Feedback-directed Random Test Generation

Authored by C. Pacheco et al. and originally presented at ICSE 2007

Juyoung Jeon

July 3 2018

Motivation

- Unit test cases check the correctness of a certain unit (module) in a target program

- For rigorous checking, a set of diverse test cases that check comprehensive behaviors of a target unit is needed

- It is difficult for human to come up with comprehensive unit test cases (intentionally randomized test cases)

Background: Java Unit Test Cases (1/2)

- A Java unit test case runs a Java method with an input and then checks if the method results the expected output
  - target unit: a public method on a Java object
  - input (determinant): an input to a public method
    - target object state
    - method arguments
  - output (produced result)
    - return value
    - any thrown exception
    - target object state

Background: Java Unit Test Cases (2/2)

- A Java unit test case can be represented as a method sequence
    - target unit: last method call in the sequence
    - input: a sequence of method calls up to the last method call
        - an object is created by a constructor method call
        - an object can be used as an argument to a method call on another object
        - an object is updated through a sequence of method calls
    - output: an assertion on the return value of the last method call

Ex. Apache Commons Collection CircularFifoQueue

- Target program: `CircularFifoQueue`
  - 3 constructors and 23 public methods where 9 are inherited from `AbstractCollection`
  - 797(= 394 + 403) Lines and 97 (= 38+ 59) Branches

- Test cases targeted for method `add()`

```java
public boolean add(final E element) {
    if (null == element)
    throw new NullPointerException
    ( "Attempted to add null object to queue");
    if (isAtFullCapacity())
        remove();
    elements[end++] = element;
    if (end >= maxElements)
        end = 0;
    if (end == start)
        full = true ;
    return true ;
}
```

```java
@Test
public void testAddSingleElement() {
    CircularFifoQueue t0 = new CircularFifoQueue(5);
    t0.add(0);
    Assert.assertArrayEquals(t0.toArray(), {0}); }

@Test
public void testAddElementsOverflow(){
    CircularFifoQueue t0 = new CircularFifoQueue(5);
    for(int i =0; i<5; i++)
        t0.add(i);
    t0.add(5) ;
    Assert.assertArrayEquals(t0.toArray(),
        {1,2,3,4,5}); }

@Test (expected = NullPointerException.class)
public void testAddNull() {
    CircularFifoQueue t0 = new CircularFifoQueue();
    t0.add(null); }
```

Randoop, a random unit test generator

- Randoop randomly generates unit test cases for a given Java program
    - generate random method sequences
- Input: a set of target Java classes
- Output: two sets of JUnit test cases (JUnit test methods)
    - Failing tests (error-detection test cases)
        - the generated test cases that throw uncaught exceptions
        - these report potential errors of the target classes
    - Passing tests (regression test cases)
        - the generated test cases that do not throw any uncaught exception
        - these can be useful for making sure code contracts are preserved along the target program revisions

Tutorial

- bit.ly/2KJHOen

Ex. Revisit CircularFifoQueue with Randoop

- Generate 500, 1000, 1500, 2000, 2500, 3000 test cases using Randoop

	500	**1000**	**1500**	**2000**	**2500**	**3000**
Avg time	6.8	12.2	17.8	24	29.1	35.4
Line cov (circular/abstract collection) (%)	74/75.5	76/85.8	76/86.8	76/88.2	76/88.2	76/88.2
Branch cov (%)	72/62.6	75/76.8	75/79.1	75/80.2	75/83	75/83

Automated Unit Test Generation Process

1. **initialize a set of test cases $\boldsymbol{T}$ as $\boldsymbol{N}$ 0-length test cases, and $\boldsymbol{T_f}$ as empty**

2. **create a new test case**
   1. concatenate a random number of the existing test cases
   2. append a *feasible* method call to the concatenated test case
      - a feasible method is one that does not introduce any syntax error
3. continue the loop if the test case is structurally the same as any existing test case in T or T_{Fail}
4. run each newly generated test cases
5. add the failing test cases to T_{Fail}, non failing test cases to T
6. set redundant extended test cases of T as not extensible
7. repeat from Step 2 until the termination condition is satisfied

