AN ABSTRACT OF THE THESIS OF

Kazuki Kaneoka for the degree of Master of Science in Computer Science presented on March 23, 2017.

Title: Feedback-Based Random Test Generator for TSTL

Abstract approved: _____

Alex Groce

Software testing is the process of evaluating the accuracy and performance of software, and automated software testing allows programmers to develop software more efficiently by decreasing testing costs. We compared two advanced random test generators, a Feedback-Directed Random Test Generator (FDR) and a Feedback-Controlled Random Test Generator (FCR), for an automated software testing tool in Python 2.x, the Template Scripting Testing Language (TSTL).

An FDR generates test inputs incrementally. Feedback from previous trials is used to generate new inputs. As each test input is executed, the software properties are assessed to determine if there is any value. Because of this process of gradually generating new tests, the FDR avoids redundant and illegal test inputs commonly produced by traditional random test generators. An FCR employs a different feedback technique. It controls the feedback to produce varied test inputs using multiple input containers. In our experiments, we compared the performance of our test generators with TSTL's generator in terms of coverage, time-efficiency, and error-detection capability.

Feedback-Based Random Test Generator for TSTL

By

Kazuki Kaneoka

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented March 23, 2017
Commencement June, 2017

Master of Science thesis of Kazuki Kaneoka presented on March 23, 2017

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Kazuki Kaneoki, Author

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor Dr. Alex Groce of the School of Electrical Engineering and Computer Science at Oregon State University. His enthusiasm and eagerness to provide guidance whenever I needed it was of great value to my experience. I would also like to thank Xin Liu and Zixuan Zhao for their support and suggestions. Finally, I must express my profound gratitude to my family for their continuous support and encouragement throughout my studies and research. There is no doubt that I would not have accomplished this achievement without them. Thank you for everything.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 Overview

In software engineering, software testing is the examination of the properties of the software system. Generally, software testing is to establish 1) whether the software design satisfies the requirements it is supposed to be, 2) whether it computes the expected results from its inputs, and 3) what its limitations are under different circumstances. Specifically, determining the accuracy and reliability of the software system is a complex procedure. The procedure usually consists of 1) generating test case, which is the test input data to evaluate software, 2) defining the expected results of executing the test case under software, 3) executing the generated test case and then, 4) comparing the output with the defined expectation.

However, to complete the procedure by human hand is tedious and unproductive work. This is the reason why software-test automation is an active area to research because it can make the complex procedure easier and it eventually leads the software development efficiently by increasing the productivity of software development [2, 6, 10, 20]. Although automated software testing is active area in research and industry, it has faced challenging because it is necessary 1) to generate test cases automatically and 2) to set a test oracle, defined as the means of setting the expected output of a test case.

In making the process of software testing automatically, an ideal strategy would generate test cases by following some mathematical strategy instead of human hands. We can let a computer to create test cases automatically following some mathematical strategy but it should not generate test cases too much. Otherwise, the test cases cannot be executed in reasonable time, because it is impossible to generate and examine all possible test cases. Therefore, a good test case generator that generates high-quality test cases, in terms of evaluating a software system, is required for an automated software testing. Different types of generators employs different types of strategy for producing test cases and each one has distinct advantage.

A wide range of test case generators have been experimented with in the software-testing field. A Random Test Generator (RTG), one that produces test case randomly, is fundamental software-testing strategy. It is worth mentioning that 1) an RTG can be implemented rather simply, 2) it executes the system quickly, and 3) avoids programmer's bias, as well as reveals hidden information that the programmer may not recognize [1, 5]. However, it is argued that an RTG is inefficient because it requires a large amount of test case to achieve high-code coverage, meaning the identification of how much percent of source codes in software are being executed using test cases [25]. High coverage is important to software testing because bugs are never found in lines of codes that are not covered, although high coverage does not guarantee the detection of failures [13, 22].

An RTG may require more test cases to cover certain parts of the software, even if only one test case is sufficient. For example, because of its randomness, an RTG may generate multiple test cases to cover the same lines of codes. In other

words, those multiples test cases are syntactically or semantically same. In addition, an RTG may generate unnecessary test cases, meaning a test case that the software does not accept as an input. For instance, there is no need to test the binary operation of dividing by zero. One possible idea to avoid those problems are to analyze software system before generating test cases. However, it is impractical to manually analyze and obtain the necessary input data for such complex software. Thus, the consequences of random generation are considered acceptable for software testing and it is worth to improve the idea.

A Feedback-Directed Random Test Generator (FDR) was introduced to improve upon the RTG by mitigating the redundancy and illegality common to random generation [26]. With an RTG, test case is generated by the input domain with some probability distribution. An FDR, however, creates test cases incrementally through random sequences of methods produced by the input domain. The increment of the input domain can avoid generating a test case that is syntactically identical to previous test cases, and can avoid generating a test case that is semantically illegal for the program. It is a flexible technique for general software testing since the technique does not require pre-defined test cases and analyzing a software. It can grow test cases from scratch. Because of its versatility and practicality, the FDR has been widely adopted in industry and academic. Specifically, Pacheco introduced Randoop, a test generator for Java based on the concept of the FDR [27]. Randoop is a means of unit testing, defined as a strategy of software testing that assesses one or more components of the software. Furthermore, the FDR is also used in the field of software-testing research for evaluating and comparing a

researcher's work [7, 12]. Despite resolving the weakness of RTG, FDR creates another problem; the random method sequences are biased by the method selected at the beginning of the generating process because its test cases are generated by appending a new methods to previous test cases.

The Feedback-Controlled Random Test Generator (FCR) is an advanced form of the FDR [29]. The FDR can avoid generating the redundant and useless test cases of the RTG, by creating test cases incrementally; however, the generated test cases are biased by the method that is selected initially, meaning that the test cases have the same syntactic prefix. Additionally, it is impossible to determine which method is the best to begin with for detecting failures in the system. FCR approaches this problem by managing multiple test input sources. Each test input source works independently of the others to generate test cases. In other words, test case in one test input source differs from another test case in another test input source. Because of this, FCR retains the functionality of FDR and reduces bias at the same time.

