Feedback-Controlled Random Test Generation

Kohsuke Yatoh[1]*, Kazunori Sakamoto[2], Fuyuki Ishikawa[2], Shinichi Honiden[12]
[1]University of Tokyo, Japan,
[2]National Institute of Informatics, Japan
{k-yatoh, exkazuu, f-ishikawa, honiden}@nii.ac.jp

ABSTRACT

Feedback-directed random test generation is a widely used technique to generate random method sequences. It leverages feedback to guide generation. However, the validity of feedback guidance has not been challenged yet. In this paper, we investigate the characteristics of feedback-directed random test generation and propose a method that exploits the obtained knowledge that excessive feedback limits the diversity of tests. First, we show that the feedback loop of feedback-directed generation algorithm is a positive feedback loop and amplifies the bias that emerges in the candidate value pool. This over-directs the generation and limits the diversity of generated tests. Thus, limiting the amount of feedback can improve diversity and effectiveness of generated tests. Second, we propose a method named feedback-controlled random test generation, which aggressively controls the feedback in order to promote diversity of generated tests. Experiments on eight different, real-world application libraries indicate that our method increases branch coverage by 78% to 204% over the original feedback-directed algorithm on large-scale utility libraries.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Algorithms, Reliability, Verification

Keywords

Random testing, Test generation, Diversity

*The author is currently affiliated with Google Inc., Japan, and can be reached at kyatoh@google.com.

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771805

1. INTRODUCTION

Feedback-directed random testing [17] is a promising technique to automatically generate software tests. The technique can create random method sequences using public methods from the classes of a system-under-test (SUT). It is a general and test oracle independent technique to generate software tests. Due to its generality and flexibility, many researchers have used feedback-directed random testing. Some researchers leveraged feedback-directed random testing as a part of their proposed methods [5, 25]. Others used feedback-directed random testing to prove their theories on random testing [11, 12]. There is an interesting study that mined SUT specifications by analyzing the dynamic behavior of SUT observed during feedback-directed random testing [18]. In addition, feedback-directed random testing has already been adopted by industries and undergone intensive use [19].

Despite its importance, characteristics of feedback-directed random testing have seldom been studied. To the best of our knowledge, some studies have proposed extensions to feedback-directed random testing [14, 27], but they failed to analyze the nature of feedback-directed random testing. Specifically, the idea of feedback guidance had never been challenged. In this paper we investigate characteristics of feedback-directed random testing by using a model SUT and propose a new technique that exploits the obtained knowledge that excessive feedback over-directs generation, amplifies bias, and limits the diversity of generated tests.

We address two research questions in this paper.

RQ1: Why does the test effectiveness stop increasing at different points depending on random seeds?

RQ2: Can our proposed technique lessen the dependency on random seeds and improve the overall performance of test generation?

The resulting test effectiveness of feedback-directed random testing should differ because of its randomness. However, the observed difference is much larger than expected. For example, the interquartile range marks 10% in our preliminary experiment on the model SUT. This spoils the credibility of feedback-directed random testing.

There are three contributions in this paper.

- We hypothesize that *feedback guidance over-directs the generation and limits the diversity of generated tests* and show that both average score and variance of test effectiveness improve by limiting the amount of feedback.

- We propose an algorithm named feedback-controlled random testing, which controls the amount of feedback adaptively.
- We demonstrate advantages of our algorithm through experiments on eight different real-world application libraries. The average score of obtained test effectiveness increases over 204% in the best case, while keeping the interquartile range under 1.2%.

The rest of this paper is organized as follows. Section 2 introduces the feedback-directed generation algorithm and discusses the performance characteristics of the algorithm. Our hypothesis for this characteristics is posed and confirmed in Section 3. Our feedback-controlled generation algorithm is described in Section 4, and evaluated in Section 5. We review related work in Section 6 and finally summarize our study in Section 7.

2. ANALYSIS OF FEEDBACK-DIRECTED RANDOM TEST GENERATION

In this section, we introduce the algorithm of feedback-directed random test generation and discuss its characteristics. Feedback-directed random testing is a technique to create random test inputs. It constructs method sequences from public methods extracted from a given list of classes under a test. Each method sequence makes a unit test of SUT. In each method sequence, inputs are created, mutated, and consumed by method calls. The generated method sequences can be used for any purpose, like fault detection by contracts [17] or regression test generation [19]. It can also be used to guide dynamic symbolic execution [9, 26] or to mine specifications of software [18].

The main difference between feedback-directed random testing and other random test generation techniques is that feedback-directed random testing utilizes feedback from the previously generated tests to guide the generation. Feedback-directed random testing starts from trivial method sequences that contain only a single method call or a constant. Then, it creates longer method sequences by concatenating a newly selected method and previously created method sequences. A method call normally returns a value, so method sequences can supply these return values as arguments for later method calls. When choosing a method sequence to extend, feedback-directed random testing leverages feedback from the execution results of sequences. Sequences that throw exceptions are not worth extending, because an exception prevents execution of successive method calls. When generating a method sequences without this feedback, it is likely that a long method sequence is terminated with an exception and the latter part of the sequence remains unexecuted. By using feedback, it can guide the generation and efficiently make long method sequences that terminate normally.

2.1 Algorithm

Before starting analysis, we can benefit from formally defining the feedback-directed random test generation algorithm. It takes a class list as input and outputs method sequences. Generated method sequences include public methods from the input classes.

Figure 1 shows the algorithm of feedback-directed test generation. The original paper [17] suggests using some optimizations, but we omitted them to focus on the fundamental flow of the algorithm. The algorithm takes the class list of SUT as an input. At first, the pool is initialized with random primitive values (*createNewPool*). Each primitive is encoded as a single sequence that creates a variable initialized with the primitive. Then, the algorithm continues generating method sequences until the time limit is reached (*generate*). In the main loop, new method sequences are created by combining a public method randomly selected from the class list with previously generated sequences that supply values compatible with the method parameters (*createSequence*). The created sequence is executed (*execute*). After execution, some sequences are revealed to be inextensible (*isExtensible*). Sequences that throw exceptions cannot be extended, because even if we concatenate a method after them, the method will never be executed. Sequences that violated contracts are also not worth extending. If the sequence is extensible, it is added to the pool (*addToPool*).

