CS5351 Software Engineering
2024/2025 Semester B

Program Testing, Part 2

Prof. Zhi-An Huang

Email: **huang.za@cityu-dg.edu.cn**

Our Fuzzing Exercise

Blue: No new coverage achieved
Green: New coverage achieved
Red: Crash the program

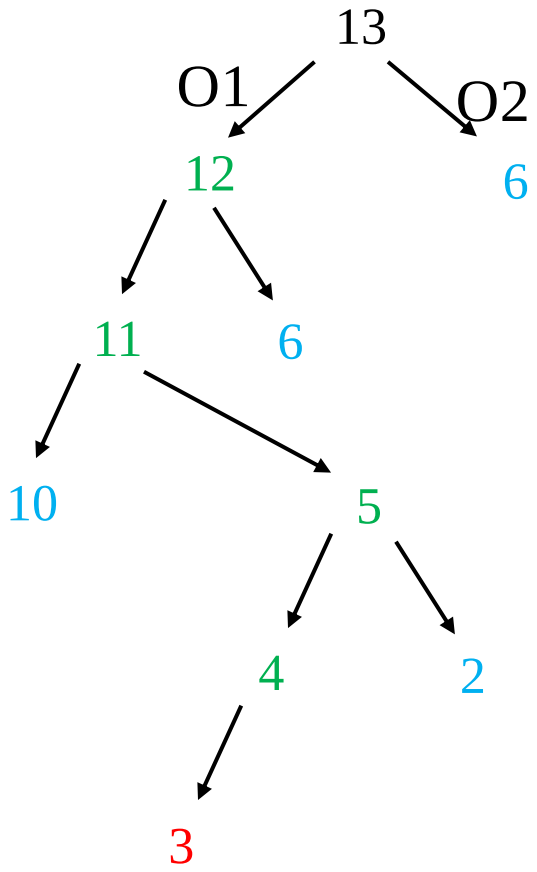
Queue: <13>

Queue: <13, 12>

Queue: <13, 12, 11>

Queue: <13, 12, 11, 5>

Queue: <13, 12, 11, 5, 4>



The input sequence is: <12, 6, 11, 6, 10, 5, 4, 2, 3>

The content of the Queue is: <13, 12, 11, 5, 4>

Fuzzing

- can generate system-level test cases as they generate inputs accepted by UI or `main()` as arguments.

- could not help developers when the integration and system testing have not been performed.

- Apart from manually crafting unit test cases, what kinds of *automated* facilities help developers deliver higher productivity and quality?

1. Concurrency Testing

- Programs can be multi-threaded and/or multi-process architecture
- These programs are nondeterministic
  - Running the same test case multiple times may give different outputs.
  - E.g., data race on x in the example on the right.