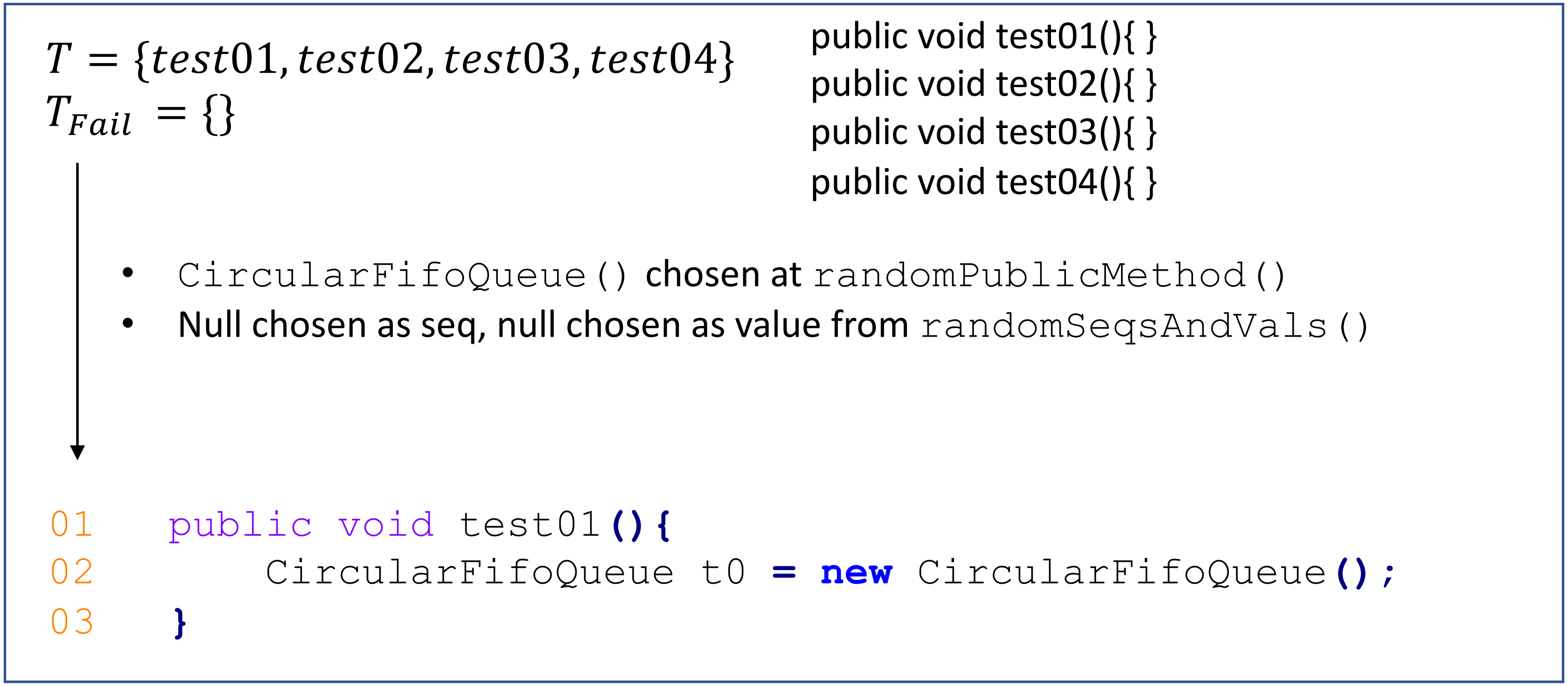
Step 2. Extending Method Sequences

- Produce a new method sequence by concatenating an existing method sequence and a new method call $m(...)$
  - append $m(...)$ for N times in a row at a given probability p
    - N: uniformly chosen at random between 2 and the given max value
- Create one new method sequence for zero or more test cases

Example

Pool of Types ={`CircularFifoQueue`, `Integer`, `String`}

N = 4



$T = \{test01, test02, test03, test04\}$

$T_{Fail} = \{\}$

public void test01(){ }
public void test02(){ }
public void test03(){ }
public void test04(){ }

- `CircularFifoQueue()` chosen at `randomPublicMethod()`
- Null chosen as seq, null chosen as value from `randomSeqsAndVals()`

```
public void test01(){
    CircularFifoQueue t0 = new CircularFifoQueue();
}
```