After the main loop is terminated, generated method sequences are held in the value pool. We can output them to create regression tests or to point out contract violations.

```
procedure FeedbackDirectedGeneration(classes)
    pool = createNewPool()
    while time limit not reached do
        generate(pool, classes)
    end while
    return pool
end procedure

procedure generate(pool, classes)
    newSeq = createSequence(pool, classes)
    r = execute(newSeq)
    if isExtensible(newSeq, r) then
        addToPool(pool, newSeq)
    end if
end procedure
```

Figure 1: Feedback-directed generation algorithm

2.2 Performance Characteristics

To analyze the performance characteristics of feedback-directed random test generation, we conducted a preliminary experiment on a widely used utility library. Throughout this paper, we used a machine with Intel Xeon X5650 (2.67GHz) and 100GB RAM, which runs CentOS 7.0. The testing environment is isolated by Docker 1.3.2[1]. We ran an open source feedback-directed random test generator Randoop[2] [16] with OpenJDK 1.7 installed on the standard Ubuntu 14.04 container. We selected Apache Commons Collections version 4.0 as a model SUT. It is the most popular collection library according to MVNRepository[3]. It is a relatively large library with 58,186 lines of codes and has a variety of classes ranging from simple proxies to complex data structures.

We ran Randoop with ten different random seeds and plotted the branch coverage in Figure 2. This graph raises the first research question.

RQ1: Why does the test effectiveness stop increasing at different points depending on random seeds?

[1] `https://www.docker.com/`
[2] `https://code.google.com/p/randoop/`
[3] `http://mvnrepository.com/`

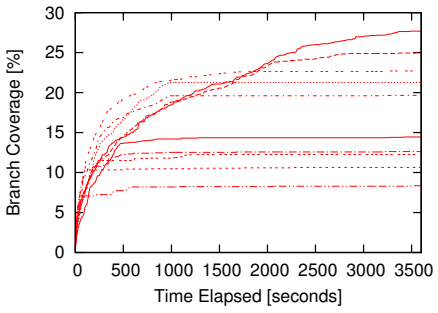


Figure 2: Branch coverage of Commons Collections 4.0 achieved by feedback-directed random testing with ten different random seeds

```
public class Stone {
    public final boolean black;
    public Stone(boolean black) {
        this.black = black;
    }
    public Stone clone() {
        return new Stone(this.black);
    }
    public boolean isBlack() {
        return this.black;
    }
}
```

Figure 3: Example SUT for illustrating over-directedness of feedback-directed random test generation

The aim of this paper is to give an answer to this question and propose a workaround method to remedy the problems.

RQ1 can be divided into two sub-questions:

RQ1-1: Why does the coverage differ depending on random seeds?

RQ1-2: Why does the coverage stop increasing at certain points?

Randomized techniques naturally depend on random seeds, and the results may change according to seeds. However, the results of feedback-directed random testing are quite unstable. This leads to the first sub-question (RQ1-1). We suspect that the reason lies in the mechanism of feedback-directed random testing.

For the second sub-question (RQ1-2), there are three possibilities. The first two possibilities seek the cause in outer factors: SUT and coverage criteria. However, we deny the first two possibilities and find the cause in feedback-directed random testing itself. Now, we review the three possibilities one by one.

The coverage has already reached the maximum value. This is the most optimistic viewpoint. Due to the existence of dead codes, statement or branch coverage does not always reach 100 percent. However, this possibility is immediately denied because the coverage differs among seeds. Apparently, there are some code lines that are covered in some seeds but not in others.

The coverage does not reflect the increase of actual fault detection capacity. How coverage is correlated with actual fault detection is being actively discussed [11, 13]. We cannot say the actual fault detection capacity is not increasing only due to the fact that the branch coverage is not increasing. However, we can at least say that faults in codes not executed in test suites can never be found, so an unexplored code region means a lack of certain fault detection capacity. Thus, fault detection capacity also stops increasing in this aspect.

The feedback-directed random test generation is no longer generating interesting tests due to its incompleteness. We believe this is the correct answer to RQ1-2. The mechanism of feedback-directed random test generation has a fundamental weakness and stops generating interesting sequences after running for a certain time. The feedback mechanism itself has a pitfall inside. In the following section, we present clues that the feedback mechanism over-directs the generation and limits the diversity of test inputs. This over-directedness can explain both the instability and low effectiveness.

3. OVER-DIRECTEDNESS

Our analysis indicates that feedback-directed random testing has some troublesome performance characteristics. We dare to say that the feedback mechanism has intrinsic weakness in itself. We hypothesize that the over-directedness of feedback-directed random testing limits the diversity of generated tests. The over-directedness affects the performance in two ways.

- The coverage score strongly depends on to which direction the algorithm guides the generation.
- The limited diversity prevents coverage from increasing after some points.

In this section, we prove the existence of the over-directedness and show how it worsens the performance of feedback-directed random testing.

3.1 Existence of Over-Directedness

Figure 3 lists an example SUT that illustrates the over-directedness of feedback-directed random test generation. The class represents a stone colored black or not. When `clone` is called, it returns a newly created stone the same color as `this`.

If black stones and non-black stones are generated in the same probability (50% each), they form a uniform distribution. The expected numbers of black and non-black stones are the same, and the variance of the distribution is small. This is false if you are using feedback-directed random testing.

The actual distribution of the number of generated stones is shown in Figure 4. We conducted the feedback-directed random testing until 100 sequences were made. We ran it with 1000 different random seeds and plotted the number of generated stones. Each point represents the numbers of black or non-black stones generated by a seed.

The total number of generated stones varies from around 300 to 600 in accordance with the different lengths of generated sequences. However, notably the distribution does not converge to the center line, which represents the expected number of each stone if they are generated in the same probability. The distribution is rather biased to each axis. For

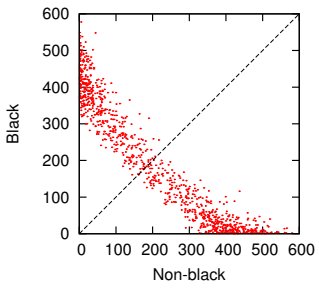


Figure 4: Distribution of `Stone` instances generated in feedback-directed random testing

example, only black stones are generated in some directions, but only non-black stones are generated in some other directions. This indicates that the distribution of feedback-directed random testing is different from the pure random testing.