In our experiments, we designed and implemented an FDR and an FCR for the Template Scripting Testing Language (TSTL), with the goal of creating an automated software testing tool in Python 2.x (there is also a beta version in Java) [17, 23]. TSTL already supports some test generators. However, since test generators work differently in different situations, having an FDR and an FCR produces a more diverse array of results in TSTL.

The rest of this paper is structured as follows. In Chapter 2, TSTL, FDR, and FCR is summarized for review. The design and implementation of an FDR and an FCR into TSTL is discussed in Chapter 3. The performance of the FDR and the FCR

compared with a random test generator in TSTL is presented in Chapter 4. Finally, our conclusion is found in Chapter 5.

CHAPTER 2: LITERATURE REVIEW

Chapter 2 presents our project's background information. It reviews an automated software-testing tool, the Template Scripting Testing Language (TSTL). Afterwards, two advanced random test generators are discussed, a Feedback-Directed Random Test Generator (FDR) and a Feedback-Controlled Random Test Generator (FCR).

2.1 The Template Scripting Testing Language (TSTL)

2.1.1 Overview

TSTL is an automated software-testing tool in Python 2.x provided by Groce and it facilitates software-testing automation by creating test harnesses for programmers [17]. Per Groce, a test harness defines as a set of test cases and a set of properties that corresponds to those test cases. An automated software testing is easy for programmers if a test harness is provided since we can simply execute a test case defined by the harness and evaluate the output. However, writing test harnesses is a daunting task, specifically for human hands [14, 15].

There are couple points why a test harness is challenging for human hands. Firstly, a test harness should be written in the same language as the Software Under Test (SUT), defined as the software being tested, but writing test harnesses in this way involves a great deal of repetition, a common source of human error. In addition, it is normal in real industry software testing for the methods of the SUT to require

complex input parameters. Manually preparing the inputs for testing the methods is a rather frustrating endeavor. The test harness must be able to adapt to multiple testing situations. In other words, programmers should easily be able to implement the harness to test the SUT using different strategies. TSTL defines and works a test harness and let programmers to focus on software testing.

TSTL produces test harnesses based on the notion of a domain-specific language (DSL) [16]. The reason TSTL supports the concept of DSL is that it can provide abstractions and notations for a specific language [11]. DSL consists of an external part, which has its own syntax of DSL, and an internal part, which is a stick with the language of the SUT. Generally, the external part is used to generate conditions that are difficult for programmers to write, and an internal part to utilize the benefit of the language under SUT.

2.1.2 The Communication between TSTL and a Test Generator

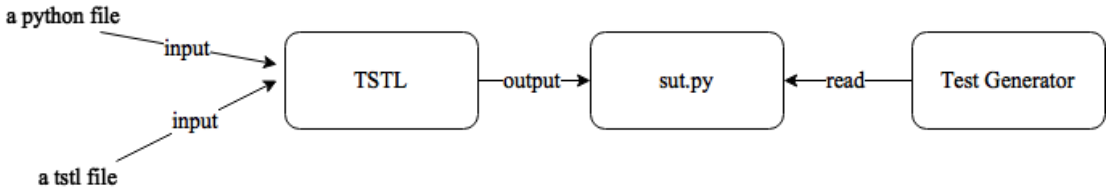

Figure 1: The Communication between TSTL and a Test Generator

TSTL provides software-testing automation by creating a test harness that a test generator utilizes to produce test cases. In *Figure 1*, we highlight how TSTL interacts with a test generator. TSTL takes two inputs, a python file, which is the SUT, and a TSTL file, which defines the SUT properties, such as 1) the possible input domain of each method, 2) the possible methods that will be used, and 3) the test oracles. By

giving a python file and a TSTL file, TSTL compiles and produces an output file, named *sut.py*, which contains all the necessary and supporting information for a test generator to generate test cases. Eventually, a test generator accesses *sut.py* to generate and execute test cases to evaluate SUT.

2.1.3 Test Cases in TSTL

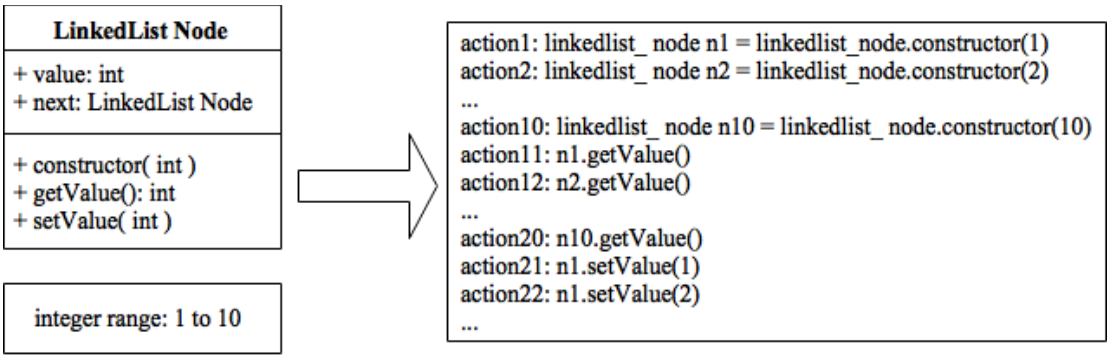

Figure 2: Test Cases in TSTL

In TSTL, a test generator produces test cases using *sut.py*; the test cases are sequences of predefined methods with an input domain, named *action*. *Figure 2* shows an example of how *action* is defined in *sut.py*. Assuming the SUT is a linked-list node implementation, the list node contains two member variables: an integer value of a node and a pointer of the next node. There are three member functions in the linked-list node: a constructor, *getValue()*, and *setNext()*. By providing some information for the input domain, such as the range of integer value being between one and ten, TSTL defines all possible *action* in *sut.py*. It should be noted that TSTL defines *action* to satisfy a property of the SUT. For instance, a constructor should be called before a setter and a getter. TSTL defines *action* to behave following this manner. In TSTL, a

test case is a sequence of *action*. Therefore, a test generator generates test cases by selecting and executing *action* from *sut.py*.