Example

T={TC0}

```
public void test01(){
    CircularFifoQueue t0 = new CircularFifoQueue();
}
```

`CircularFifoQueue()` chosen as method from `randomPublicMethod()`
Null chosen as seq, null chosen as value from `randomSeqsAndVals()`

```
public void testSentaticEqual(){
  CircularFifoQueue t1 = new CircularFifoQueue();
  }
```

Example

T={TC0, TC1}

```
public void test01(){
    CircularFifoQueue t0 = new CircularFifoQueue(); }
```

```
public void test02(){
    CircularFifoQueue t1 = new CircularFifoQueue();
    t1.add(Integer(3));
}
```

`get()` chosen as method from `randomPublicMethod()`

<TC0, TC1> chosen as seq, <t0, 2> chosen as value from `randomSeqsAndVals()`

```
public void testContractViolation(){
  CircularFifoQueue t0 = new CircularFifoQueue();
  CircularFifoQueue t1 = new CircularFifoQueue();
  t1.add(Integer(3));
  Object test = t0.get(2);
  }
```

Example

T={TC0, TC1}

```
public void test01(){
    CircularFifoQueue t0 = new CircularFifoQueue(); }
```

```
public void test02(){
    CircularFifoQueue t1 = new CircularFifoQueue();
    t1.add(Integer(3));
}
```

`peek()` chosen as method from `randomPublicMethod()`

<TC0, TC1> chosen as seq, <t1> chosen as value from `randomSeqsAndVals()`

```
public void testSemanticEqual(){
  CircularFifoQueue t0 = new CircularFifoQueue();
  CircularFifoQueue t1 = new CircularFifoQueue();
  t1.add(Integer(3));
  t1.peek();
  }
```

Example

T={TC0, TC1}

```
public void test01(){
   CircularFifoQueue t0 = new CircularFifoQueue(); }
```

```
public void test02(){
   CircularFifoQueue t1 = new CircularFifoQueue();
   t1.add(Integer(3));
}
```

`Add()` chosen as method from `randomPublicMethod()`

<TC0, TC1> chosen as seq, <t0, String(”Hi”)> chosen as value from `randomSeqsAndVals()`

```
public void testPass(){
  CircularFifoQueue t0 = new CircularFifoQueue();
  CircularFifoQueue t1 = new CircularFifoQueue();
  t1.add(Integer(3));
  t0.add(String(“Hi”));
}
```

Automated Unit Test Generation Process

1. initialize a set of test cases T as N 0-length test cases, and T_f as empty
2. create a new test case
3. **Continue the loop if the test case is structurally the same as any existing test case in T or T_{Fail}**
4. **run each newly generated test cases**
5. **Add the failing test cases to T_{Fail}, non failing test cases to T**
6. **Set redundant extended test cases of T as not extensible**
   - **An extended test case is redundant if the resulting target object state is equal (i.e., `equal()`) to that of the original test case (i.e., before the extension)**
7. repeat from Step 2 until the termination condition is satisfied
   - a termination condition is specified as an amount of time, the number of generated test cases, custom Stopper provided or IEventListener indicated to stop

Steps 3-6. Filtering Out Extended Test Cases

- Step 3. Do not add *syntactically redundant* test cases

- Step 5. Discriminate *failing* test cases
  - not worthwhile to extend/grow afterward

- Step 6. Set *semantically* redundant test cases as not extensible
  - check if all resulting objects are equivalent with the ones created before
    - Pitfall: miss errors of method calls that have the same abstract value but behaves differently on method calls

Example

```
public void testSentaticEqual(){
  CircularFifoQueue t1 = new CircularFifoQueue();
}
```

The new test case is structurally the same with existing test case test01.

Continue Loop

Example

```
public void testContractViolation(){
  CircularFifoQueue t0 = new CircularFifoQueue();
  CircularFifoQueue t1 = new CircularFifoQueue();
  t1.add(Integer(3));
  Object test = t0.get(2);
  }
```

The new test case is not structurally the same with any existing test case.