This behavior is not instinctive but a result of the over-directedness. The feedback mechanism of feedback-directed random testing is classified as positive feedback. In feedback-directed random testing, the tendency of former test generation is amplified by the feedback mechanism. This is because methods in SUT ordinarily produce a value that is strongly correlated with the arguments. Similar arguments lead to similar return values, and the return values are used as similar arguments.

There exists a limited heuristic to avoid this situation. The authors of feedback-directed random testing leverage the `equals` method to avoid duplicated values. In the `Stone` case, if `Stone` implements the `equals` method, the situation can be avoided. However, that heuristic does not apply to real settings. For example, `List` class has many similar but different states. `List`s with the same prefix will behave almost the same with `find(`x`)` calls if x can be found in the prefix. The `equals` heuristic cannot decide which `List`s are similar or not.

3.2 Effect of Over-Directedness

To estimate the effect of over-directedness on testing real software, we conducted a preliminary experiment on the model SUT. We compared two configurations:

1. Run feedback-directed random testing only once, 3600 seconds.

2. Run feedback-directed random testing 36 times. Each run has a time limit of 100 seconds. Thus, total time is 3600 seconds.

The first configuration is a baseline. The second configuration is equivalent to resetting value pools during feedback-directed random testing. The feedback is reset every 100 seconds. In other words, directedness is limited to 100 seconds. For each configuration, we calculated the cumulative code coverage. We used 100 different random seeds and plotted the results in Figure 5.

As shown in Figure 5, the second configuration outperforms the first. Even the maximum coverage score of the first configuration is lower than the minimum coverage score of the second. The variance of the second configuration is much smaller than that of the first. This means the second configuration is better in terms of both the score and stability.

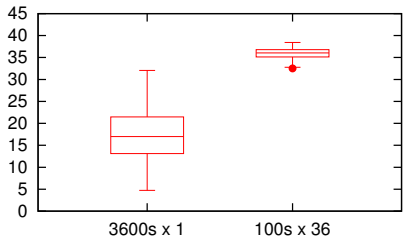


Figure 5: Box plots of branch coverage of Commons Collections 4.0, comparing two configurations

These results indicate that the amount of feedback and directedness must be controlled carefully. In the first configuration, we used full feedback but obtained poor results. In the second configuration, we limited the feedback and obtained good results. However, without sufficient feedback, we cannot generate sequences of method calls, which is the original purpose of the feedback-directed random testing. Thus, we must find the appropriate amount of feedback. Finding the appropriate amount of feedback can be seen as balancing the trade-off between search depth and breadth. By directing the generation, we can search deeply, and by un-directing the generation, we can search widely. In the next section, we propose a technique that adaptively balances the trade-off.

4. FEEDBACK-CONTROLLED RANDOM TEST GENERATION

We propose a method to control the amount of feedback and balance the trade-off between search depth and breadth. It is named feedback-controlled random test generation, because it controls in which direction to search by utilizing the feedback information. Our basic idea is to use multiple pools concurrently, instead of using a single pool. Each pool has different contents, so different pools guide the generation toward different directions. By holding multiple pools at the same time and concurrently using them, the feedback-controlled generation algorithm can dynamically decide in which direction to search further and in which direction to stop searching by leveraging the feedback information.

Instead of managing a single pool, the algorithm manages a set of pools (*pools*). When considering a set of pools, there exist three basic operations on a set.

Selecting an item. Equivalent to selecting a pool to use.

Adding items. Equivalent to introducing new pools and initiating new directions in which to search.

Deleting items. Equivalent to stopping searching a pool further.

By combining the above operations, we can control the depth and breadth of search directions. The next search direction is determined by selecting a pool to use. Adding a new pool widens the breadth, but the new pool requires some of the time budget, which is otherwise used to deepen the existing directions. Deleting pools increases the time budget for the remaining pools, but the deleted directions cannot

```
parameter: INP, MNP

procedure FeedbackControlledGeneration(classes)
    pools = {}
    for i = 1, INP do
        pools = pools ∪ {createNewPool()}
    end for
    while time limit not reached do
        if shouldAddPool(pools) then
            pools = pools ∪ {createNewPool()}
        end if
        pool = selectPool(pools)
        generate(pool, classes)
        if |pools| > MNP then
            pools = deletePools(pools, MNP / 2)
        end if
    end while
    return pool
end procedure
```

Figure 6: Feedback-controlled generation algorithm

be searched in any more. For each of the above operations, there may be various strategies. In this research, we adopted simple heuristics.

The algorithm of feedback-controlled random testing is listed in Figure 6. It requires two additional parameters: *INP* (the initial number of pools) and *MNP* (the maximum number of pools). Initially, the set of pools contains *INP* new pools. Then, until time limit is reached, the generation continues. When *shouldAddPool* returns *true*, a new pool is added to the set. Before each generation attempt, the pool to use is selected by *selectPool*. The generation itself is the same as the original feedback-directed random testing. After the generation, if the number of pools exceeds *MNP*, the set of pools is updated by $deletePools(pools, n)$, which return n filtered pools.

4.1 Selecting Pool

The function *selectPool* takes a set of pools and returns the pool to be used. This is equivalent to determining in which direction to search further. Instead of selecting a pool at random or in rotation, we prioritize the pools with their score defined below and select the pool with the highest score.

The scoring function is defined as

$$score(pool) = \begin{cases} \frac{coverage(pool)}{consumedTime(pool)} & (coverage(pool) > 0) \\ \infty & (coverage(pool) = 0) \end{cases}$$

where $coverage(pool)$ stands for the current achieved coverage of the pool and $consumedTime(pool)$ stands for the sum of time consumed by the pool, including the method sequence creation and sequence execution. The *score* function represents the coverage score divided by the time consumed. As shown in Figure 7, $score(pool)$ monotonically decreases. Thus, the expected coverage gain can be estimated by $score(pool)$.