2.2 Random Test Generator (RTG)

In software testing, an RTG is considered a basic but essential technique amongst test case generators [19, 24]. It can be implemented without analyzing a sophisticated system, and can generate test cases randomly from the input domain. Therefore, the cost of RTG implementation is inexpensive and the process can be automated easily. On the contrary, some researchers have mentioned that the RTG is inefficient for software testing in terms of coverage and failure detection because it simply employs the randomness and does not apply any strategies for them.

2.3 Feedback-Directed Random Test Generator (FDR)

2.3.1 Overview

Common strategies used to generate test case automatically can be classified into two categories: random testing and systematic testing. As mentioned previously, one is random testing to generate test cases randomly. This can be done without knowing and analyzing the SUT. On the other hand, another one is systematic testing and it is for determining 1) whether the SUT contains the proper functions to satisfy the requirements, 2) whether the SUT computes the requirements correctly, and 3) whether the SUT performs efficiently [21]. Some researchers argue that random testing can be as efficient as systematic testing [8, 18], and yet, others suggest that

random testing is less efficient when compared to systematic testing since random testing does not investigate the software and simply generate test cases [9, 25, 28].

It is advantageous to have both random and systematic properties when generating test cases if it is possible. Pacheco proposed advanced random test generation, which is FDR, to achieve these two properties by generating test cases incrementally [26]. FDR generates test cases from scratch; it 1) selects some available methods with the input domain, 2) executes them, and 3) evaluates the results to checks whether the test cases are valuable to create the next ones. By this way, it can avoid producing test cases that it has been generated previously, and it only utilizes the useful test cases to generate the next test cases, which traditional random testing cannot achieve.

2.3.2 Algorithm: The FDR

Originally, the FDR was proposed as a test generating approach for an object-oriented unit test. Thus, an FDR takes a set of methods with their parameter domains as inputs and generates a set of test cases as outputs for the SUT.

```
Input:  a set of methods with their input domains, and a time-limit
Output: a set of methods sequences

1:  fdr(methods, time-limit)
2:      errorSeqs := {}
3:      nonErrorSeqs := {}
4:      while a time-limit not reached do
5:          if nonErrorSeqs is empty then
6:              newSeq := {}
7:          else
8:              newSeq := selectSeqRandomly(nonErrorSeqs)
9:          end if
10:         m := selectMethodRandomly(methods)
11:         newSeq := appendSeq(newSeq, m)
12:         isOK := execute(newSeq)
13:         if isOK is true then
14:             nonErrorSeqs := nonErrorSeqs ∪ {newSeq}
15:         else
16:             errorSeqs := errorSeqs ∪ {newSeq}
17:         end if
18:     end while
19: return errorSeqs and nonErrorSeqs
```

Figure 3: Algorithm of an FDR

Figure 3 shows the algorithm of an FDR. The FDR creates method sequences, as test cases. It first initializes two empty sets of method sequences: a set of *errorSeqs*, method sequences that result in errors, and another set of *nonErrorSeqs*, which is the sequences that succeed in execution with no errors (*lines 2-3*). Afterwards, it generates sequences continuously until a time limit is reached (*lines 4-21*). For every iteration of the main loop, the FDR builds each method sequence using previous sequences. At the beginning of each iteration to create a new sequence, *newSeq* is initialized by selecting from previous sequences depending on the

nonErrorSeqs. *newSeq* is initialized as empty if no sequence is available in *nonErrorSeqs* (*lines 6-10*). Next, it chooses one or multiple methods (*m)* and append the selected methods to *newSeq* in order to create a new sequence (*line 12*). After creating a new sequence, *newSeq*, the FDR determines if it causes errors by executing it (*line 15*). If it is safe to execute, the sequence is added into *nonErrorSeqs* to be utilized in future iterations (*line 17*). Otherwise, it is saved in *errorSeqs* (*line 19*).

2.4 Feedback-Controlled Random Test Generator (FCR)

2.4.1 Overview

The FDR improves upon the RTG by not generating duplicate and invaluable test cases through generating method sequences incrementally. However, Yatoh points out that FDR causes one problem such that the generated method sequences by FDR are biased [29].

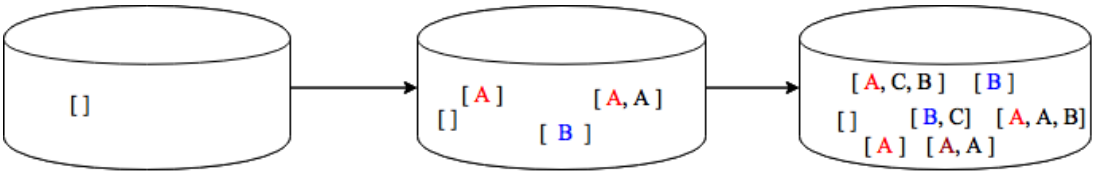

Figure 4: Biased Test Inputs

Figure 4 illustrates what bias means here. An FDR uses method sequences that have been generated previously to create new ones. Method sequences are generated by selecting a previous sequence and appending an executable method randomly into the previous sequence. FDR repeats this step continuously to generate test cases until timeout is reached. Because of utilizing previous sequences to generate new sequences, the methods selected at the beginning affect what new

sequences will look like. This is a bias problem. The FCR introduces a more recent concept to solve the problem, called *POOL*, which is defined as a container that holds previously-generated method sequences. The sequences in *POOL* are still biased. However, the FCR utilizes a set of multiple *POOL*s. Therefore, a method sequence that uses a *POOL* does not affect the sequences of the other *POOL*s. Eventually, the FCR produces less biased sequences and has a greater variety of them.