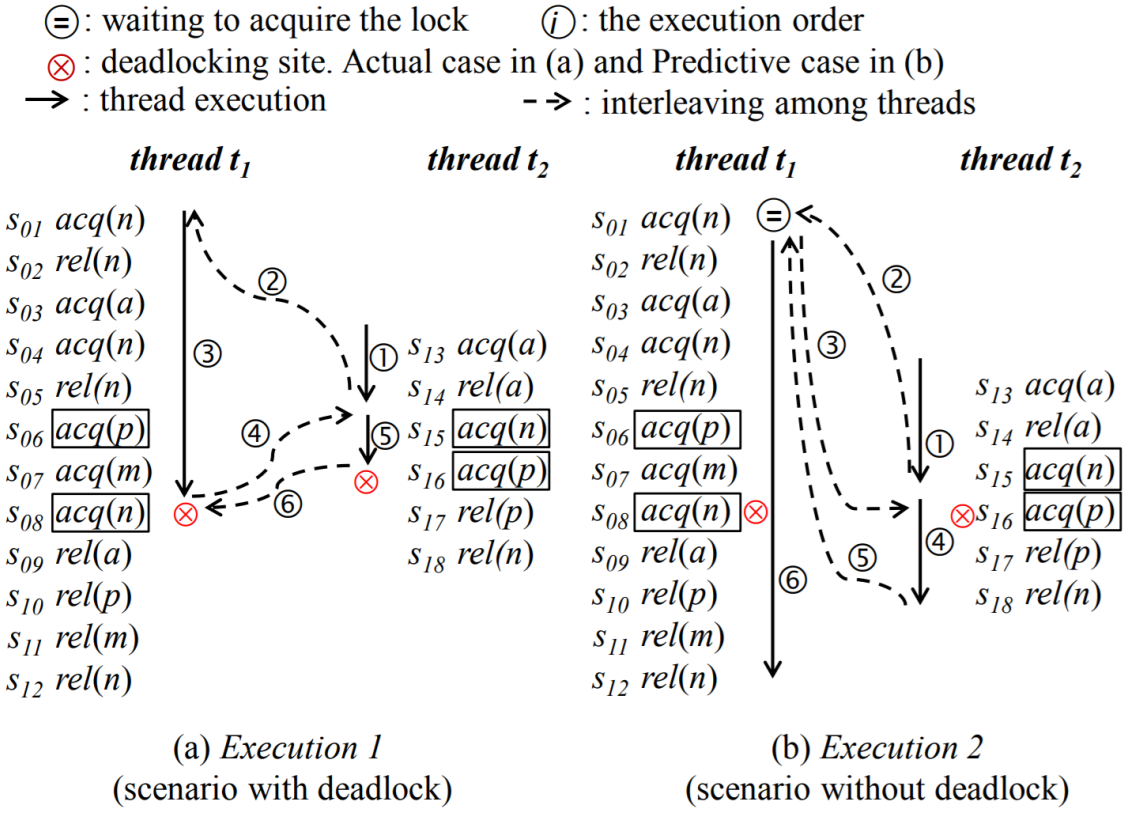
Thread T1	Thread T2

x=0; y=0;
while (x==0)
  y++;
 	
if (y%2)
  y++;
else
  y--;
	x++;

1. Concurrency Testing

- Strategy 1
  - Run the same test case many times without controlling how program threads are coordinated in each test execution.
    - Ineffective. If you have no other choices, just using it is better than none.
- Strategy 2
  - Run the same test cases multiple times with the use of a controller to control when threads execute their program statements relative to one another.
    - E.g., use calfuzzer to analyze each test execution. Slower
    - E.g., in routine regression testing, apply threadsanitizer https://github.com/google/sanitizers
    - E.g., in some research group, they use Intel Pin or IBM ASM for binary instrumentation.
- Concurrency testing: Can check individual program executions faster, can suppress false positives, limited in ability, false negative
  - Thread sanitizer

Example:
Examine Execution 2 to Generate Execution 1

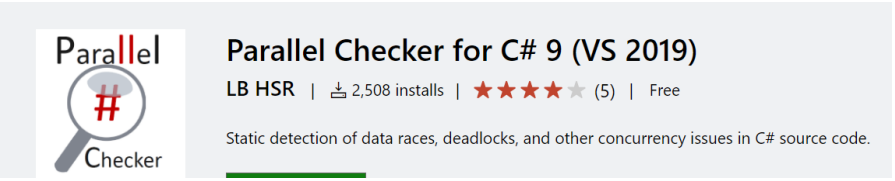


(a) *Execution 1*
(scenario with deadlock)

(b) *Execution 2*
(scenario without deadlock)

Figure 1. Example deadlock adapted from `JDBC Connector 5.0` [2] (Bug ID: 2147). The acronym `n`, `a`, `p`, and `m` are `Connection`, `Statement`, `ServerPreparedStatement`, and `Connection.Mutex`, respectively.

Examples only:



2. Static Analysis

- Static analysis is to inspect the code without running it.
- Many modern IDEs have incorporated static analysis to spot potential concurrency issues
  - E.g. https://devblogs.microsoft.com/cppblog/concurrency-code-analysis-in-visual-studio-2019/ , Coverity (https://en.wikipedia.org/wiki/Coverity): has a free cloud-based scanner for your open-source projects
- Static analysis is a routine task in large projects.
- Static analysis: Can check the whole codebase, applicable to check Library/Framework, slow, non-scalable, many false positives
- The mathematics are quite extensive. We will not introduce here.