We reviewed in Section 2 that the coverage score differs a lot in accordance with the random seed. Thus, prioritizing pools by the expected coverage gain can improve the overall coverage increase. When $consumedTime(pool)$ is near zero, the scoring function has a spike. This works as an appetite

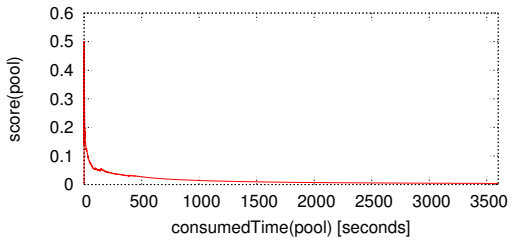


Figure 7: Shape of $score(pool)$

for newly created pools. Because the direction of newly created pools is unknown, it is better to grow them in order to know the directions to which they will guide the generation.

One drawback of this scoring function is that it does not consider the overlap of coverage among pools. For example, when two pools have exactly the same contents and produce the same method sequences, this function cannot prevent this overlap. The *deletePools* function will detect and remove such overlaps.

4.2 Adding Pool

The timing to add a new pool is determined by the function *shouldAddPool*. The newly added pool has $score = \infty$ and will be selected by the next *selectPool*. Adding a pool corresponds to initiating a new direction in which to search. However, adding too many pools slows down the pace of exploration in current directions. We adopted a simple approach in which new pools are added every second. There may be a better approach, but currently it is out of our scope.

4.3 Deleting Pool

If the number of pools exceeds *MNP*, *deletePools* is called to select pools to drop. It returns a subset of *pool* with size n. This corresponds to pruning directions in which to search. In general, it is hard to predict which direction will hit an uncovered region. To maximize the possibility of hitting an uncovered region, we use an adaptive strategy. The uniqueness of a pool is defined as follows:

$$uniqueness(pool) = \frac{\sum_{c \in covered(pool)} uniqueness(pool, c)}{|covered(pool)|}$$

$$uniqueness(pool, c) = \frac{count(c, pool)}{\sum_{p \in allpools} count(c, p)}$$

where $count(c, pool)$ returns the number *pool* passed coverage location c and $covered(pool) = \{c \mid count(c, pool) > 0\}$.

The purpose of $uniqueness(pool)$ is to measure uniqueness of coverage locations the pool passes. It is an average of each uniqueness of coverage locations the pool passes, and uniqueness of coverage locations is determined by how many times other pools passed the location. If a pool passes locations that are not passed by other pools, the uniqueness of the pool becomes higher. In contrast, if a pool passes only locations passed by other pools, the uniqueness becomes lower.

4.4 Global Resetting

If the SUT and testing environment act deterministically, the above algorithm should work fine. However, the actual SUT includes nondeterministic behaviors, and the testing environment is not deterministic either.

There are cases in which SUT have essentially nondeterministic specifications. This is not rare, and our model SUT, Apache Commons Collection, has nondeterministic functions. For example, `PassiveExpiringMap` uses system clocks to determine the validity of an entry. Even if a method sequence including `PassiveExpiringMap` executes normally at a time, the same sequence may throw an exception later. Note that, even for this case, random testing makes sense because contracts can verify the behavior of such nondeterministic methods. Another example is `ReferenceIdentityMap`, which uses the native `hashCode` implementation for hashing. Because a typical Java Virtual Machine (JVM) uses internal memory addresses to calculate native hash codes, essentially random values are returned by `hashCode`. Though the interface specifications of `ReferenceIdentityMap` do not include nondeterministic behaviors, its inner structure changes nondeterministically in accordance with `hashCode`. This is also the same for general `Maps` if element classes do not override the `hashCode` method.

Alongside SUT, the testing environment is a source of nondeterministic behaviors. The amounts of available memory and CPU resources change dynamically, which affects the behavior of generated method sequences. For example, if a method sequence requires huge memory, the first execution may succeed but successive executions may fail due to out-of-memory error. The execution time of sequences also depends on the condition of the physical machine and system. If the CPU is busy, execution will take longer than usual. The implementation of Randoop uses an execution timeout to detect an infinite loop and may misjudge some long running executions as an infinite loop. In addition to the above examples, JVM has instabilities when conducting random testing. In general, random testing requires a huge amount of memory and CPU time by creating many objects and exercising SUT in an unusual way. As a result, JVM instance becomes unstable after running random testing for a while.

Those nondeterministic behaviors contradict the assumption of the algorithm that the method sequences in the pools can be extended and slow down the generation significantly if they happen. To remedy the effect of nondeterministic behaviors, feedback-controlled generation algorithm still needs to restart the JVM and reset pools periodically. We call this global resetting.

5. EVALUATION

To answer RQ2 ("Can our proposed technique lessen the dependency on random seeds and improve the overall performance of test generation?"), we evaluated our technique through a sequence of experiments[4]. Firstly, we confirmed the validity of our heuristics on the model SUT. The feedback-controlled generation algorithm uses two heuristic functions: *score* and *uniqueness*. We compared these two functions with random selection and random deletion strategy in order to prove that our heuristic functions can guide the generation to effective directions. We also evaluated the effect of global resetting and show that the combination of the feedback-controlled generation algorithm and global resetting outperforms other settings. Then, we surveyed the relevance of algorithm parameters. Our algorithm uses three parameters: *INP*, *MNP*, and resetting period. We show that these parameters are not relevant to the results. Our algorithm can adapt the direction irrelatively to these parameters, so end users do not need to adjust the parameters carefully. Finally, we evaluated our technique on real-world application libraries. We show that our technique can produce tests with higher coverage than the original feedback-directed random test generation. Throughout this section, we conducted the experiments in the same environment used in Section 2. For each experiment, we sampled results with ten different random seeds unless otherwise mentioned and calculated the median (Med.) and interquartile range (IQR) of coverage score. The median of coverage score represents the average score of each algorithm or configuration, whereas the interquartile range represents the variance of results.