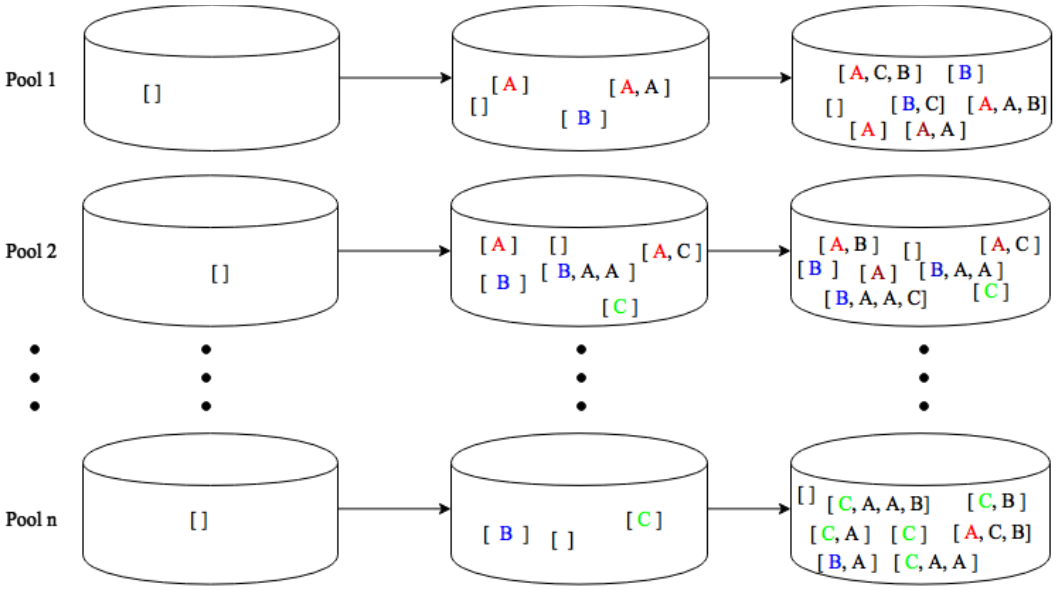

Figure 5: Biased Test Inputs with Multi-Pools

Figure 5 illustrates how the method sequences in each *POOL* are biased. As shown in the figure, each sequences are still biased in their own *POOL*. However, they are not affected each other in different *POOL*s.

Since FCR handles multi *POOL*s, it is necessary to consider how to manage them.

- It is unwise to use a single *POOL* too much because this would produce the same results as an FDR,

- Ideally, a better *POOL* will generate better sequences. Thus, it is necessary to define what *POOL* is better to generate better sequences.
- How many *POOL*s is appropriate? Too much *POOL*s results less time to investigate each *POOL* while too few *POOL*s causes less biased-free.

In FCR, *POOL* is managed by three procedures 1) how to add *POOL*, 2) how to select *POOL*, 3) how to delete *POOL*.

*2.4.2 Adding POOL*s

In an FCR, it prepares multiple *POOL*s to deal with bias problem. The question is how many of them we should prepare and whether we should add new *POOL* or not during the entire of procedure. For example, having more *POOL*s can reduced the amount of biased sequences. However, each *POOL* will have less time to generate method sequences if there are too much. Another thing that we need to consider is that suppose we prepare 100 *POOL*s. After some iterations, sequences in each *POOL* are biased differently. However, we do not know whether those biased sequences are enough or not to generate good test cases. Therefore, we still should create a new *POOL* in order to get new biased sequence. In original strategy, Yatoh created a new *POOL* every second [29].

2.4.3 Selecting POOL

One problem of using a set of *POOL*s is that it is difficult to know which to select. In the paper, *Feedback-Controlled Random Test Generation*, a *POOL* was selected based on a score function, defined as follows:

$$getScore(POOL) = \begin{cases} \frac{|coverage(POOL)|}{time(POOL)}, & if\ coverage(POOL) \neq \emptyset \\ \infty, & otherwise \end{cases}$$

where *coverage(POOL)* is a set of coverage created by executing method sequences in the given *POOL*, and *time(POOL)* is the elapsed time using *POOL* to generate method sequences. The score represents how efficient *POOL* is in terms of providing more coverage in less time. This is vital because bugs are never found in lines of codes that have not be executed [13, 22]. Because of this, the FCR considers *POOL* is better if it provides better coverage based on the score function. The FCR calculates scores for each *POOL* and selects one, assigning the maximum score when it generates test cases.

2.4.4 Deleting POOLs

As mentioned in section 2.4.3, the FCR adds a *POOL* every second to vary its biased sequences. However, too many *POOL*s can substantially diminish the amount of time each *POOL* is used, leading to insufficient length in the method sequences. It is because of this that the number of *POOL*s must be reduced at some points. The FCR sets the maximum number of *POOL*s and deletes half of them when the limit is reached. A *POOL* is deleted when it is determined to be the least unique; the uniqueness of a POOL is defined as follows:

$$uniqueness(POOL) = \frac{\sum_{c \in coverage(POOL)} uniqueness(c, POOL)}{|coverage(POOL)|}$$

$$uniqueness(c, POOL) = \frac{count(c, POOL)}{\sum_{p \in POOLS} count(c, p)}$$

where *count(c, POOL)* returns how many times the *POOL* covers the location *c*.

In the above equation, the uniqueness of a given *POOL* is the average of the uniqueness of each coverage (*c*) covered by the *POOL*. Furthermore, the uniqueness of a coverage (*c*) covered by a *POOL* is a percentage of the number of times the *POOL* covers (*c*) amongst the number of times all *POOL*s covers (*c*). In the FCR, it is better to have more sequences. Because of this, it keeps *POOL*s that are unique in terms of covering coverage locations that other *POOL*s do not, ensuring that all biased sequences in each *POOL* are covered uniquely.

2.4.5 Algorithm: The FCR

The FCR is an alternative to the FDR in that it manages multiple *POOL*s. The input of the FCR is a set of methods using the SUT, same with FDR, and two additional parameters: the initial number of *POOL*s and the maximum number of *POOL*s. It outputs a set of method sequences which is the same as the FDR.