3. Automated Test Case Generator

- The target is to produce some useful test cases (but without proper check on test outcomes)
  - E.g., Evosuite (https://www.evosuite.org/)
  - Still immature for industry use in general. (see Ref [12])
- Idea in Evosuite
  - Given a set of JUnit test methods, each of which contains a sequence of statements
  - Like fuzzing, apply a series of mutation operators to modify some statements (e.g., exchange statements across test methods, delete some statements, add some statement, change parameter value) to generate a new set of Junit test methods
  - No idea on whether a test method detects a program failure

Test Oracle

Test Oracle Problem

- Recalling from Evosuite's and fuzzing's limitations that they could not point out functional failures of the program being written.
  - Sometimes, we simply cannot get it: how do you know a Deep Learning engine produces a correct classification?

- We need to formulate either a *correctness criterion* or *an error condition* for test cases to know whether the program actually passes these test cases.
- This is called the ***test oracle problem***.

Test Oracle

- Generic Test Oracle [automated]
  - There are some errors that we know them being errors. These errors are general and applicable to all types of situations and programs (E.g., crash, deadlock.)
    - They are widely used in automated testing tools.
- Pattern-based Test Oracle [automated]
  - Sometimes, some code patterns in the code tend to be erroneous. They should be removed from the code if possible. They are formulated heuristically.
    - Many static analysis tools use them.
    - E.g., FindBugs for Java, pylint for Python, safety for checking requirement.txt for Python

Test Oracle

- Reference Test Oracle [automated]
  - For programs with clear specialized purposes, one knows the expected test results and test cases. For instance, to test for browser compatibility, we can run the predefine benchmark test suite available on the web.
- Regression Test Oracle [automated]
  - For programs with versioning, many test cases will be applicable to multiple program versions without revision. In these cases, the test oracle or output of a program version on a test case can be used to check the output of another program version on the same test case.

Test Oracle

- Program-Specific Test Oracle [manual]
  - Each program has its own purpose and functionality. There is no universally applicable test oracle to verify functional outputs fitting every program.
  - In a vast majority of cases, developers write their own Assertion Statements (e.g., assertEqual() in JUnit) or validate the outputs manually
    - There is no better option yet.
  - If developers have some knowledge about the application domain of a program, they can use their “common sense” to spot failures from testing.

Test Oracle

- Test Oracle across Multiple Executions
  - Suppose developers have some knowledge about the application domain of a program.
  - They will know something could be wrong by comparing multiple test results and the expectations on the relations on test results
  - It is known as ***Metamorphic Testing*** [automated]
  - Is experimentally used in testing Autonomous Driving Systems
  - Is used in startup

Google acquires GraphicsFuzz for its mobile graphics card benchmarking

KYLE WIGGERS @KYLE_L_WIGGERS AUGUST 6, 2018 9:51 AM

As GraphicsFuzz explains, its flagship product — ShaderTest GLES — uses shaders, or programs that run in the graphics pipeline and guide the rendering process, to detect bugs in drivers through a process it calls "metamorphic testing." Its reference shaders produce an image, after which

Idea of Metamorphic Testing

- Consider a deterministic program `P` that finds a shortest path from a directed graph `G1` from node `x` to node `y`.
- Test case `P(G1, x, y)` outputs a sequence `L1` of nodes in `G1` to go from `x` to `y`.

Test case part:

- By deleting some nodes (and their connecting edges) in `G1` where each of them is not in `L1`, we construct another graph `G2`.
- Test case `P(G2, x, y)` outputs `L2`.

Oracle part:

- Expectation: `L1 = L2`. If violated, at least one of the two test cases is a failed test case.

