
  3. Check whether this new sequence is already created before or not. If it is, ignore the sequence and create another one. Otherwise, succeed to create a new sequence.

  By this way, *FRT* can avoid redundant sequences, which Random Test Generation usually produces.

- Checking contracts and filters:
  Contract is the property that method or object should always hold. For example, Object *o* should always return *true* for *o equals o*. AVL is always height balanced.
  Filter is the evaluation for whether a new sequence, which *FRT* creates incrementally, should be stored into previous sequences or not. *FRT* executes a new sequence and check the result of execution (runtime information) to determine it.
  NOTE: a new sequence is stored into previous sequences means that *FRT* uses this sequence to create a future sequence.

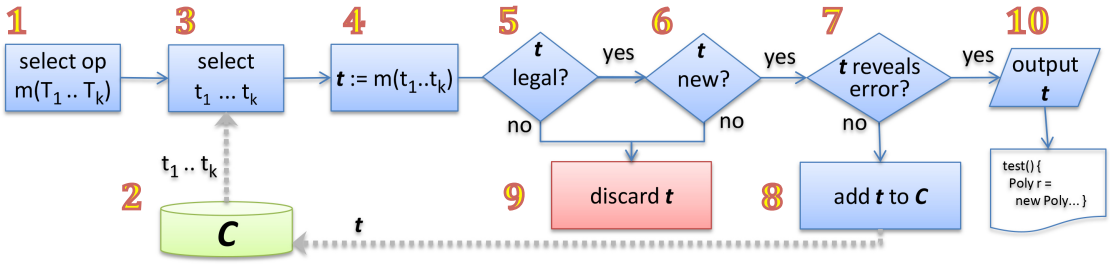
How I Implemented *Feedback-directed Random Generation* in *TSTL*:
Basically, I implemented *Feedback-directed Random Generation* (*FRT*) in *TSTL* followed by *GenerateSequences* in Figure 3 in the paper, *Feedback-directed random test generation* [1].

GenerateSequences(*classes*, *contracts*, *filters*, *timeLimit*)

```
1  errorSeqs ← {} // Their execution violates a contract.
2  nonErrorSeqs ← {} // Their execution violates no contract.
3  while timeLimit not reached do
4     // Create new sequence.
5     m(T1 ... Tk) ← randomPublicMethod(classes)
6     ⟨seqs, vals⟩ ← randomSeqsAndVals(nonErrorSeqs, T1 ... Tk)
7     newSeq ← extend(m, seqs, vals)
8     // Discard duplicates.
9     if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
10       continue
11    end if
12    // Execute new sequence and check contracts.
13    ⟨o⃗, violated⟩ ← execute(newSeq, contracts)
14    // Classify new sequence and outputs.
15    if violated = true then
16       errorSeqs ← errorSeqs ∪ {newSeq}
17    else
18       nonErrorSeqs ← nonErrorSeqs ∪ {newSeq}
19       setExtensibleFlags(newSeq, filters, o⃗) // Apply filters.
20    end if
21 end while
22 return ⟨nonErrorSeqs, errorSeqs⟩
```

Figure 3. Feedback-directed generation algorithm for sequences.

The following figure is about how *FRT* works from Slides in Randoop Publications [2].



Feedback-directed Random Test Generation works as:

1. Randomly pick up method *m* (1).
2. Pick up input arguments t_1, …, t_k for the method *m* from component *C* (2, 3).
3. Execute method *m*. *t* is a return value (runtime information) (4).
4. Check whether *t* is legal, new, or reveals error (5, 6, 7).
5. Accordint to *STEP 4* (runtime information), *t* is added to *C*, be discarded, or output (8, 9, 10).

The *main* function of my implementation:

```
01 while time.time() - start < timeout:
02     seq = rgen.choice(nseqs)[:]
03     sut.replay(seq)
04     if rgen.randint(0, 9) == 0:
05           n = rgen.randint(2, 100)
06           ok, propok, classTable =
                                    appendAndExecuteSeq(seq, n, eseqs, nseqs)
07     else:
08           ok, propok, classTable =
                                    appendAndExecuteSeq(seq, 1, eseqs, nseqs)
09     if ok and propok and filters(classTable):
10           nseqs.append(seq)
```

The *main* function works as:

1. While not time out, repeat *STEP 2 - 4* (line 01).
2. Randomly pick *seq* from *nseqs* and replay it (line 02 - 03),
   where *seq* is list of actions and *nseqs* is list of non-error *seqs*.
3. Create a new *seq* incrementally by generating and executing a *seq* with length *n*. Then, append it into the *seq*, which is selected at *STEP 2* (line 04 - 08):
   $n = 1$ for 90% and n = between 2 and 100 in equally likely for 10%.
4. Check whether the new *seq*, which is just created at *STEP 3*, should use for next loop or not. If *True*, put it into *nseqs* (line 09 - 10),
   where *nseqs* is list of error *seqs*.