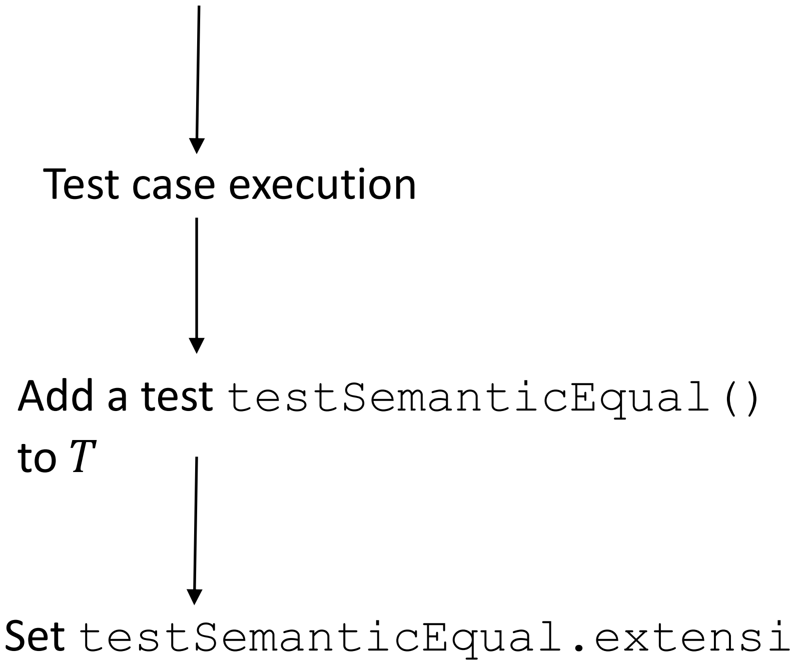
Test case execution

The new test case fails (`NoSuchElementException`)

Add a test case `testContractViolation()` to T_{Fail}

Example

```
public void testSemanticEqual(){
  CircularFifoQueue t0 = new CircularFifoQueue();
  CircularFifoQueue t1 = new CircularFifoQueue();
  t1.add(Integer(3));
  t1.peek();
  }
```



Test case execution

Add a test `testSemanticEqual()` to T

Set `testSemanticEqual.extensible = false`

The new test case is not structurally the same with any existing test case.

The new test case does not fail.

The new test case is semantically redundant (t0 and t1 is equivalent to that of t0 and t1 in test01)

Example

```
01 public void testPass(){
02   CircularFifoQueue t0 = new CircularFifoQueue();
03   CircularFifoQueue t1 = new CircularFifoQueue();
04   t1.add(Integer(3));
05   t0.add(String(“Hi”));
06 }
```

↓

The new test case is not structurally the same with any existing test case.

Test case execution

↓

The new test case does not fail.

Add a test `testPass()` to T

↓

The new test case is not semantically redundant (t0 after Line 05 is not equivalent with any object resulted by the existing test cases)