[4]The implementation and experimental data can be downloaded from `http://www.klazz.net/pub/issta2015/`

Table 1: Coverage scores with or without heuristics

		statement		branch	
Select	Delete	Med.	IQR	Med.	IQR
random	random	53.1	12.2	22.1	13.0
score	random	60.9	9.8	32.8	12.3
random	uniqueness	55.5	7.6	25.7	8.7
score	uniqueness	61.2	4.3	32.7	4.4

Table 2: Coverage scores with or without global resetting

		statement		branch	
Algorithm	Reset	Med.	IQR	Med.	IQR
directed	none	47.8	11.9	17.0	10.0
directed	100 sec.	63.2	1.4	36.2	1.7
controlled	none	61.2	4.3	32.7	4.4
controlled	600 sec.	67.1	1.6	40.6	2.0

5.1 Validity of Score and Uniqueness Heuristics

First, we confirm the validity of our two heuristic functions *score* and *uniqueness*. We compared four settings: each selects pool at random or by using *score* and deletes the pool at random or by using *uniqueness*. We conducted an experiment with the model SUT, Apache Commons Collections 4.0 using parameters $INP = 10$ and $MNP = 100$. We did not use global resetting, so the effects of *score* and *uniqueness* are emphasized. The coverage statistics after 3600 seconds are listed in Table 1.

From the table, we can say that *score* function increases the average coverage and *uniqueness* function improves the variance of coverage. The average scores of both statement coverage and branch coverage are increased by *score*. By using *uniqueness*, the variance of coverage scores decreases. This result indicates that *score* can select pools that increase the coverage on average but becomes helpless when the algorithm falls into over-directedness. This is because *score* does not consider the overlap of directions. The *uniqueness* heuristic can fix over-directedness by removing pools with similar contents and decrease the variance of results. Though the variance is reduced by *uniqueness*, some seeds still perform much worse than other seeds. This is considered to be a consequence of nondeterministic behavior of SUT and the testing environment that will be overcome by global resetting.

5.2 Validity of Global Resetting

Global resetting is necessary to handle the nondeterministic behavior of SUT and the testing environment. We measured the effect of global resetting with two algorithms: the original feedback-directed generation algorithm and our feedback-controlled generation algorithm. We again used the model SUT and parameters $INP = 10$ and $MNP = 100$.

We used the resetting period 100 seconds for the feedback-directed generation algorithm, because 100 seconds is the default timeout of Randoop. For the feedback-controlled generation algorithm, we used a longer resetting period of 600 seconds, so that pool deletion would occur 11 times during a run. We discuss the relevance of the resetting period later. Coverage statistics after 3600 seconds are listed in Table 2. We can see that global resetting improves the average score and variance of results. Even if the generation is stuck due to the nondeterministic behaviors, it can be recovered by restarting the whole system.

5.3 Relevance of Parameters

In the experiments above, we confirmed the validity of our heuristic functions and global resetting. We now review the relevance and significance of parameters over the results using the model SUT.

Firstly, we investigated the relevance of the parameter *MNP*. We fixed *INP* to one and resetting period to 600 seconds and changed *MNP* among 10, 100, and 500. The coverage statistics after 3600 seconds are shown in Table 3. The results of $MNP = 10$ are slightly better than those of other configurations, and those of $MNP = 500$ are slightly worse, but the difference of median scores is under 3%. We can conclude that the algorithm is robust with respect to *MNP* and that *MNP* is not strongly relevant to the results.

Secondly, we investigated the relevance of the parameter *INP*. We fixed *MNP* to 100 and the resetting period to 600 seconds and performed experiments with $INP = 1, 10, 50$. The coverage statistics after 3600 seconds are in Table 4. The results do not seem to be strongly relevant to the parameter. This is because the set of pools is soon dominated by the newly added pools.

Finally, we investigated the relevance of the resetting period. We used parameters $INP = 1$ and $MNP = 10$, which performed slightly better than the other parameters tested in the above experiments. We conducted an experiment with six different periods: 20, 50, 100, 300, 600, and 1200 seconds. The results, listed in Table 5, indicate that extremely long resetting periods (600 and 1200 seconds) produce lower scores than shorter periods. With long resetting periods, there are fewer chances of global resetting (five for 600 seconds and only two for 1200 seconds), which may lead to the lower results. The shortest resetting period, 20 seconds, performs slightly lower than moderate periods (50, 100, and 300 seconds). The moderate resetting periods produce similar results. The resetting period has a stronger relevance on results than the other two parameters: *INP* and *MNP*. However, its sweet spot is large, and the results are not very sensitive to the parameter in the spot. Though the burden to find this region is left to end users, it does not require sensitive parameter tuning.

5.4 Real Libraries

From the experiments above, we analyzed the characteristics of our feedback-controlled random testing algorithm on the model SUT. Now, we review its generality by conducting experiments on eight different real-world application libraries. We set the parameters as $INP = 1$, $MNP = 10$, and the resetting period 100 seconds. Table 6 lists the libraries used in our experiments. We selected the test target libraries from MVNRepository, which is a catalog of Java libraries published to maven repositories. It provides libraries' category and popularity information. We selected eight popular libraries from seven different categories. We selected one library for each category. One exception is the Core Utilities category. We selected Guava and Commons Lang, because both are top-ranked libraries throughout all categories.

To evaluate our proposals and compare them with the original feedback-directed random testing, we conducted testing with three configurations.

`baseline`: The original feedback-directed generation algorithm is used, and no reset occurs during the whole time budget.

`reset`: The original feedback-directed generation algorithm is used, but global resetting occurs every 100 seconds. This is to assess the effect of resetting feedback.

`control`: Our feedback-controlled generation algorithm is used, and global resetting occurs every 100 seconds. This is to evaluate our feedback-controlled generation algorithm.

Experiments were conducted on eight libraries, each with the three configurations above. We used 30 different random seeds for these experiments. The coverage statistics are listed in Table 7. We sampled at two points (60^5 and

Table 3: Coverage scores with different *MNP* values

			statement		branch	
INP	*MNP*	Reset	Med.	IQR	Med.	IQR
1	10	600 sec.	68.5	1.8	41.7	2.4
1	100	600 sec.	68.2	2.1	41.3	2.4
1	500	600 sec.	67.1	1.5	40.1	1.5

Table 4: Coverage scores with different *INP* values

			statement		branch	
INP	*MNP*	Reset	Med.	IQR	Med.	IQR
1	100	600 sec.	68.2	2.1	41.3	2.4
10	100	600 sec.	67.1	1.6	40.6	2.0
50	100	600 sec.	67.8	0.9	41.2	1.0

Table 5: Coverage scores with different resetting periods

			statement		branch	
INP	*MNP*	Reset	Med.	IQR	Med.	IQR
1	10	20 sec.	69.8	1.1	43.5	0.7
1	10	50 sec.	70.7	0.4	44.3	0.9
1	10	100 sec.	70.8	0.8	43.9	0.9
1	10	300 sec.	70.3	1.2	43.8	0.8
1	10	600 sec.	68.5	1.8	41.7	2.4
1	10	1200 sec.	65.9	3.2	38.8	4.4

[5]We omitted $t = 60$ for reset, because reset is essentially the same as baseline before the first global resetting at $t = 100$.