The *appendAndExecuteSeq* function of my implementation:

```
01 def appendAndExecuteSeq(seq, n, eseqs, nseqs):
02     ok = False
03     propok = False
04     classTable = dict.fromkeys(sut.actionClasses(), 0):
05     while n > 0:
06           n -= 1
07           a = sut.randomEnabled(rgen)
08           seq.append(a)
09           if equal(seq, eseqs) or equal(seq, nseqs):
10                 continue
11           ok = sut.safely(a)
12           propok = sut.check()
13           classTable[sut.actionClass(a)] += 1
14           if contract(ok, propok, seq, eseqs):
15                 break
16     return (ok, propok, classTable)
```

The *appendAndExecuteSeq* function works as:

1. Initialize return values (line 02 - 04).
2. While $n > 0$, repeat *STEP 3 – 7* (lien 05).
3. Randomly pick up an action and append it into *seq* (line 06 - 07).
4. Check whether this *seq* is already generated before or not (line 09 - 10).
5. Check whether the action, which is selected at *STEP 3*, is executed without any violation or not (line 11 - 12).
6. Update *classTable* (line 13),
   where it is hash table using *actionClass* as key and its frequency as value.
7. If there is violation at *STEP 5*, put the *seq* into *eseqs* (line 14).

Evaluation:
Environment:

- Macbook Air
- Processor 1.7 GHz Intel Core i7
- Memory 8 GB 1600 MHz DDR3
- OS X Yosemite Version 10.10.5

I ran *finaltester.py* and *mytester.py* with AVL and MySQL Parser. *finaltester.py* and *mytester.py* were slightly different. As I mentioned in the previous section, I implemented *finaltester.py* based on *GenerateSequences* in Figure 3 in the paper, *Feedback-directed random test generation* [1]. However, it lacked one feature of *FRT*, which was component. *finaltester.py* picked up method with corresponding parameter as action from *sut.enabled* function instead of getting from component. On the other hand, *mytester.py* generated component by adding actions for each *sut.poolUses* first, and then, pick up action from component.

Experiment 1 with AVL:

- Files:
  avl.py under *cs569sp16/SUTs/* directory
  avl.tstl under *cs569sp16/SUTs/* directory

I ran *finaltester.py* with 300 seconds for 10 times. The statistic results were:

Bug Detect (%)	Mean (seconds)	SD (seconds)	Max (seconds)	Min (seconds)
1.00	184.95	5.12	196.88	180.02

I ran mytester.py with 300 seconds for 10 times. The statistic results were:

Bug Detect (%)	Mean (seconds)	SD (seconds)	Max (seconds)	Min (seconds)
1.00	291.90	2.30	294.45	287.00

Both of them, the statistic results for coverage were:
Branch coverage was: 190
Statement coverage was: 141

The length of sequence, which detected bug were:
91 for *finaltester.py*
79 for *mytester.py*

Comments for Experiment 1:
Both *finaltester.py* and *mytester.py* could detect bug if they spent enough time. It was feature of *FRT* because *FRT* created sequence inclemently. It was also understandable *finaltester.py* was faster than *mytester.py* to detected bug because *finaltester.py* did not utilize component, but *mytester.py* did. Furthermore, *mytester.py* needed shorter length of sequence to find bug since *mytester.py* had fewer redundant by component.

In addition, I tried to pick a good sequence by scoring each sequence and selecting the best one always. I tried 2 ways of how to score each sequence:

1. $coverage\ covered\ by\ the\ sequence / time\ of\ executing\ the\ sequence$
2. $coverage\ covered\ by\ the\ sequence / length\ of\ the\ sequence$

However, picking a good sequence based on score didn't work since a good sequence and the scores had no relationships.

Experiment 2 with MySQL Parser:

- Files:
  cdefs.py under *cs569sp16/projects* directory
  mysqlparse.py under *cs569sp16/projects* directory

I ran *finaltester.py* with 300 seconds for 10 times. The statistic results were:
Branch coverage was: 512
Statement coverage was: 356

I ran *mytester.py* with 300 seconds for 10 times. The statistic results were:
Branch coverage was: 482
Statement coverage was: 340

Comments for Experiment 2:
It had low coverage for both branch and statement. During 300 seconds, it created many sequences. However, the lengths of more than half, approximately 60 % of them, were between 1 and 20. In other words, more than 60% of them was not so interesting sequence. It was the reason why FRT showed low coverage and also not finding bug.

Conclusion:
Feedback-directed Random Test Generation (*FRT*) was superior to normal random test generation in terms of avoid redundant. However, it still had same issue with random test generation such that we couldn't know which sequence was good to find bug. Both algorithms picked sequence randomly since it was difficult to identify which one detected bugs.

Another thing was that original *FRT* used component *C*, but mine did not have exactly same one. Initially, component *C* contained the minimum primitive data where we needed to execute method. Then, update component *C* by runtime information. So, we had fewer options to pick up method because of component *C*, and we had more options for method after updating component *C*. The reason why mine did not use component was that *TSTL* considered method and input arguments together as action, and we could get it by *sut.enabled* function. I doubted there were some differences between component *C* with method and actions in *TSTL*.

Reference:

[1] C. Pacheco, S.K. Lahiri, M. D. Ernst, and T. Ball. *Feedback-directed random test generation*. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE'07, pages 75-84, 2007.

[2] C. Pacheco. *Directed Random Testing*. *http://randoop.googlecode.com/files/thesis_talk_post.pdf*, pages 40, 2009.