Input:	a set of methods with their input domains, and time-limit
Output:	a set of method sequences
1:	fcr(methods, time-limit)
2:	*pools* := {}
3:	**for** i=1 **to** INP **do**
4:	*pools* := *pools* ∪ {createNewPool()}
5:	**end for**
6:	**while** a time-limit not reached **do**
7:	**if** need to add new Pool **then**
8:	*pools* := *pools* ∪ {createNewPool()}
9:	**end if**
10:	*pool* := selectPool(*pools*)
11:	fdr(*pool*, methods)
12:	**if** number of pools is more than number of MNP **then**
13:	*pools* := deletePools(*pools*, MNP/2)
14:	**end if**
15:	**end while**
16:	**return** getAllErrorSeqs(*pools*) and getAllNonErrorSeqs(*pools*)

Figure 6: Algorithm of an FCR

- *POOL*: a container of *errorSeqs* and *nonErrorSeqs*.
- *POOLS*: a set of *POOL*s
- *INP*: the initial number of *POOLS*
- *MNP*: the maximum number of POOLS

Figure 6 indicates how an FCR works. The *POOL* constructer *createNewPool* is called *INP* times to initialize *POOLS* (*lines 2-5*). After initialization, the FCR manages *POOLS* to generate method sequences (*lines 6-15*). At the beginning of the main loop, it assesses whether it needs to create a new *POOL* and adds it to *POOLS* if necessary (*lines 7-9*). Then, scores are calculated in each POOL and select the one which provides the maximum score. The, run it in the FDR to generate method sequences (*lines 10-11*). It also determines if the number of *POOLS* exceeds *MNP*. If so, the FCR deletes half of *POOLS* (*lines 12-14*). Finally, it returns all *errorSeqs* and *nonErrorSeqs* among *POOLS*.

CHAPTER 3: DESIGN AND IMPLEMENTATION

The previous chapter discussed the way in which a test generator communicates with the Template Scripting Testing Language (TSTL) to generate test cases, and how Feedback-Directed Random Test Generators (FDR) and a Feedback-Controlled Random Test Generators (FCR) work. In this chapter, our design and implementation of an FDR and an FCR for TSTL, named *feedbacktester.py*, is presented.

3.1 The Workflow of Main Components: feedbacktester.py

The defining characteristic of *feedbacktester.py* is how we manage *POOL* and *POOLS* since we can consider FDR as special case of FCR such that FCR with having a single *POOL* is FDR. We defined two object-oriented classes in order to manage *POOL* in *feedbacktester.py*.

First class is *POOL* which contains the following member variables and functions:

- member variables: the information to generate method sequences
- member function: method sequences produced by the FDR

Another class is *POOLS* which contains:

- member variables: a set of *POOL*
- member functions: adding, selecting, and deleting *POOL*

In *feedbacktester.py*, FDR and FCR were implemented by managing the above two classes.

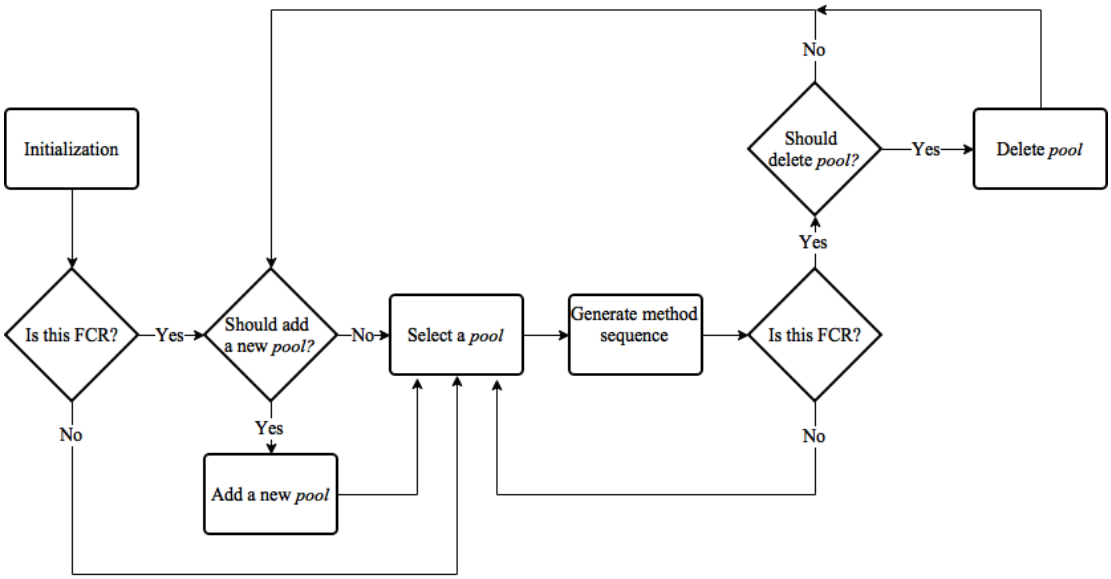

Figure 7: The Workflow of Main Components: feedbacktester.py

Excepting initialization step, there are four main components in *feedbacktester.py* as shown in *Figure 7* and those components are controlled by the defined classes, *POOL* and *POOLS*. Firstly, at the initialization, the classes, *POOL* and *POOLS* are prepared. If it is generating method sequences under the algorithm of the FCR, *POOLS* identifies 1) whether it should create a new *POOL*, 2) selects a *POOL* from *POOLS*, 3) generates method sequences using the selected *POOL*, and 4) deletes and updates *POOLS* if necessary. If an FDR is used to generate method sequences, *POOLS* holds a single *POOL* only. It means that it always selects the same *POOL* to generate method sequences. By this way, FDR and FCR shared same workflow but generated test cases differently.