Table 6: Libraries used in experiments

ID	Name Ver.	Category	#File	#Line	#Statement	#Branch
collections	Commons Collections 4.0	Collections	286	58186	8263	1965
lang	Commons Lang 3.3.2	Core Utilities	132	66628	9942	3090
guava	Guava 18.0	Core Utilities	469	129249	20642	3740
math	Commons Math 3.3	Math Libraries	900	202839	30679	8414
codec	Commons Codec 1.9	Base64 Libraries	56	13948	2437	611
gson	Gson 2.2.4	JSON Libraries	63	12216	2529	670
h2	H2 Database Engine 1.3.176	Embedded SQL Databases	472	158926	44859	10913
jetty	Jetty Server Core 9.2.6	Web Servers	103	32316	7379	1848

3600 seconds) in order to compare short-term and long-term performance of each configuration. For each configuration, we calculated the median and interquartile range of coverage score. For reset and control, we also conducted a Wilcoxon rank-sum test with the baseline and calculated the p-value. If the result has a significant difference (here, $p < 0.01$), we show the difference as an increase over baseline, which is calculated by $\frac{median_c - median_b}{median_b}$ where $median_c$ and $median_b$ represent the medians of the configuration and baseline, respectively. Changes in the median of coverage are illustrated in Figure 8. The effect of resetting and feedback-controlling seems to differ depending on the target library. There are three patterns.

Large Utility Libraries. Commons Collections, Commons Lang, Guava, and Commons Math are representatives of this pattern. For these libraries, the branch coverage of reset and control exceeds 78% (lang) to 204% (math) of the coverage of baseline after 3600 seconds. As we can see from Figure 8, reset and control have different curves. At first, control outperforms reset. However, the differences become smaller with time. For Guava, time $>$ 1000, the ranking changes and reset becomes the best strategy, though the difference is only 2.5% after 3600 seconds. This is considered to be because the feedback-controlled generation algorithm makes an assumption on the form of *score* function. It de-prioritizes pools with lower *score* at first and limits the possibility to hit an uncovered region for the sake of rapid coverage gain. Thus, if the time budget is limited, the feedback-controlled generation algorithm fits well, and if there is plenty of time, repeatedly running the original feedback-directed generation algorithm is an option.

Small Libraries. For this pattern, the relevance of reset and feedback-control is not apparent. Commons Codec and Gson are typical libraries in this pattern. Note that for this kind of library, the original algorithm performs better and seems to have less room for improvement than the large libraries above. Though the coverage scores converge to the same level after 3600 seconds, the speed of the increase differs for the first 60 seconds. This indicates that even for this kind of library, controlling feedback may be beneficial if the time budget is limited.

Configuration Intensive Libraries. The three configurations did not show statically significant differences regarding coverage for this pattern. This pattern includes H2 and Jetty. These libraries need careful configuration to work properly: H2 needs database configurations, and Jetty needs server settings. In addition, Jetty requires servlet implementations. Therefore, to test these libraries effectively, developers have to configure them and provide implementations for testing. Because the feedback-directed or feedback-controlled generation algorithm does not support functionality for automatic configuration generation or automatic stub generation, neither could test these configuration-intensive libraries effectively.

Overall, our feedback-controlled generation algorithm can speed-up the increase of coverage by prioritizing pools. However, the control is not fully adaptive and sacrifices long-term effectiveness for short-term efficiency. This is a point we want to improve in the future, but the current algorithm can perform much better than the other algorithm when the time budget is limited. When the time budget is not limited, the feedback-directed generation algorithm with resetting is a good option, but there seems to be no merit in running the feedback-directed generation algorithm for a long time. One drawback of feedback-directed or feedback-controlled algorithm is that it lacks the capability of configuration generation or stub generation. They require efforts from developers when testing configuration-intensive libraries. From this experiment, it is unknown which algorithm performs better with these configuration-intensive libraries when proper configurations and stubs are provided.

5.5 Threats to Validity

5.5.1 Internal Threats to Validity

Random testing is an inherently stochastic technique. To make things worse, as mentioned in Section 4, both SUT and the testing environment have nondeterministic behaviors. These stochastic and nondeterministic behaviors make it difficult to validate random test generation algorithms. To validate our technique, we took 30 different random seeds and calculated the median and interquartile range of coverage score. However, there remains a possibility that the sampled results are biased from the true distribution of all possible results.

5.5.2 External Threats to Validity

The results of test generation depend on SUT. To validate the generality of our results, we choose eight different SUT from seven different categories. Each SUT is a popular library according to MVNRepository and considered to be high quality software. However, the results are unknown when our technique is applied to libraries that have other categories or are low quality. We also limited our test target to application libraries, which are easier to test automatically. Testing end-user applications is out of the scope of this paper.