Metamorphic Relation

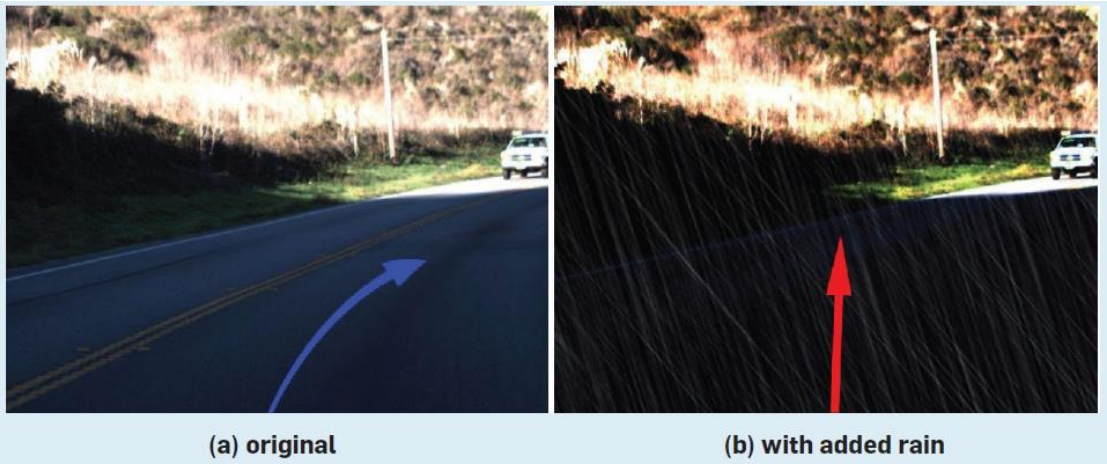
- An expected relation over multiple test inputs and test outputs to be held.
- Such an expected relation is called a **metamorphic relation**.
  - E.g., `P(G1, x, y) = L1 ∧ P(G2, x, y) = L2 ∧ L1 = L2`
  - The test case {`G1, x, y`} is called a **source test case**.
  - The test case {`G2, x, y`} is called a **follow-up test case**.

Simple Metamorphic Relations Could Be Effective to Detect Bugs

- A compiler `P` (e.g., the objective-c compiler) to generate the binary code `B1` from a source code `S1`.
- Suppose `t` is a test case of `S1`.
- Compile `S1`, run `B1(t)`, and then remove some statements in `S1` that `t` does not pass through them to construct a new program `S2`, compile `S2` to generate the binary code `B2`.
  - Expectation: `B1(t)` and `B2(t)` produce the same output.
- Thousands of bugs have been detected from the *gcc* compiler

MT for Driverless Car [11]

- MR: “the car should behave similarly under variations of the same input (such as the same scene under different lighting conditions).” [11]
  - “Using these MRs, they generated realistic synthetic images based on seed images. These synthetic images mimic real-world phenomena (such as camera lens distortions and different weather conditions). Using MT, together with a notion of neuron coverage (the number of neurons activated), the researchers found a large number of "corner case" inputs leading to erroneous behavior in three DNN models.”



Input pictures used in a metamorphic test revealing inconsistent and erroneous behavior of a DNN