3.2 The Workflow of the Component: Generating Method Sequences

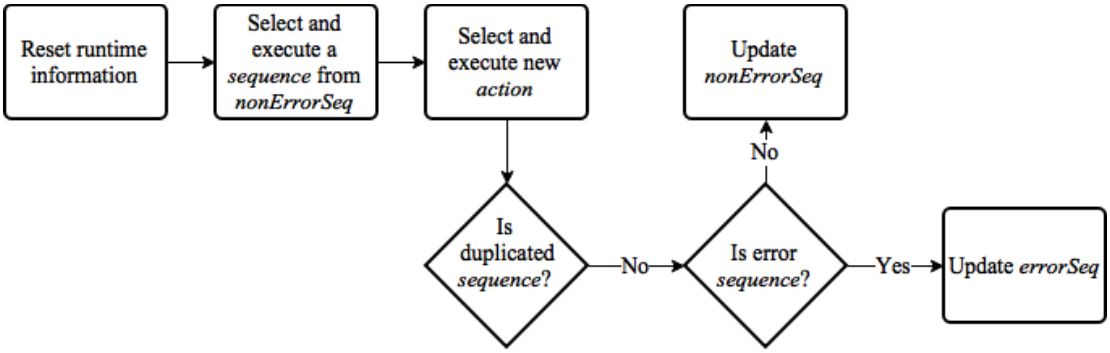

Figure 8: The Workflow of the Component: Generating Method Sequences

In this section, we describe the workflow of generating method sequences (shown in *Figure 8*). As mentioned in the previous section, we use the selected *POOL* to generate method sequences, and the selected *POOL* contains all necessary information to generate sequences. *errorSeqs* and *nonErrorSeqs* are the main information to generate test cases that *POOL* holds. They are defined as follows:

- *errorSeqs*: a set of *sequences* that causes errors
- *nonErrorSeqs*: a set of *sequences* that execute with no errors

Additionally, we defined *sequence* and *action* as following:

- *sequence*: a list of *action*s
- *action*: a pre-defined method with input value in *sut.py*

At the beginning of the workflow, it reset the runtime information which are the information of generating and executing method sequences previously. After resetting, a *sequence* from *nonErrorSeqs* is selected and each *action* in the *sequence* is executed one by one. Then, single or multiple *action* are selected to executed and appended into the selected *sequence* to generate a new *sequence*. Finally, it is

determined whether the generated *sequence* is duplicated or not. If not, the *sequence* is placed in *nonErrorSeqs*. Otherwise, it is put into *errorSeqs*.

It should note that our implementation must reset the runtime information when generating method sequence each time even it causes inefficient in terms of time complexity. There are three reasons why the runtime information needs to be reset each time.

- First reason is why it is necessary to reset a runtime information is because how TSTL defines *action*. As mentioned in *section 2.1.3*, *action* is defined by following the property of the Software Under Test (SUT). It means that it is runtime information stating which *action* is executable after another *action* is performed.
- The second reason is that the FDR and FCR generates method sequences incrementally. In our implementation, we selected previous *sequences* from *nonErrorSeqs* and appended an executable *action* to generate a new *sequence*. Because of TSTL, we had to execute each *action* in the previous *sequence* to know which *action* was executable.
- The third reason is that the method sequences are generated repeatedly until timeout is reached. This means that the runtime information of the previous method sequence not only still existed, but also affected the generation of a new method sequence.

CHAPTER 4: EXPERIMENTS

Chapter 4 presents our experiments to measure the performance of our implementation of Feedback-directed Random Test Generator (FDR) and Feedback-controlled Random Test Generator (FCR), named *feedbacktester.py*. We compared *feedbacktester.py* with a test generator from The Template Scripting Testing Language (TSTL), *randomtester.py* in terms of coverage, time efficacy, and error-detection.

4.1 Performance Measurement

There are several ways of measuring performance for software testing [3]. The P-measure is the probability of finding at least one failure in the test cases. The E-measure is the expected number of failures reported in the test cases. When using those measurements, it is necessary to prepare test cases prior to execution since we need to know what those test cases look like. However, *feedbacktester.py* and *randomtester.py* generate test cases at runtime with accessing *sut.py*. Therefore, we needed another approach to measure our generator. F-measure is defined as how many test cases are generated to detect the first failure [4]. Using F-measure, we evaluated the coverage performance of *feedbacktester.py* and *randomtester.py* as the following ways:

- How many test cases a generator creates to cover the max coverage is covered

- How many seconds a generator takes to cover the max coverage is covered

Similarly, we evaluated the error-detection of those test generators for the following:

- How many test cases a generator creates to find a first failure
- How many seconds a generator takes to find a first failure

4.2 Case Study

Case Study	Lines	Classes	Contains Bug?	Description
avl	288	2	Yes	AVL Tree implementation
simplejson	4071	2	No	JSON parser
sortedcontainers	3609	6	No	a sorted containers for list, dictionary, and set
sympy	415794	617	Yes	symbolic computer algebra system
my_xml	637	6	No	XML parser as read-only

Table 1: Case Study

In our experiments, we needed a python file and a TSTL file for the Software Under Test (SUT) to create *sut.py,* and we used the files for various SUTs provided by Groce [17]. We reported the results of our experiments using the following SUTs: AVL Tree implementation, JSON parser, a sorted containers of List, Dictionary, and Set, a symbolic mathematics library, and XML parser (shown in *Table 1*). Some case studies, avl and sympy, contained bugs in order to measure the ability of a generator's detecting failures.

4.3 Results and Discussion

In this section, we show the results and discussions of our experiments for each test generator, FDR, FCR, and RTG.

- We ran *feedbacktester.py* and *randomtester.py* for 600 seconds because of FCR. We set Initial Number of *POOL*s (INP) as 10 and Maximum Number of *POOL*s (MNP) as 100 for FCR and FCR added a new *POOL* in each second. It meant that it took 90 seconds to reach MNP and to delete *POOL*. Thus, we needed the long execution time to measure FCR and spent 600 seconds.
- We used multiple seeds for generating random numbers to investigate the variety of each generators and we used random seeds as between 1 to 10.

The following tables show our experiment results for each case studies.