5.6 Limitation

This research only deals with the test effectiveness in terms of statement or branch coverage. The actual capacity of fault

Table 7: Coverage statistics of eight different popular libraries

Statement Coverage

		baseline		reset				control			
ID	Time	Med.	IQR	Med.	IQR	p	Inc.	Med.	IQR	p	Inc.
collections	60	24.6	3.0	-	-	-	-	44.0	2.3	< 0.01	78%
	3600	45.5	16.8	63.6	1.2	< 0.01	39%	70.6	0.8	< 0.01	55%
lang	60	42.2	12.4	-	-	-	-	58.5	6.4	< 0.01	38%
	3600	62.4	3.4	75.3	1.2	< 0.01	20%	76.6	0.6	< 0.01	22%
guava	60	18.7	4.7	-	-	-	-	23.1	3.4	< 0.01	23%
	3600	27.7	9.6	42.2	0.7	< 0.01	52%	40.3	0.6	< 0.01	45%
math	60	14.3	5.5	-	-	-	-	21.2	5.7	< 0.01	48%
	3600	25.3	14.0	49.9	1.4	< 0.01	96%	50.4	1.0	< 0.01	98%
codec	60	73.1	1.7	-	-	-	-	75.2	0.6	< 0.01	2%
	3600	80.5	0.5	81.5	0.4	< 0.01	1%	81.9	0.3	< 0.01	1%
gson	60	49.5	3.2	-	-	-	-	55.8	1.2	< 0.01	12%
	3600	60.6	1.2	63.6	0.8	< 0.01	4%	63.3	0.5	< 0.01	4%
h2	60	17.7	4.8	-	-	-	-	16.3	9.7	0.18	-
	3600	27.2	5.9	33.8	4.0	< 0.01	24%	28.2	0.7	0.12	-
jetty	60	18.5	2.5	-	-	-	-	20.2	1.2	< 0.01	8%
	3600	28.5	1.2	29.5	6.4	< 0.01	3%	28.5	1.7	0.98	-

Branch Coverage

		baseline		reset				control			
ID	Time	Med.	IQR	Med.	IQR	p	Inc.	Med.	IQR	p	Inc.
collections	60	6.0	1.5	-	-	-	-	14.1	1.9	< 0.01	136%
	3600	16.9	10.8	36.0	1.6	< 0.01	112%	44.4	0.8	< 0.01	162%
lang	60	16.1	6.6	-	-	-	-	35.1	6.8	< 0.01	118%
	3600	33.0	5.3	56.1	0.5	< 0.01	69%	58.8	0.4	< 0.01	78%
guava	60	6.7	2.0	-	-	-	-	10.0	2.5	< 0.01	50%
	3600	12.3	7.6	30.0	0.8	< 0.01	144%	27.5	0.6	< 0.01	123%
math	60	5.6	2.9	-	-	-	-	9.8	3.7	< 0.01	75%
	3600	11.4	7.4	33.6	1.4	< 0.01	195%	34.7	1.1	< 0.01	204%
codec	60	45.9	2.4	-	-	-	-	51.1	1.1	< 0.01	11%
	3600	61.4	0.7	64.0	0.7	< 0.01	4%	65.1	0.8	< 0.01	6%
gson	60	28.3	4.4	-	-	-	-	36.0	0.9	< 0.01	27%
	3600	41.9	1.2	44.5	0.6	< 0.01	6%	44.6	0.9	< 0.01	6%
h2	60	4.8	2.6	-	-	-	-	5.0	3.3	0.53	-
	3600	10.6	3.4	16.3	2.0	< 0.01	54%	12.3	0.3	0.01	-
jetty	60	2.7	1.1	-	-	-	-	3.4	1.0	< 0.01	29%
	3600	10.8	0.9	12.2	0.7	< 0.01	12%	12.2	0.4	< 0.01	12%

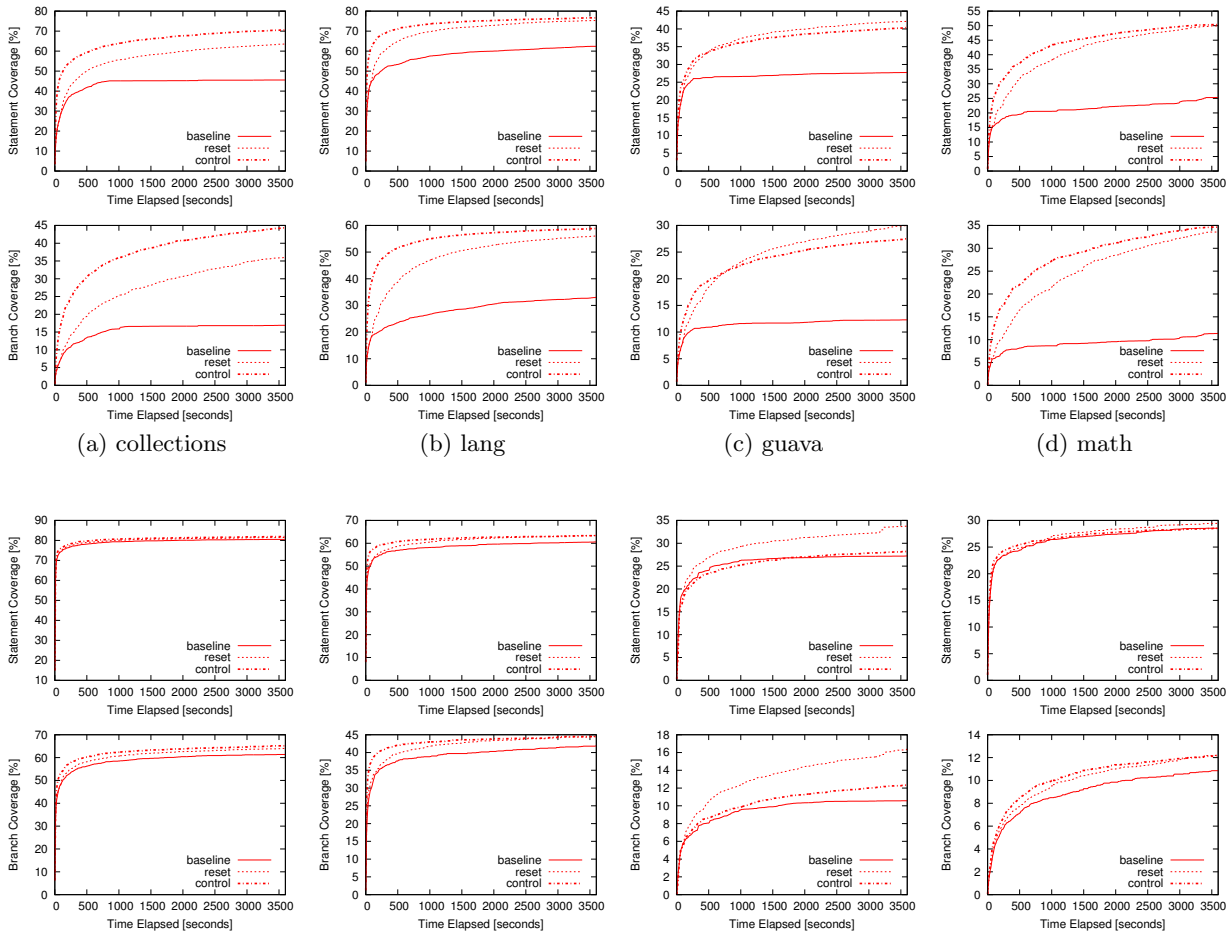
detection depends much on test oracles used with test generators. The relationship between coverage and fault detection is a controversial topic, and there is an ongoing discussion on how coverage is correlated with test effectiveness [11, 13]. For this research, we treat coverage as a goal in order to exclude the influence of test oracles and measure the ability for test generation.