Metamorphic Relation

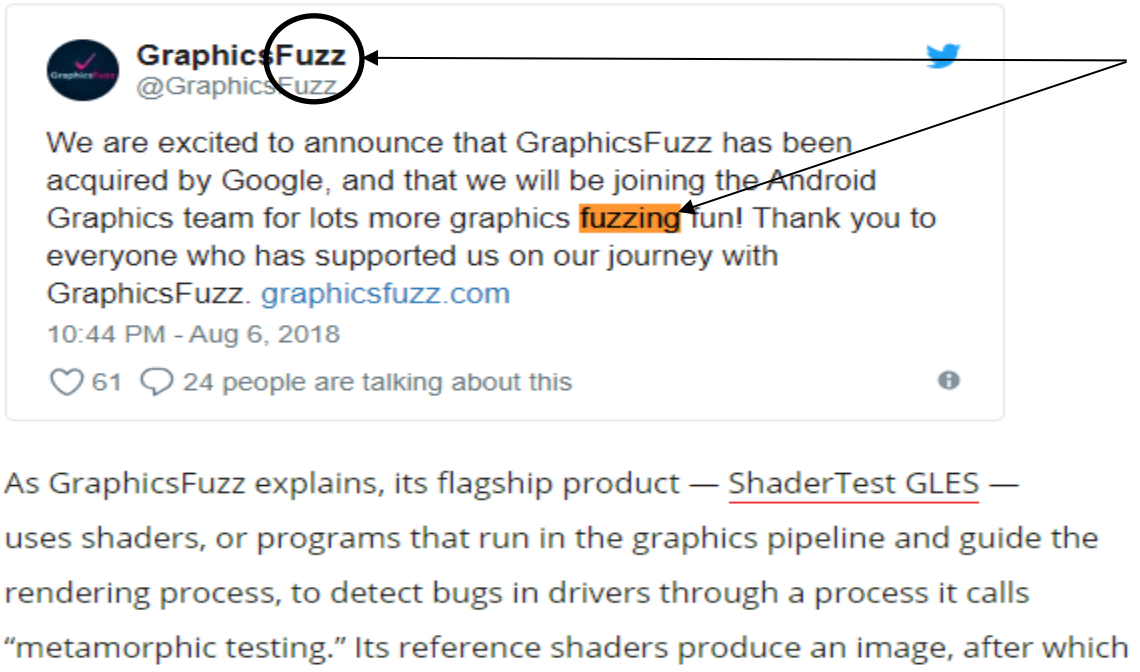
- Often, the “Common Sense in the application domain.”

- Metamorphic testing can be applied beyond Equivalent Inputs (and thus Equivalent Outputs).
- Developers knowing the application domains can formulate their own relations for their programs.
  - *e.g.*, For a person with a fixed income, spending less means saving more.
  - *e.g.*, Shipment by ship should take longer estimated time and lower cost than Shipment by flight.
  - *e.g.*, Buying more units of the same item should be charged more.

Fuzzing + Metamorphic Testing

- An effective use of both techniques

Metamorphic testing



Fuzzing

- Fuzzing to generate one test case as a source test case for metamorphic testing to follow up.

Fuzzing + Automated Test Oracle

- E.g., ContractFuzzer for testing Ethereum smart contracts

delegatecall with msg.data as its parameter. This makes an attacker can call any public function of _walletLibrary with the data of Wallet. So, the attacker calls the initWallet function (defined at line 10) of the _walletLibrary smart contract and become the owner the wallet contract. Finally, he can send the ether of the wallet to his own address to finish the attack. This attack has led to $30 million loss to the parity wallet users.

Table 1. Dangerous Delegate Call in Parity Wallet Contract

```
1  contract Wallet{
2      function() payable { //fallback function
3      if (msg.value > 0)
4          Deposit(msg.sender, msg.value);
5      else if (msg.data.length > 0)
6          _walletLibrary.delegatecall(msg.data);
7    }
8  }
9  contract WalletLibrary {
10 function initWallet(address[] _owners, uint _required, uint
   _daylimit) {
11     initDaylimit(_daylimit);
12     initMultiowned(_owners, _required);
13     }
14 }
```

Test Oracle for Dangerous DelegateCall

The test oracle ***DangerDelegateCall*** checks whether there is a ***deletegate*** call invoked during the execution of the current contract and that the function called by the delegate call is obtained from the input (e.g., msg.data) of the initial call to the contract. In another word, the test oracle checks whether the contract under test invokes a delegate call whose target function is provided by a potential attacker.