Test Generator	Test Cases	Coverage (%)	Failures	Test Cases (Coverage)	Seconds (Coverage)	Test Cases (1st Failure)	Seconds (1st Failure)
FDR	16320.600	76.677	0.700	1703.400	58.113	8495.333	305.478
FCR	5189.100	76.422	15.200	1718.800	201.884	342.000	36.415
RTG	4891.400	76.038	2.400	93.200	10.307	1983.400	215.378

Table 2: Results for avl

Test Generator	Test Cases	Coverage (%)	Failures	Test Cases (Coverage)	Seconds (Coverage)	Test Cases (1st Failure)	Seconds (1st Failure)
FDR	537.300	32.456	N/A	430.200	485.963	N/A	N/A
FCR	69.700	24.351	N/A	39.800	347.302	N/A	N/A
RTG	39.900	33.623	N/A	31.400	478.817	N/A	N/A

Table 3: Results for simplejson

Test Generator	Test Cases	Coverage (%)	Failures	Test Cases (Coverage)	Seconds (Coverage)	Test Cases (1st Failure)	Seconds (1st Failure)
FDR	1699.444	22.905	N/A	1575.333	555.921	N/A	N/A
FCR	450.300	9.977	N/A	270.700	356.830	N/A	N/A
RTG	503.000	29.173	N/A	433.857	519.919	N/A	N/A

Table 4: Results for sortedcontainers

Test Generator	Test Cases	Coverage (%)	Failures	Test Cases (Coverage)	Seconds (Coverage)	Test Cases (1st Failure)	Seconds (1st Failure)
FDR	57.800	17.927	0.400	49.100	464.194	31.333	234.458
FCR	8.889	16.813	0.667	4.500	288.095	7.400	364.348
RTG	3.875	19.878	0.500	3.286	435.950	2.000	197.705

Table 5: Results for sympy

Test Generator	Test Cases	Coverage (%)	Failures	Test Cases (Coverage)	Seconds (Coverage)	Test Cases (1st Failure)	Seconds (1st Failure)
FDR	13927.700	33.135	N/A	6084.300	233.239	N/A	N/A
FCR	5502.100	32.649	N/A	1876.200	185.888	N/A	N/A
RTG	27871.100	33.784	N/A	1761.300	38.145	N/A	N/A

Table 6: Results for XML

Each column represents as following description:

- Test Cases: total number of test cases that a generator creates and executes until timeout
- Coverage: the max coverage that a generator covers until timeout
- Failures: total number of failures that a generator detect
- Test Cases (Coverage): number of test cases that a generator to cover the max coverage
- Seconds (Coverage): how many seconds a generator takes to cover the max coverage
- Test Cases (1st Failure): number of test cases that a generator to detect 1st failure
- Seconds (1st Failure): how many seconds a generator takes to detect 1st failure

- N/A indicates that we could not obtain results for some reasons. For example, SUT did not contains bugs or the running process was killed by OS because of extreme memory consumption.

According to *Table 2* and *Table 5*:

- For coverage efficiency, RTG showed best score. For both *avl* and *sympy*, all generators outputted similar coverage score. However, RTG needed lesser test cases and seconds to cover it.
- For detecting failures, FCR resulted best score. For instance, FCR used minimum test cases and seconds to find 1st failure for *avl*. Furthermore, FCR found more numbers of failures than others for *avl* and *sympy*. As mentioned in Chapter 2, FCR utilized unique POOL. It implied that the failures in those two SUTs were located in less frequent coverage locations.
- When we compared FDR and FCR in *avl* and *sympy*, FCR was obviously better than FDR for detecting failures, but it showed different for covering coverage. For *avl* shown in *Table 2*, FDR was faster than FCR to cover max coverage. Approximately, FDR spent 60 seconds. As I mentioned before, FCR needed at least 90 seconds to handle its algorithm. This result was because *avl* was relatively small program and FDR algorithm was enough to cover the entire of the program. On the other hand, FCR was faster than FDR to cover the max coverage in *sympy* since *sympy* was huge program and needed more complex algorithm such that FCR had advantage for coverage.

As shown in *Table 3*, *Table 4*, and *Table 6*:

- RTG was better than feedback based generators in terms of coverage. Firstly, all generators covered few coverage because the functions defined from *simplejson* and *sortedcontainers* were called recursively in those tstl files. It meant that the amount of paths became exponential and it was impossible to cover high coverage in practical time. This may work against to feedback based generators since then generate test cases incrementally.
- FCR provided the least coverage score compared with others. FCR managed multi *POOL*s and selected the most unique one to generate test cases. In other words, it was necessary for FCR to have enough coverage information to obtain its benefit.
- FDR was better than FCR in *simplejson* and *sortedcontainers*. FDR was specious case of FCR in our implementation, using a single *POOL*. It implied that it was better to focus on single *POOL* when it had low coverage. In other words, FCR could not glow each *POOL* to calculate uniqueness.
- *Table 6* also showed similar trend with other case studies. All generators covered same coverage but we could say that RTG was more efficient than others for coverage because it needed less test cases and times.

To sum up, the following insights were obtained:

- Our implementation of FCR, *feedbacktester.py*, was effective in terms of error-detection. However, it required to obtain some amount of coverage to get advantage of its algorithm since it needed to calculate the uniqueness of *POOL*, whose uniqueness required coverage information.

- The differences between the FDR and the FCR was that FCR achieved greater coverage in less time when we could obtain enough coverage information. The FCR used a multi-*POOL* to generate more unique method sequences. However, FDR was appropriated if the SUT was simply enough or it was hard to get enough coverage information.
- FDR showed low performance than others but FCR was superior in terms of detecting bugs. However, since FCR was based on FDR in our implementation, we could provide better results if we could improve FDR implementation more. One possible way might improve its time complexity. FDR was necessary to reset the runtime information in our current implementation. If we could avoid it, it would increase the time complexity.
- The RTG used fewer test cases to perform greater coverage compared to the FDR and FCR. We concluded this result was that RTG was simple and feedback based generators were complex. RTG spent much less time than those two generators to generate test cases. In contrast, FDR and FCR needed huge calculations to create test cases. For examining the performance in our experiment, generating and executing test sequences simply results better than applying the complex strategy.