Set `testSemanticEqual.extensible = true`

Evaluation

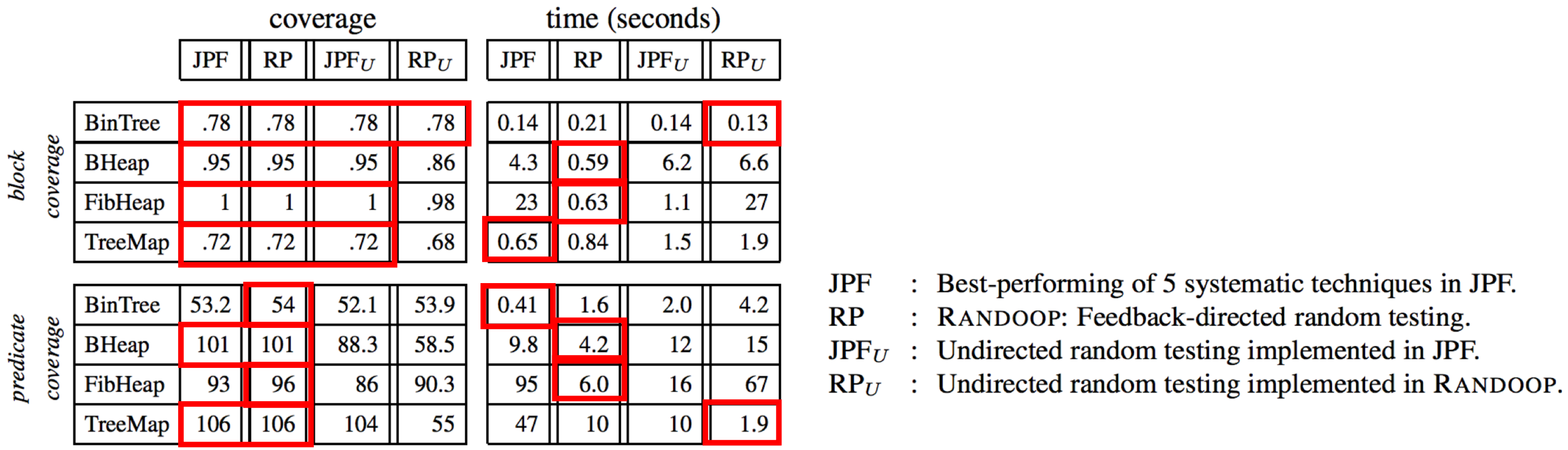
- Research questions

1. How much block/condition coverage can be achieved by Randoop test generations?
   - Randoop vs. a systematic testing tool JPF
   - 5 container libraries (e.g., heap, tree)
   - JPF : specified sequence length from 1 to 30/ randoop: 2min time limit / Undirected Random: sequence length up to 50

2. Do test cases generated for a buggy program with Randoop effectively reveal the bugs?
   - 14 widely used libraries (e.g., Java Util, Java Xml)
   - JPF: out of memory, randoop: 2min time limit(setting), Jcrasher: maximum possible parameter that limits number of method call that chains together

3. Do test cases generated for a correct program with Randoop effectively reveal the bugs in the later versions?
   - Java JDK on Sun1.5, Sun 1.6 Beta, IBM 1.5
   - Default Time limit : 2min

RQ1. How much coverage can be achieved by Randoop?



		coverage				time (seconds)			
		JPF	RP	JPF_U	RP_U	JPF	RP	JPF_U	RP_U
block coverage	BinTree	.78	.78	.78	.78	0.14	0.21	0.14	0.13
	BHeap	.95	.95	.95	.86	4.3	0.59	6.2	6.6
	FibHeap	1	1	1	.98	23	0.63	1.1	27
	TreeMap	.72	.72	.72	.68	0.65	0.84	1.5	1.9
predicate coverage	BinTree	53.2	54	52.1	53.9	0.41	1.6	2.0	4.2
	BHeap	101	101	88.3	58.5	9.8	4.2	12	15
	FibHeap	93	96	86	90.3	95	6.0	16	67
	TreeMap	106	106	104	55	47	10	10	1.9

JPF : Best-performing of 5 systematic techniques in JPF.
RP : RANDOOP: Feedback-directed random testing.
JPF_U : Undirected random testing implemented in JPF.
RP_U : Undirected random testing implemented in RANDOOP.

- Repeated each run ten times with different seeds; average reported
- Findings
  - BinTree, BHeap, TreeMap: Randoop achieved the same block and condition coverage as shape abstraction
  - FibHeap: Randoop achieved higher condition coverage than shape abstraction
  - Randoop generated test cases the fastest in four out of eight experiments and did not take longer than 10 seconds in all cases

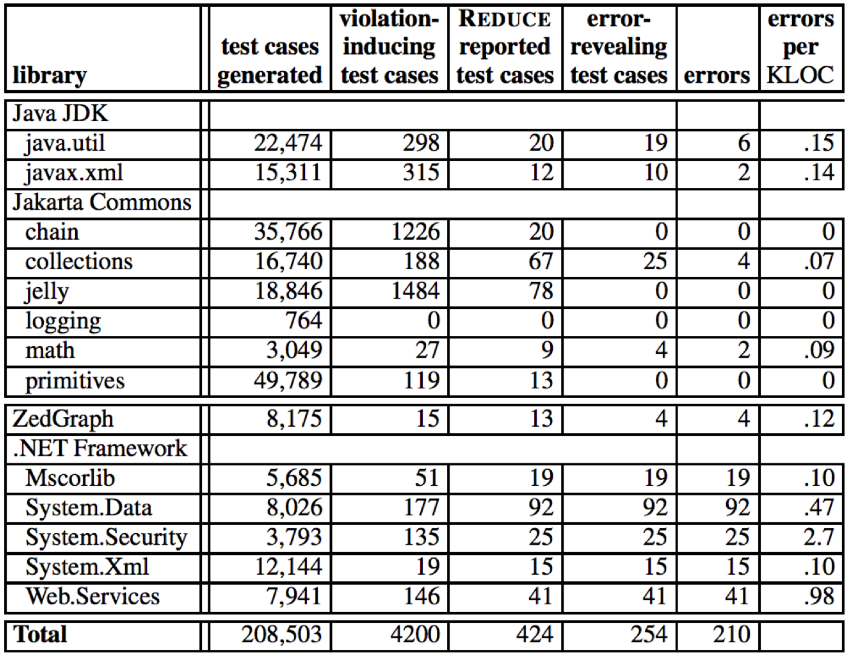
RQ1. How much coverage can be achieved by Randoop?