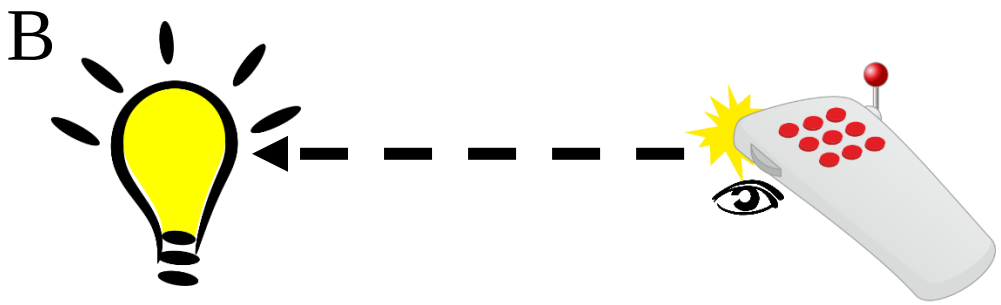
Table 5 Summary of Vulnerability Detected

Vulnerability Type	Number of Vulnerabilities	Percentage	True Positive Rate
Gasless Send	138	2.06%	100%
Exception Disorder	36	0.54%	100%
Reentrancy	14	0.21%	100%
Timestamp Dependency	152	2.27%	96.05%
Block Number Dependency	82	1.23%	96.34%
Dangerous Delegatecall	7	0.10%	100%
Freezing Ether	30	0.45%	100%
Total	459	/	/

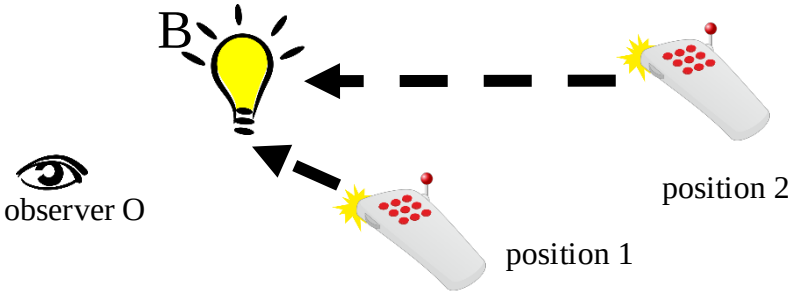
Example

MT and traditional testing find different kinds of bugs

This smart light bulb B ensures Y luminance at the remote-control position where Y is set in the sLight app on the control.



In traditional testing, we set sLight(Y) and measure Y through the remote control. Nonetheless, if you do it at home or in a factory, you may use alternative methods for testing, …



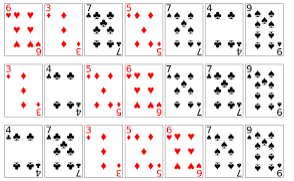
MR1: If position 2 > position 1,
then O observes B brighter when the remote is at position 2

MR2: If position 2 = position 1, then B looks similar in brightness from O when the remote is at either position.

How to implement MR2? [Just circle around the bulb]

Exercise 1 on Identification of Metamorphic Relations

Any metamorphic relations you can find to test the class X?



```
Class X {
   int cards[52]; // a deck of 52 distinct cards

   public X(int seed); // initialize the cards[] array
   // use seed to initialize the random number
   // generator for shuffling cards
   public void shuffle(int n);
    // randomly shuffle the deck of cards for n times
   public int draw(int i);
   // return cards[i], and cards[i] is deleted from cards[]
   // and the size of cards[] is reduced by 1
}
```

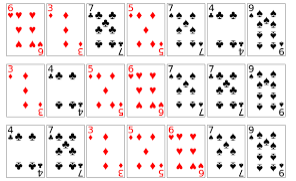
Possible Solution for Metamorphic Relations

- Each instance of X is deterministic as the seed is inputted when the instance is constructed to initialize the random number generator used to shuffle cards.
- MR1: For test cases with the same seed, the values returned by different positions should be different.

```
Random rand = new Random();
   int p1 = rand.nextInt(100) + 1; // get a value from 1 to 100
    int p2 = rand.nextInt(100) + 1; // get a value from 1 to 100
    int s = rand.nextInt(100) + 1;
   X x = new X(s);  x.shuffle(p1);
   X y = new Y(s);  y.shuffle(p2);
   // assertNotEquals(x.draw(0), x.draw(0));  // not MR
   int y1 = y.draw(0);
   int y2 = y.draw(0);
   int x1 = x.draw(0);
   assertTrue( x1 <> y1 || x1 <> y2);
```