CHAPTER 5: CONCLUSION

Our group presented a fully-automated test generator for the Template Scripting Testing Language (TSTL) based on two advanced random test generators: a Feedback-Directed Random Test Generator (FDR) and a Feedback-Controlled Random Test Generator (FCR). Unlike the generators supported by TSTL, we employed an incremental technique in the generation of test cases. Based on our observations, the proposed implementation is applicable in the scalable input domain, because the FDR and FCR are scalable.

In our experiments, we compared the performance of *feedbacktester.py* with *randomtester.py* from TSTL in terms of coverage, running time, and failure detection. Our experiments showed that FCR in *feedbacktester.py* was best in terms of detecting bugs. However, feedback based generators had low performance for coverage. RTG was better since it used less test cases and seconds to obtain more coverages. It was because feedback based generators required heavily calculations but random generator allowed us to generate test cases with light calculation. According to our experiments, light calculation, selecting and executing test cases randomly, was superior to heavy calculation, feedback based test generations.

Therefore, to reduce time complexity for feedback based generations might be next step for our project. Currently, our implementation, *feedbacktester.py*, belonged to *sut.py* provided by TSTL. In other words, *feedbacktester.py* was strongly tied with

the implementation of TSTL. Because of it, our generator was necessary to reset the runtime information every time when it generated test cases. This process was inefficient. If our generator could be free from TSTL on this part, time complexity would become much better. In addition, calculating uniqueness of *POOL*s was also heavy calculation. It was worth to consider about how to manage coverage information for calculating the uniqueness.

BIBLIOGRAPHY

[1] V.D. Agrawal. When to Use Random Testing. In *IEEE Trans. Computers*, no. 11, pp. 1054-1055, November 1978.
[2] Y. Amannejad, V. Garousi, R. Irving and Z. Sahaf. A Search-Based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, Cleveland, OH, pp. 302-311. 2014.
[3] T.Y. Chen and Y.T. Yu. On the Expected Number of Failures Detected by Subdomain Testing and Random Testing. In *IEEE Trans. Software Eng.* no. 2, pp. 109-119, Feb. 1996.
[4] T.Y. Chen, H. Leung, and I.K. Mak. Adaptive Random Testing. In *Proceedings of Ninth Asian Computing Science Conf.* pp. 320-329. 2004.
[5] L. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on software testing and analysis*, pp.84-94, 09 July 2007.
[6] E. F. Collins and V. F. de Lucena. Software Test Automation practices in agile development environment: An industry experience report. In *2012 7th International Workshop on Automation of Software Test (AST)*, Zurich, pp. 57-63. 2012.
[7] B. Daniel and M. Boshernitsan, Predicting Effectiveness of Automatic Testing Tools, In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sept. pp.363-366, 2008.
[8] J. W. Duran and S. C. Ntafos. An evaluation of random testing. In *IEEE Transactions of Software Engineering*, July 1984, Vol.SE-10(4), pp.438-444 [Peer Reviewed Journal].
[9] R. Ferguson and B. Korel. The chaining approach for software test data generation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 01 January 1996, Vol.5(1), pp.63-86 [Peer Reviewed Journal]
[10] I. Fernandez, A. D. Cerbo, E. Dehnhardt and M. Tipaldi. Test automation for critical space software. In *2016 IEEE Metrology for Aerospace (MetroAeroSpace)*, Florence, pp. 551-555. 2016.
[11] M. Fowler. Domain-Specic Languages. Addison-Wesley Professional. 2010.
[12] S. Galler and A. Bernhard, Survey on test data generation tools, In *International Journal on Software Tools for Technology Transfer*, Vol.16(6), pp.727-751, 2014. [Peer Reviewed Journal]
[13] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 72–82, 2014.
[14] A. Groce and R. Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *Workshop on Dynamic Analysis*. pp. 22-28. 2008.

[15] A. Groce and J. Erwig. Finding common ground: choose, assert, and assume. In *Work-shop on Dynamic Analysis*, pp. 12-17, 2012.
[16] A. Groce, A. Fern, M. Erwig, J. Pinto, T. Bauer, A. Aipour. Learning-based test programming for programmers. In *International Symposium on Leveraging Applications of Formal Methods*. Verication and Validation, pp. 752-786. 2012.
[17] A. Groce, J. Pinto, P, Mittal, P, Azimi, K. Kellar, and J. O'brien, TSTL: the template scripting testing language, In *International Journal on Software Tools for Technology Transfer*, 2 December 2016. [Peer Reviewed Journal]
[17] A. Groce. Template Scripting Testing Language tool. Retrived from https://github.com/agroce/tstl. 2017.
[18] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. In *IEEE TSE*, 16(12):1402-1411, Dec. 1990.
[19] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pp. 970-978, Wiley, New York, 1994.
[20] M. Harman. Automated Test Data Generation using Search Based Software Engineering. In *Automation of Software Test , 2007. AST '07. Second International Workshop on*, Minneapolis, MN pp. 2-2. 2007.
[21] M. Höge, A. Hohmann, K. V. D. Horst, A. Evans. and H. Caeyers. User participation in the TWB II project - the first test cycle. In *Report of the ESPRIT II project 6005 Translator's Workbench II (TWB II)*. Mercedes-Benz AG, SITE and CEC Language Services, Stuttgart, Paris. Luxembourg. 1993.
[22] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 435–445, 2014.
[23] K. Kellar. TSTL-Java. Retrieved from https://github.com/flipturnapps/TSTL-Java. 2015.
[24] P. S. Loo and W. K. Tsai. Random testing revisited. In *Information and Software Technology*, vol. 30, no. 7, pp. 402-417, 1988.
[25] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT/LCS/TR-921, MIT Lab for Computer Science, September 2003.
[26] 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, pp.75-84. 2007.
[27] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA'07, pages 815–816, 2007.
[28] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In ISSTA, pages 37-48, July 2006.
[29] K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. Feedback-Controlled Random Test Generation. In *Proceedings of the 2015 International Symposium on software testing and analysis*, pp.316-326, 13 July 2015.