- Feedback-directed random generation
  - effective in generating complex test inputs
  - Competitive with systematic generation even when state space is larger (FibHeap, BHeap)
- Repetition of method calls was crucial
  - sequences with several element repetition reached predicates that were not reached those without repetition

RQ2. Do test cases generated for a buggy program with Randoop effectively reveal the bugs?

- Result
  - violation inducting: T_fail, Test Cases that violate contracts
  - reduced : Subset of failing tests
  - error-revealing : Tests that reveal errors from Reduced
  - errors (bugs) : Distinct errors

- Observation



library	test cases generated	violation-inducing test cases	REDUCE reported test cases	error-revealing test cases	errors	errors per KLOC
Java JDK						
java.util	22,474	298	20	19	6	.15
javax.xml	15,311	315	12	10	2	.14
Jakarta Commons						
chain	35,766	1226	20	0	0	0
collections	16,740	188	67	25	4	.07
jelly	18,846	1484	78	0	0	0
logging	764	0	0	0	0	0
math	3,049	27	9	4	2	.09
primitives	49,789	119	13	0	0	0
ZedGraph	8,175	15	13	4	4	.12
.NET Framework						
Mscorlib	5,685	51	19	19	19	.10
System.Data	8,026	177	92	92	92	.47
System.Security	3,793	135	25	25	25	2.7
System.Xml	12,144	19	15	15	15	.10
Web.Services	7,941	146	41	41	41	.98
Total	208,503	4200	424	254	210	

RQ2. Do test cases generated for a buggy program with Randoop effectively reveal the bugs?

- Systematic testing (JPF)
  - For all libraries JPF ran out of memory without reporting any errors
  - JPF samples **tiny, localized** portion of space
  - Randoop explores tiny fraction of enormous state space
  - Sparse, global sampling can reveal errors efficiently in large libraries

RQ2. Do test cases generated for a buggy program with Randoop effectively reveal the bugs?

- Undirected Random Testing
  - Ran randoop disabling user filters or contracts
    - Did not find any errors in java.util, javax.xml
    - Unable to create sequence that uncovered infinite loop in System.Xml
  - JCrasher (independent implementation of undirected random test generation)
    - Ran 639 sec
    - 698 failing test cases (randoop : 424)
    - 3 error revealing (randoop: 254)
    - One distinct error (randoop: 210)
    - Creates many **redundant and illegal inputs**

RQ3. Regression and compliance testing

- Find **inconsistencies** between different implementations of the same JDK (regression error)
  - Three comercial implementations : Sun JDK 1.5, Sun JDK 1.6 Beta2, IBM JDK 1.5

- Generate 41046 passing test cases for Sun JDK 1.5 and then run the test cases on Sun JDK 1.6 Beta2 and IBM JDK 1.5
  - 98 failures were found
    - Sun JDK 1.6 : 25 cases failed
    - IBM JDK 1.5 : 73 cases failed
    - 44 test cases revealed inconsistencies
      - 12 distinct bugs (errors) are found

Comparing Automated Test Generation Approaches

Feedback-directed Random Test Generation (Randoop)	**Unguided Random Test Generation**	**Systematic Testing (model checking- JPF)**
• Scalability • Sparse & global sampling • Less redundant and meaningless test cases	• Sparse & global sampling • Scalability • Simple implementation	• Dense & local sampling

Improvements to be made

- Test cases without redundancy
  - Example
    - Adding one element or two elements for CircularFifoQueue of size 5 will be two different states, but the consequence is the same

- Various Test Cases
  - Test cases from different search spaces
  - More complex operations?