6. RELATED WORK

When conducting random testing, the diversity of generated tests matters. Adaptive random testing [2, 3] aims to improve diversity by measuring distances of candidate inputs and sampling the most distant input from already tested inputs. It was originally proposed for numerical inputs, which have natural distance metrics, but Ciupa et al. proposed a distance function for objects in an object-oriented programming language [4]. Adaptive random testing can achieve higher test effectiveness than naive random testing with the same number of test inputs generated. However, the heavy computation cost that comes from calculating distances makes adaptive random testing practically impossible [1]. A state-of-the-art adaptive technique [21] overcomes this weakness by leveraging the geometric structure of input space to speed up generation, but it assumes only numeric inputs and is not simply applicable to object-oriented programming languages. We also adopt adaptive approach in deleting pools. Our approach compares coverage overlaps instead of pairwise distances. We showed time-effectiveness of our approach by experiments with a fixed time budget rather than a fixed test size.

Search-based test generation is another kind of randomized technique. It starts from random seed sequences and evolves them by using meta heuristics like hill climbing, simulated annealing, or a genetic algorithm (GA) [15], instead of using feedback. EvoSuite [8] is a state-of-the-art search-based testing tool using GA. It proved its usability in an experiment on 100 randomly sampled open source projects from Sourceforge [6, 7]. We formulated the problem of finding an appropriate amount of feedback as a search problem and used search techniques to solve the problem.

There have been several studies that have improved the effectiveness of feedback-directed or general random test-



(a) collections (b) lang (c) guava (d) math

(e) codec (f) gson (g) h2 (h) jetty

Figure 8: Average coverage scores of eight different popular libraries

ing. For example, MSeqGen [22] assists random testing by statically mining method sequences from source code. RecGen [28] analyzes object field access to recommend sequences. GenRed [14] extends feedback-directed generation by introducing input on demand creation and coverage-based method selection. Swarm testing [12] switches between configurations to generate peculiar test inputs. These techniques do not directly target the value pool in feedback-directed random generation and can be combined with our approach.

Besides random testing, dynamic symbolic execution (concolic execution) [10, 20] is a promising technique to generate software tests. Pex [23] is a widely used test generation tool based on dynamic symbolic execution. Dynamic symbolic executions inherently require seed sequences to instruct. Manually written parameterized unit tests [24] were used for this purpose in common, but a recent study [26] leverages feedback-directed random testing to create seed sequences. Garg et al. also combined feedback-directed random testing and dynamic symbolic execution to achieve higher test coverage and fault detection [9]. For these studies, time budget for random testing is limited because it is an aid for time-consuming dynamic symbolic execution. Our feedback-controlled generation algorithm can outperform the feedback-directed generation algorithm, especially in a limited time budget. Thus, our algorithm fits well as a fully compatible substitution of feedback-directed generation algorithm for these investigations.

7. CONCLUSION

In this paper, we analyzed the characteristics of feedback-directed random testing and found that excessive feedback over-directs the test generation and limits the diversity of tests generated. We proposed an algorithm, called feedback-controlled random test generation, which controls the amount of feedback in order to promote diversity. Experiments on eight popular application libraries indicate that our approach is highly effective in terms of test coverage for large-scale utility libraries or with a limited time budget.

8. REFERENCES

[1] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, pages 265–275, 2011.

[2] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case

diversity. *J. Syst. Softw.*, 83(1):60–66, Jan. 2010.

[3] T. Y. Chen, H. Leung, and I. Mak. Adaptive random testing. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321 of *Lecture Notes in Computer Science*, pages 320–329. 2004.

[4] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 71–80, 2008.

[5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS'11, pages 627–638, 2011.

[6] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 178–188, 2012.

[7] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *Proceedings of the 2013 IEEE 6th International Conference on Software Testing, Verification and Validation*, ICST'13, pages 362–369, 2013.

[8] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, Feb. 2013.

[9] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for c/c++ using concolic execution. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 132–141, 2013.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[11] 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.

[12] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA'12, pages 78–88, 2012.

[13] 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.

[14] H. Jaygarl, K.-S. Lu, and C. K. Chang. Genred: A tool for generating and reducing object-oriented test cases. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference*, COMPSAC '10, pages 127–136, 2010.

[15] P. McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[16] 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.

[17] 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.

[18] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 288–298, 2012.

[19] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE'11, pages 23–32, 2011.

[20] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE'13, pages 263–272, 2005.

[21] A. Shahbazi, A. F. Tappenden, and J. Miller. Centroidal voronoi tessellations—a new approach to random testing. *IEEE Trans. Softw. Eng.*, 39(2):163–183, Feb. 2013.

[22] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE'09, pages 193–202, 2009.

[23] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, 2008.

[24] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE'13, pages 253–262, 2005.

[25] J. Yi, D. Qi, S. H. Tan, and A. Roychoudhury. Expressing and checking intended changes via software change contracts. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA'13, pages 1–11, 2013.

[26] C. Zhang, A. Groce, and M. A. Alipour. Using test case reduction and prioritization to improve symbolic execution. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 160–170, 2014.

[27] S. Zhang, Y. Bu, X. S. Wang, and M. D. Ernst. Dependence-guided random test generation. *CSE 503 Course Project Report, University of Washington*, May 2010.

[28] W. Zheng, Q. Zhang, M. Lyu, and T. Xie. Random unit-test generation with mut-aware sequence recommendation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE'10, pages 293–296, 2010.