Exercise 2 on Identification of Metamorphic Relations

Any metamorphic relations you can find to test the class Y?



```
Class Y {
   int cards[52]; // a deck of 52 distinct cards

   public X(); // initialize the cards[] array
   // internally initialize the random number
   // generator for shuffling cards
   public void shuffle(int n);
    // randomly shuffle the deck of cards for n times
   public int draw(int i);
   // return cards[i], and cards[i] is deleted from cards[]
   // and the size of cards[] is reduced by 1
}
```

Possible Solution for Metamorphic Relations

MR2. Two test cases using the same seed and shuffles the deck for p and q times, respectively. For each test case, call draw(0) 52 times and sum up the returned values. The sums computed for the two test cases should be the same.

```
Random rand = new Random();
int p = rand.nextInt(100) + 1; // get a value from 1 to 100
int s = rand.nextInt(100) + 1;
int sum_x = 0; X x = new X(s); x.shuffle(p);
for (int i=0;i<52;i++) sum_x += x.draw(0);
int sum_y = 0; X y = new X(s); x.shuffle(p*p);
for (int i=0;i<52;i++) sum_y += y.draw(0);
assertEquals(sum_x, sum_y);
```

This metamorphic relation is applicable to both Class X and Class Y.

There are many other relations.

References and Resources

1. Evosuite, available at http://www.evosuite.org/
   a) Command line tutorial: http://www.evosuite.org/documentation/tutorial-part-1
   b) Marven Integration: http://www.evosuite.org/documentation/tutorial-part-2
   c) Eclipse plugin, Jenkins plugin etc are available
   d) Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 416-419. DOI=http://dx.doi.org/10.1145/2025113.2025179
   e) Annibale Panichella and Urko Rueda Molina. 2017. Java unit testing tool competition: fifth round. In Proceedings of the 10th International Workshop on Search-Based Software Testing (SBST '17). IEEE Press, Piscataway, NJ, USA, 32-38. https://dl.acm.org/citation.cfm?id=3105434
   f) Fitsum Kifetew, Xavier Devroey, and Urko Rueda. 2019. Java unit testing tool competition: seventh round. In Proceedings of the 12th International Workshop on Search-Based Software Testing (SBST '19). IEEE Press, Piscataway, NJ, USA, 15-20. DOI: https://doi.org/10.1109/SBST.2019.00014
2. Javin. Paul. 10 Useful Unit and Integration Testing tools for Java Programmers. Available at https://dzone.com/articles/10-essential-testing-tools-for-java-developers

References and Resources

3. ASM https://asm.ow2.io/
   - Introduction of ASM: https://s3-eu-west-1.amazonaws.com/presentations2012/30_presentation.pdf
4. Safety https://github.com/pyupio/safety
5. Pylint https://www.pylint.org/
6. FindBugs http://findbugs.sourceforge.net/
7. Bo Jiang, Ye Liu, and W.K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18), September 3–7, Montpellier, France, 10 pages. https://doi.org/10.1145/3238147.3238177
   - Source code: https://github.com/gongbell/ContractFuzzer
8. Prasetya I.S.W.B. (2014) T3, a Combinator-Based Random Testing Tool for Java: Benchmarking. In: Vos T., Lakhotia K., Bauersfeld S. (eds) Future Internet Testing. FITTEST 2013. Lecture Notes in Computer Science, vol 8432. Springer, Cham
9. Fraser G. (2018) A Tutorial on Using and Extending the EvoSuite Search-Based Test Generator. In: Colanzi T., McMinn P. (eds) Search-Based Software Engineering. SSBSE 2018. Lecture Notes in Computer Science, vol 11036. Springer, Cham
10. Yan Cai, Shangru Wu, W. K. Chan: ConLock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs. ICSE 2014: 491-502
11. Metamorphic Testing of Driverless Cars. By Zhi Quan Zhou, Liqun Sun Communications of the ACM, March 2019, Vol. 62 No. 3, 61-67.